The Collected Works of Bjarne Stroustrup

Bjarne Stroustrup (born in Aarhus, Denmark, in 1950) is a Danish computer scientist, most notable for the creation and development of the C++ programming language. He is a visiting professor at Columbia University, and works at Morgan Stanley as a Managing Director in New York.

Bjarne Stroustrup

Stroustrup in 2010

Born	30 December 1950 (age 68) Aarhus, Denmark
Residence	New York City, New York, U.S.
Nationality	Danish
Alma mater	Aarhus University (MSc) University of Cambridge (PhD)
Known for	C++
Awards	Grace Murray Hopper Award (1993) ACM Fellow (1994) IEEE Fellow (1994)

	William Procter Prize for Scientific Achievement (2005) Dr. Dobb's Excellence Award (2008) Dahl–Nygaard Prize (2015) Fellow of the Computer History Museum (2015) IET Faraday Medal (2017) Charles Stark Draper Prize (2018) Computer Pioneer Award (2018)
	Scientific career
Institutions	Aarhus University University of Cambridge Texas A&M University Bell Labs Morgan Stanley Columbia University
Thesis	*Communication and control in distributed computer systems* (1979)
Doctoral advisor	David Wheeler
Website	stroustrup.com

- **Multiple Inheritance for C++**
- **An Overview of the C++ Programming Language**
- **A history of C++**
- **Direction for ISO C++**

- Remember the Vasa!
- A Tour of C++
- The C++ Programming Language
- The UNIX System: Data Abstraction in C
- Multiparadigm Programming in Standard C++
- Speaking C++ as a Native
- What is "Object-Oriented Programming"? (1991 revised version)
- The Design of C++0x
- C++ Dynamic Cast in Autonomous Space Systems
- C and C++: Case Studies in Compatibility
- Exception Safety: Concepts and Techniques
- Evolving a language in and for the real world: C++ 1991-2006
- Lock-free Dynamically Resizable Arrays
- Specifying C++ Concepts
- C and C++: Siblings
- C++ Programming Styles and Libraries
- "New" Value Terminology
- Abstraction and the C++ machine model
- A Brief Look at C++
- C++ Dynamic Cast in Autonomous Space Systems

- Fast dynamic casting
- Model-based Product-Oriented Certification
- Open Multi-Methods for C++
- Learning Standard C++ as a New Language
- Design and evaluation of C++ open multi-methods
- Runtime Concepts for the C++ Standard Template Library
- Why C++ is not just an Object Oriented Programming Language
- Supporting SELL for High-Performance Computing
- A rationale for semantically enhanced library languages
- Sibling Rivalry: C and C++
- Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software
- Serving the C++ Community
- Wrapping C++ Member Function Calls
- Rejuvenating C++ Programs through Demacrofication
- Open and Efficient Type Switch for C++
- Software Development for Infrastructure

Multiple Inheritance for C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Multiple Inheritance is the ability of a class to have more than one base class (super class). In a language where multiple inheritance is supported a program can be structured as a set of inheritance lattices instead of (just) as a set of inheritance trees. This is widely believed to be an important structuring tool. It is also widely believed that multiple inheritance complicates a programming language significantly, is hard to implement, and is expensive to run. I will demonstrate that none of these last three conjectures are true.

1 Introduction

This paper describes an implementation of a multiple inheritance mechanism for C++[4,6.] It provides only the most rudimentary explanation of what multiple inheritance is in general and what it can be used for†. The particular variation of the general concept implemented here is primarily explained in term of this implementation.

First a bit of background on multiple inheritance and C++ implementation technique is presented, then the multiple inheritance scheme implemented for C++ is introduced in three stages:

[1] The basic scheme for multiple inheritance, the basic strategy for ambiguity resolution, and the way to implement virtual functions.

[2] Handling of classes included more than once in an inheritance lattice; the programmer has the choice whether a multiply included base class will result in one or more sub-objects being created.

[3] Details of construction of objects, destruction of objects, and access control.

Finally, some the complexities and overheads introduced by this multiple inheritance scheme are summarized.

2 Multiple Inheritance

Consider writing a simulation of a network of computers. Each node in the network is represented by an object of class `Switch`, each user or computer by an object of class `Terminal`, and each communication line by an object of class `Line`. One way to monitor the simulation (or a real network of the same structure) would be to display the state of objects of various classes on a screen. Each object to be displayed is represented as an object of class `Displayed`. Objects of class `Displayed` are under control of a display manager that ensures regular update of a screen and/or data base. The classes `Terminal` and `Switch` are derived from a class `Task` that provides the basic facilities for co-routine style behavior. Objects of class `Task` are under control of a task manager (scheduler) that manages the real processor(s).

Ideally `Task` and `Displayed` are classes from a standard library. If you want to display a terminal class `Terminal` must be derived from class `Displayed`. Class `Terminal`, however, is already derived from class `Task`. In a single inheritance language, such as the original version of C++[4] or Simula67, we have only two ways of solving this problem: deriving `Task` from `Displayed` or deriving `Displayed`

† An earlier version of this paper was presented to the European UNIX Users' Group conference in Helsinki, May 1987. This paper has been revised to match the multiple inheritance scheme that was arrived at after further experimentation and thought. For more information see references 5 and 6.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

from `Task`. Neither is ideal since they both create a dependency between the library versions of two fundamental and independent concepts. Ideally one would want to be able choose between saying that a `Terminal` is a `Task` *and* a `Displayed`; that a `Line` is a `Displayed` *but not* a `Task`; and that a `Switch` is a `Task` *but not* a `Displayed`.

The ability to express this using a class hierarchy, that is, to derive a class from more than one base class, is usually referred to as *multiple inheritance*. Other examples involve the representation of various kinds of windows in a window system[7] and the representation of various kinds of processors and compilers for a multi-machine, multi-environment debugger[1].

In general, multiple inheritance allows a user to combine independent (and not so independent) concepts represented as classes into a composite concept represented as a derived class. A common way of using multiple inheritance is for a designer to provide sets of base classes with the intention that a user creates new classes by choosing base classes from each of the relevant sets. Thus a programmer creates new concepts using a recipe like ‘‘pick an A and/or a B’’. In the window example, a user might specify a new kind of window by selecting a style of window interaction (from the set of interaction base classes) and a style of appearance (from the set of base classes defining display options). In the debugger example, a programmer would specify a debugger by choosing a processor and a compiler.

Given multiple inheritance and N concepts each of which might somehow be combined with one of M other concepts, we need N+M classes to represent all the combined concepts. Given only single inheritance, we need to replicate information and provide N+M+N*M classes. Single inheritance handles cases where N==1 or M==1. The usefulness of multiple inheritance for avoiding replication hinges on the importance of examples where the values of N and M are both larger than 1. It appears that examples with N>=2 and M>=2 are not uncommon; the window and debugger examples described above will typically have both N and M larger than 2.

3 C++ Implementation Strategy

Before discussing multiple inheritance and its implementation in C++ I will first describe the main points in the traditional implementation of the C++ single inheritance class concept.

An object of a C++ class is represented by a contiguous region of memory. A pointer to an object of a class points to the first byte of that region of memory. The compiler turns a call of a member function into an ‘‘ordinary’’ function call with an ‘‘extra’’ argument; that ‘‘extra’’ argument is a pointer to the object for which the member function is called.

Consider a simple class `A`†:

```
class A {
    int a;
    void f(int i);
};
```

An object of class A will look like this

```
-----------------
|    int a; |
-----------------
```

No information is placed in an `A` except the integer `a` specified by the user. No information relating to (non-virtual) member functions is placed in the object.

A call of the member function `A::f`:

```
A* pa;
pa->f(2);
```

is transformed by the compiler to an ‘‘ordinary function call’’:

† In most of this paper data hiding issues are ignored to simplify the discussion and shorten the examples. This makes some examples illegal. Changing the word `class` to `struct` would make the examples legal, as would adding `public` specifiers in the appropriate places.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
f__F1A(pa,2);
```

Objects of derived classes are composed by concatenating the members of the classes involved:

```
class A { int a; void f(int); };
class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

Again, no ‘‘housekeeping’’ information is added, so an object of class `C` looks like this:

```
-----------------
|    int a; |
|    int b; |
|    int c; |
-----------------
```

The compiler ‘‘knows’’ the position of all members in an object of a derived class exactly as it does for an object of a simple class and generates the same (optimal) code in both cases.

Implementing virtual functions involves a table of functions. Consider:

```
class A {
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};

class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

In this case, a table of virtual functions, the `vtbl`, contains the appropriate functions for a given class and a pointer to it is placed in every object. A class `C` object looks like this:

```
-----------------
|    int a; |   vtbl:
|       vptr .........> -----------------
|    int b; |   |     A::f  |
|    int c; |   |     B::g  |
-----------------   |     C::h  |
             -----------------
```

A call to a virtual function is transformed into an indirect call by the compiler. For example,

```
C* pc;
pc->g(2);
```

becomes something like:

```
(*(pc->vptr[1]))(pc,2);
```

A multiple inheritance mechanism for C++ must preserve the efficiency and the key features of this implementation scheme.

4 Multiple Base Classes

Given two classes

```
class A { ... };
class B { ... };
```

one can design a third using both as base classes:

```
class C : A , B { ... };
```

This means that a `C` is an `A` and a `B`. One might equivalently† define `C` like this:

† Except for possible side effects in constructors and destructors (access to global variables, input operations, output operations, etc.).

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
class C : B , A { ... };
```

4.1 Object Layout

An object of class `C` can be laid out as a contiguous object like this:

```
-----------------
|       |
|     A part    |
|       |
-----------------
|       |
|     B part    |
|       |
-----------------
|       |
|     C part    |
|       |
-----------------
```

Accessing a member of classes `A`, `B` or `C` is handled exactly as before: the compiler knows the location in the object of each member and generates the appropriate code (without spurious indirections or other overhead).

4.2 Member Function Call

Calling a member function of `A` or `C` is identical to what was done in the single inheritance case. Calling a member function of `B` given a `C*` is slightly more involved:

```
C* pc;
pc->bf(2);  // assume that bf is a member of B
        // and that C has no member named bf
        // except the one inherited from B
```

Naturally, `B::bf()` expects a `B*` (to become its `this` pointer). To provide it, a constant must be added to pc. This constant, `delta(B)`, is the relative position of the `B` part of `C`. This delta is known to the compiler that transforms the call into:

```
bf__F1B((B*)((char*)pc+delta(B)),2);
```

The overhead is one addition of a constant per call of this kind. During the execution of a member function of `B` the function's `this` pointer points to the `B` part of `C`:

```
pc ..................> -----------------
            |       |
            |     A part    |
            |       |
B::bf's this .........> -----------------
            |       |
            |     B part    |
            |       |
            -----------------
            |       |
            |     C part    |
            |       |
            -----------------
```

Note that there is no space penalty involved in using a second base class and that the minimal time penalty is incurred only once per call.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

4.3 Ambiguities

Consider potential ambiguities if both `A` and `B` have a public member `ii`:

```
class A { int ii; };
class B { char* ii; };
class C : A, B { };
```

In this case `C` will have two members called `ii`, `A::ii` and `B::ii`. Then

```
C* pc;
pc->ii;     // error: A::ii or B::ii ?
```

is illegal since it is ambiguous. Such ambiguities can be resolved by explicit qualification:

```
pc->A::ii;  // C's A's ii
pc->B::ii;  // C's B's ii
```

A similar ambiguity arises if both `A` and `B` have a function `f()`:

```
class A { void f(); };
class B { int f(); };
class C : A, B { };

C* pc;
pc->f();    // error: A::f or B::f ?

pc->A::f(); // C's A's f
pc->B::f(); // C's B's f
```

As an alternative to specifying which base class in each call of an `f()`, one might define an `f()` for `C`. `C::f()` might call the base class functions. For example:

```
class C : A, B {
    int f() { A::f(); return B::f(); }
};

C* pc;
pc->f();    // C::f is called
```

This solution usually leads to cleaner programs; it localizes the specification of the meaning of the name for objects of a derived class to the declaration of the derived class.

4.4 Casting

Explicit and implicit casting may also involve modifying a pointer value with a delta:

```
C* pc;
B* pb;
pb = (B*)pc;    // pb = (B*)((char*)pc+delta(B))
pb = pc;    // pb = (B*)((char*)pc+delta(B))
pc = pb;    // error: cast needed
pc = (C*)pb;    // pc = (C*)((char*)pb-delta(B))
```

Casting yields the pointer referring to the appropriate part of the same object.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
pc ...> -----------------
    |       |
    |     A part    |
    |       |
pb ...> -----------------
    |       |
    |     B part    |
    |       |
    -----------------
    |       |
    |     C part    |
    |       |
    -----------------
```

Comparisons are interpreted in the same way:

```
pc == pb;   // that is, pc == (C*)pb
        //          or equivalently (B*)pc == pb

        // that is, (B*)((char*)pc+delta(B)) == pb
        //          or equivalently pc == (C*)((char*)pb-delta(B))
```

Note that in both C and C++ casting has always been an operator that produced one value given another rather than an operator that simply reinterpreted a bit pattern. For example, on almost all machines `(int).2` causes code to be executed; `(float)(int).2` is not equal to `.2`. Introducing multiple inheritance as described here will introduce cases where `(char*)(B*)v!=(char*)v` for some pointer type `B*`. Note, however, that when `B` is a base class of `C`, `(B*)v==(C*)v==v`.

4.5 Zero Valued Pointers

Pointers with the value zero cause a separate problem in the context of multiple base classes. Consider applying the rules presented above to a zero-valued pointer:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;    // pb = (B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The second test would fail since `pb` would have the value `(B*)((char*)0+delta(B))`.

The solution is to elaborate the conversion (casting) operation to test for the pointer-value `0`:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;        // pb = (pc==0)?0:(B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The added complexity and run-time overhead are a test and an increment.

5 Virtual Functions

Naturally, member functions may be virtual:

```
class A { virtual void f(); };
class B { virtual void f(); virtual void g(); };
class C : A , B { void f(); };

A* pa = new C;
B* pb = new C;
C* pc = new C;
```

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
pa->f();
pb->f();
pc->f();
```

All these calls will invoke `C::f()`. This follows directly from the definition of `virtual` since class `C` is derived from class `A` and from class `B`.

5.1 Implementation

On entry to `C::f`, the `this` pointer must point to the beginning of the `C` object (and not to the `B` part). However, it is not in general known at compile time that the `B` pointed to by `pb` is part of a `C` so the compiler cannot subtract the constant `delta(B)`. Consequently `delta(B)` must be stored so that it can be found at run time. Since it is only used when calling a virtual function the obvious place to store it is in the table of virtual functions (`vtbl`). For reasons that will be explained below the delta is stored with each function in the `vtbl` so that a `vtbl` entry will be of the form†:

```
struct vtbl_entry {
    void    (*fct)();
    int delta;
};
```

An object of class `C` will look like this:

```
-----------------
|       |   vtbl:
|   vptr ..........>---------------------
|   A part  |   |  C::f |     0     |
|       |   ---------------------
-----------------
|       |   vtbl:
|   vptr ..........>---------------------
|   B part  |   |  C::f | -delta(B) |
|       |   |  B::g |     0     |
-----------------   ---------------------
|       |
|   C part  |
|       |
-----------------

pb->f();    // call of C::f:
        // register vtbl_entry* vt = &pb->vtbl[index(f)];
        // (*vt->fct)((B*)((char*)pb+vt->delta))
```

Note that the object pointer may have to be adjusted to point to the correct sub-object before looking for the member pointing to the `vtbl`. Note also that each combination of base class and derived class has its own `vtbl`. For example, the `vtbl` for `B` in `C` is different from the `vtbl` of a separately allocated `B`. This implies that in general an object of a derived class needs a `vtbl` for each base class plus one for the derived class. However, as with single inheritance, a derived class can share a `vtbl` with its first base so that in the example above only two `vtbl`s are used for an object of type `C` (one for `A` in `C` combined with `C`'s own plus one for `B` in `C`).

Using an `int` as the type of a stored delta limits the size of a single object; that might not be a bad thing.

† The AT&T C++ 2.0 implementation uses three fields in this structure, but only the two shown here are used by the virtual function call mechanism; see reference 3.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

5.2 Ambiguities

The following demonstrates a problem:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B { void f(); };

C* pc = new C;

pc->f();

pc->A::f();
pc->B::f();
```

Explicit qualification ‘‘suppresses’’ `virtual` so the last two calls really invoke the base class functions. Is this a problem? Usually, no. Either `C` has an `f()` and there is no need to use explicit qualification or `C` has no `f()` and the explicit qualification is necessary and correct. Trouble can occur when a function `f()` is added to `C` in a program that already contains explicitly qualified names. In the latter case one could wonder why someone would want to both declare a function virtual and also call it using explicit qualification. If `f()` is virtual, adding an `f()` to the derived class is clearly the correct way of resolving the ambiguity.

The case where no `C::f` is declared cannot be handled by resolving ambiguities at the point of call. Consider:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B {  };        // error: C::f needed

C* pc = new C;
pc->f();    // ambiguous

A* pa = pc; // implicit conversion of C* to A*
pa->f();    // not ambiguous: calls A::f();
```

The potential ambiguity in a call of `f()` is detected at the point where the virtual function tables for `A` and `B` in `C` are constructed. In other words, the declaration of C above is illegal because it would allow calls, such as `pa->f()`, which are unambiguous *only* because type information has been ‘‘lost’’ through an implicit coercion; a call of `f()` for an object of type `C` is ambiguous.

6 Multiple Inclusions

A class can have any number of base classes. For example,

```
class A : B1, B2, B3, B4, B5, B6 { ... };
```

It illegal to specify the same class twice in a list of base classes. For example,

```
class A : B, B { ... }; // error
```

The reason for this restriction is that every access to a B member would be ambiguous and therefore illegal; this restriction also simplifies the compiler.

6.1 Multiple Sub-objects

A class may be included more than once as a base class. For example:

```
class L { ... };
class A : L { ... };
class B : L { ... };
class C : A , B { ... };
```

In such cases multiple objects of the base class are part of an object of the derived class. For example, an object of class `C` has two `L`’s: one for `A` and one for `B`:

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
-----------------
|       |
| L part (of A) |
|       |
|  -----------  |
|       |
|     A part    |
|       |
-----------------
|       |
| L part (of B) |
|       |
|  -----------  |
|       |
|     B part    |
|       |
-----------------
|       |
|     C part    |
|       |
-----------------
```

This can even be useful. Think of `L` as a link class for a Simula-style linked list. In this case a `C` can be on both the list of `A`s and the list of `B`s.

6.2 Naming

Assume that class `L` in the example above has a member `m`. How could a function `C::f` refer to `L::m`? The obvious answer is ‘‘by explicit qualification’’:

```
void C::f() { A::m = B::m; }
```

This will work nicely provided neither `A` nor `B` has a member `m` (except the one they inherited from `L`). If necessary, the qualification syntax of C++ could be extended to allow the more explicit:

```
void C::f() { A::L::m = B::L::m; }
```

6.3 Casting

Consider the example above again. The fact that there are two copies of `L` makes casting (both explicit and implicit) between `L*` and `C*` ambiguous, and consequently illegal:

```
C* pc = new C;
L* pl = pc;       // error: ambiguous
pl = (L*)pc;      // error: still ambiguous
pl = (L*)(A*)pc;  // The L in C's A
pc = pl;          // error: ambiguous
pc = (L*)pl;      // error: still ambiguous
pc = (C*)(A*)pl;  // The C containing A's L
```

I don’t expect this to be a problem. The place where this will surface is in cases where `A`s (or `B`s) are handled by functions expecting an `L`; in these cases a `C` will not be acceptable despite a `C` being an `A`:

```
extern f(L*);   // some standard function

A aa;
C cc;

f(&aa);     // fine
f(&cc);     // error: ambiguous
f((A*)&cc); // fine
```

Casting is used for explicit disambiguation.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

7 Virtual Base Classes

When a class `C` has two base classes `A` and `B` these two base classes give rise to separate sub-objects that do not relate to each other in ways different from any other `A` and `B` objects. I call this *independent multiple inheritance*. However, many proposed uses of multiple inheritance assume a dependence among base classes (for example, the style of providing a selection of features for a window described in §2). Such dependencies can be expressed in term of an object shared between the various derived classes. In other words, there must be a way of specifying that a base class must give rise to only one object in the final derived class even if it is mentioned as a base class several times. To distinguish this usage from independent multiple inheritance such base classes are specified to be virtual:

```
class AW : virtual W { ... };
class BW : virtual W { ... };
class CW : AW , BW { ... };
```

A single object of class `W` is to be shared between `AW` and `BW`; that is, only one `W` object must be included in `CW` as the result of deriving `CW` from `AW` and `BW`. Except for giving rise to a unique object in a derived class, a `virtual` base class behaves exactly like a non-virtual base class.

The ‘‘virtualness’’ of `W` is a property of the derivation specified by `AW` and `BW` and not a property of `W` itself. Every `virtual` base in an inheritance DAG refers to the same object.

A class may be both a normal and a virtual base in an inheritance DAG:

```
class A : virtual L { ... };
class B : virtual L { ... };
class C : A , B { ... };
class D : L, C { ... };
```

A `D` object will have two sub-objects of class `L`, one virtual and one ‘‘normal.’’

Virtual base classes provide a way of sharing information within an inheritance DAG without pushing the shared information to an ultimate base class. Virtual bases can therefore be used to improve locality of reference. Another way of viewing classes derived from a common virtual base is as alternative and composable interfaces to the virtual base.

7.1 Representation

The object representing a virtual base class `W` object cannot be placed in a fixed position relative to both `AW` and `BW` in all objects. Consequently, a pointer to `W` must be stored in all objects directly accessing the `W` object to allow access independently of its relative position. For example:

```
AW* paw = new AW;
BW* pbw = new BW;
CW* pcw = new CW;

    -----------------
paw ..> |         ...........
    |     AW part   |   .
    |       |   v
    -----------------   .
        |               |<.......
    |     W part    |
    |       |
    -----------------
```

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
    -----------------
pbw ..> |         ...........
    |     BW part   |   .
    |       |   v
    -----------------   .
        |               |<.......
    |     W part    |
    |       |
    -----------------

    -----------------
pcw ..> |         ...........
    |     AW part   |   .
    |       |   v
    -----------------   .
    |         ...........
    |     BW part   |   .
    |       |   v
    -----------------   .
    |       |   .
    |     CW part   |   .
    |       |   v
    -----------------   .
        |               |<.......
    |     W part    |
    |       |
    -----------------
```

A class can have an arbitrary number of virtual base classes.

One can cast from a derived class to a virtual base class, but not from a virtual base class to a derived class. The former involves following the virtual base pointer; the latter cannot be done given the information available at run time. Storing a ‘‘back-pointer’’ to the enclosing object(s) is non-trivial in general and was considered unsuitable for C++ as was the alternative strategy of dynamically keeping track of the objects ‘‘for which’’ a given member function invocation operates. The uses of such a back-pointer mechanism did not seem to warrant the added implementation complexity, the extra space required in objects, and the added run-time cost of initialization.

The information necessary to cast from a virtual base class to a derived class is part of what is needed to perform a virtual call of a function declared in the virtual base and overridden by the derived class. The absence of the ‘back-pointer can thus be compensated for by defining and calling appropriate virtual functions:

```
class B { virtual void f(); ... };
class D : virtual B { void f(); ... };

void g(B* p)
{
    // instead of casting B* to a D* here call f()
}
```

Since explicit casting is often bets avoided this technique actually leads to better code in many cases.

7.2 Virtual Functions

Consider:

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
class W {
    virtual void f();
    virtual void g();
    virtual void h();
    virtual void k();
    ...
};

class AW : virtual W { void g(); ... };
class BW : virtual W { void f(); ... };
class CW : AW , BW { void h(); ... };

CW* pcw = new CW;

pcw->f();       // BW::f()
pcw->g();       // AW::g()
pcw->h();       // CW::h()
((AW*)pcw)->f();    // BW::f();
```

A `CW` object might look like this:

```
-----------------
.........       |
.   |   AW part |
v   |       |
.   -----------------
.........       |
.   |   BW part |
v   |       |
.   -----------------
.   |       |
.   |   CW part |
v   |       |   vtbl:
.   -----------------   ------------------------------
...>|   vptr ..........>| BW::f | delta(BW)-delta(W) |
|       |   | AW::g |     -delta(W)      |
|   W part  |   | CW::h |     -delta(W)      |
|       |   |  W::k |         0      |
-----------------   ------------------------------
```

In general, the delta stored with a function pointer in a `vtbl` is the delta of the class defining the function minus the delta of the class for which the `vtbl` is constructed.

If `W` has a virtual function `f` that is re-defined in both `AW` and `BW` but not in `CW` an ambiguity results. Such ambiguities are easily detected at the point where `CW`'s `vtbl` is constructed.

The rule for detecting ambiguities in a class lattice, or more precisely a directed acyclic graph (DAG) of classes, is that there all re-definitions of a virtual function from a virtual base class must occur on a single path through the DAG. The example above can be drawn as a DAG like this:

```
 ....> W { f g h k } <...
 |          |
 ^          ^

 |          |
AW { g }           BW { f }
 |          |
 ^          ^

 |          |
 ...<... CW { h } ...>...
```

Note that a call ‘‘up’’ through one path of the DAG to a virtual function may result in the call of a function (re-defined) in another path (as happened in the call `((AW*)pcw)->f()` in the example above).

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

7.3 Using virtual bases

Programming with virtual bases is trickier than programming with non-virtual bases. The problem is to avoid multiple calls of a function in a virtual class when that is not desired. Here is a possible style:

```
class W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); }
    // ...
};
```

Each class provides a protected function doing ‘‘its own stuff’’, `_f()`, for use by derived classes and a public function `f()` as the interface for use by the ‘‘general public.’’

```
class A : public virtual W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); W::_f(); }
    // ...
};
```

A derived class `f()` does its ‘‘own stuff’’ by calling `_f()` and its base classes’ ‘‘own stuff’’ by calling their `_f()`s.

```
class B : public virtual W  {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); W::_f(); }
    // ...
};
```

In particular, this style enables a class that is (indirectly) derived twice from a class `W` to call `W::f()` once only:

```
class C : public A, public B, public virtual W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); A::_f(); B::_f(); W::_f(); }
    // ...
};
```

Method combination schemes, such as the ones found in Lisp systems with multiple inheritance, were considered as a way of reducing the amount of code a programmer needed to write in cases like the one above. However, none of these schemes appeared to be sufficiently simple, general, and efficient enough to warrant the complexity it would add to C++.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

8 Constructors and Destructors

Constructors for base classes are called before the constructor for their derived class. Destructors for base classes are called after the destructor for their derived class. Destructors are called in the reverse order of their declaration.

Arguments to base class constructors can be specified like this:

```
class A { A(int); };
class B { B(int); };
class C : A , virtual B {
    C(int a, int b) : A(a) , B(b) { ... }
};
```

Constructors are executed in the order they appear in the list of bases except that a virtual base is always constructed before classes derived from it.

A virtual base is always constructed (once only) by its ‘most derived’ class. For example:

```
class V { V(); V(int); ... };
class A : virtual V { A(); A(int); ... };
class B : virtual V { B(); B(int); ... };
class C : A, B { C(); C(int); ... };

    V v(1); // use V(int)
    A a(2); // use V(int)
    B b(3); // use V()
    C c(4); // use V()
```

9 Access Control

The examples above ignore access control considerations. A base class may be `public` or `private`. In addition, it may be `virtual`. For example:

```
class D
    : B1        // private (by default), non-virtual (by default)
    , virtual B2    // private (by default), virtual
    , public B3 // public, non-virtual (by default)
    , public virtual B4 {
    // ...
};
```

Note that a access or `virtual` specifier applies to a single base class only. For example,

```
class C : public A, B { ... };
```

declares a public base `A` and a private base `B`.

A virtual class that is accessible through a path is accessible even if other paths to it use private derivation.

10 Overheads

The overhead in using this scheme is:

[1] One subtraction of a constant for each use of a member in a base class that is included as the second or subsequent base.

[2] One word per function in each `vtbl` (to hold the delta).

[3] One memory reference and one subtraction for each call of a virtual function.

[4] One memory reference and one subtraction for access of a base class member of a virtual base class.

Note that overheads [1] and [4] are only incurred where multiple inheritance is actually used, but overheads [2] and [3] are incurred for each class with virtual functions and for each virtual function call even when multiple inheritance is not used. Overheads [1] and [4] are only incurred when members of a second or subsequent base are accessed ‘‘from the outside’’; a member function of a virtual base class does not incur special overheads when accessing members of its class.

This implies that except for [2] and [3] you pay only for what you actually use; [2] and [3] impose a

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

minor overhead on the virtual function mechanism even where only single inheritance is used. This latter overhead could be avoided by using an alternative implementation of multiple inheritance, but I don't know of such an implementation that is also faster in the multiple inheritance case and as portable as the scheme described here.

Fortunately, these overheads are not significant. The time, space, and complexity overheads imposed on the compiler to implement multiple inheritance are not noticeable to the user.

11 But is it Simple to Use?

What makes a language facility hard to use?

[1] Lots of rules.

[2] Subtle differences between rules.

[3] Inability to automatically detect common errors.

[4] Lack of generality.

[5] Deficiencies.

The first two cases lead to difficulty of learning and remembering, causing bugs due to misuse and misunderstanding. The last two cases cause bugs and confusion as the programmer tries to circumvent the rules and ''simulate'' missing features. Case [3] causes frustration as the programmer discovers mistakes the hard way.

The multiple inheritance scheme presented here provides two ways of extending a class's name space:

[1] A base class.

[2] A virtual base class.

These are two ways of creating/specifying a new class rather than ways of creating two different kinds of classes. The rules for using the resulting classes do not depend on how the name space was extended:

[1] Ambiguities are illegal.

[2] Rules for use of members are what they were for single inheritance.

[3] Visibility rules are what they were for single inheritance.

[3] Initialization rules are what they were for single inheritance.

Violations of these rules are detected by the compiler.

In other words, the multiple inheritance scheme is only more complicated to use than the existing single inheritance scheme in that

[1] You can extend a class's name space more than once (with more than one base class).

[2] You can extend a class's name space in two ways rather than in only one way.

In addition, care must be taken to take the sharing into account when programming member functions of classes with virtual base classes; see section 7 above.

This appears minimal and constitutes an attempt to provide a formal and (comparatively) safe set of mechanisms for observed practices and needs. I think that the scheme described here is ''as simple as possible, but no simpler.''

A potential source of problems exists in the absence of ''system provided back-pointers'' from a virtual base class to its enclosing object.

In some contexts, it might also be a problem that pointers to sub-objects are used extensively. This will affect programs that use explicit casting to non-object-pointer types (such as `char*`) and ''extra linguistic'' tools (such as debuggers and garbage collectors). Otherwise, and hopefully normally, all manipulation of object pointers follows the consistent rules explained in §4, §7, and §8 and is invisible to the user.

12 Conclusions

Multiple inheritance is reasonably simple to add to C++ in a way that makes it easy to use. Multiple inheritance is not too hard to implement, since it requires only very minor syntactic extensions, and fits naturally into the (static) type structure. The implementation is very efficient in both time and space. Compatibility with C is not affected. Portability is not affected.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

13 Acknowledgements

In 1984 I had a long discussion with Stein Krogdahl from the University of Oslo, Norway. He had devised a scheme for implementing multiple inheritance in Simula using pointer manipulation based on addition and subtaction of constants. Reference 2 describes this work. Tom Cargill, Jim Coplien, Brian Kernighan, Andy Koenig, Larry Mayka, Doug McIlroy, and Jonathan Shopiro supplied many valuable suggestions and questions.

14 References

[1] Tom Cargill:
PI: A Case Study in Object-Oriented Programming.
OOPSLA'86 Proceedings, pp 350-360, September 1986.
[2] Stein Krogdahl:
An Efficient Implementation of Simula Classes with Multiple Prefixing.
Research Report No. 83 June 1984,
University of Oslo, Institute of Informatics.
[3] Stan Lippman and Bjarne Stroustrup:
Pointers to Members in C++
Proc. USENIX C++ Conference, Denver, October 1988.
[4] Bjarne Stroustrup:
The C++ Programming Language.
Addison-Wesley, 1986.
[5] Bjarne Stroustrup:
What is ''Objet-Oriented Programming?''.
Proc. ECOOP,
Springer Verlag Lecture Notes in Computer Science, Vol 276, June 1987.
[6] Bjarne Stroustrup:
The Evolution of C++: 1985-1989.
USENIX Computing Systems Vol 2 no 3, Fall 1989.
[7] Daniel Weinreb and David Moon:
Lisp Machine Manual.
Symbolics, Inc. 1981.

14.1 Appendix

The original multiple inheritance design as presented to the EUUG conference in Helsinki in May 1987 contained a notion of delegation†. Briefly, a user was allowed to specify a pointer to some class among the base classes in a class declaration and the object thus designated would be used exactly as if it was an object representing a base class. For example:

```
class B { int b; void f(); };
class C : *p { B* p; int c; };
```

The `:*p` meant that members of the object pointed to by `p` could be accessed exactly as if they were members of class `C`:

```
void f(C* q)
{
    q->f();     // that is, q->p->f()
}
```

An object of class `C` looked something like this after `C::p` has been initialized:

† Gul Agha: *An Overview of Actor languages*. SIGPLAN Notices, pp58-67, October 1986.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
-----------------
|    B* p; ............>-----------------
|    int c; |   |    int b; |
-----------------   -----------------
```

This concept looked very promising for representing structures that require more flexibility than is provided by ordinary inheritance. In particular, assignment to a delegation pointer could be used to reconfigure an object at run-time. The implementation was trivial and the run-time and space efficiency ideal. Unfortunately, every user of this mechanism suffered serious bugs and confusion. Because of this the delegation mechanism was not included in C++.

I suspect the fundamental problem with this variant of the notion of delegation was that functions of the delegating class could not override functions of the class derived to. Allowing such overriding would be at odds with the C++ strategy of static type checking and ambiguity resolution (consider the case of an object of a class with virtual functions delegated to by several objects of different classes).

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

From *The Handbook of Object Technology (Editor: Saba Zamir). CRC Press LLC, Boca Raton. 1999. ISBN 0-8493-3135-8.*

An Overview of the C++ Programming Language

Bjarne Stroustrup

AT&T Laboratories
Florham Park, NJ07932-0971, USA

ABSTRACT

This overview of C++ presents the key design, programming, and language-technical concepts using examples to give the reader a feel for the language. C++ is a general-purpose programming language with a bias towards systems programming that supports efficient low-level computation, data abstraction, object-oriented programming, and generic programming.

1 Introduction and Overview

The C++ programming language provides a model of memory and computation that closely matches that of most computers. In addition, it provides powerful and flexible mechanisms for abstraction; that is, language constructs that allow the programmer to introduce and use new types of objects that match the concepts of an application. Thus, C++ supports styles of programming that rely on fairly direct manipulation of hardware resources to deliver a high degree of efficiency plus higher-level styles of programming that rely on user-defined types to provide a model of data and computation that is closer to a human's view of the task being performed by a computer. These higher-level styles of programming are often called data abstraction, object-oriented programming, and generic programming.

This paper is organized around the main programming styles directly supported by C++:

§2 *The Design and Evolution of C++* describes the aims of C++ and the principles that guided its evolution.

§3 *The C Programming Model* presents the C subset of C++ and other C++ facilities supporting traditional systems-programming styles.

§4 *The C++ Abstraction Mechanisms* introduces C++'s class concept and its use for defining new types that can be used exactly as built-in types, shows how abstract classes can be used to provide interfaces to objects of a variety of types, describes the use of class hierarchies in object-oriented programming, and presents templates in support of generic programming.

§5 *Large-Scale Programming* describes namespaces and exception handling provided to ease the composition of programs out of separate parts.

§6 *The C++ Standard Library* presents standard facilities such as I/O streams, strings, containers (e.g. ***vector***, ***list***, and ***map***), generic algorithms (e.g. ***sort***( ), ***find***( ), ***for_each***( )) and support for numeric computation.

To round off, a brief overview of some of the tasks that C++ has been used for and some suggestions for further reading are given.

2 The Design and Evolution of C++

C++ was designed and implemented by Bjarne Stroustrup (the author of this article) at AT&T Bell Laboratories to combine the organizational and design strengths of Simula with C's facilities for systems programming. The initial version of C++, called ''C with Classes'' [Stroustrup,1980], was first used in 1980; it supported traditional system programming techniques (§3) and data abstraction (§4.1). The basic facilities for object-oriented programming (§4.2-4.3) were added in 1983 and object-oriented design and programming techniques were gradually introduced into the C++ community. The language was first made commercially available in 1985 [Stroustrup,1986] [Stroustrup,1986b]. Facilities for generic programming (§4.4) were added to the language in the 1987-1989 time frame [Ellis,1990] [Stroustrup,1991].

As the result of widespread use and the appearance of several independently-developed C++

implementations, formal standardization of C++ started in 1990 under the auspices of the American National Standards Institute, ANSI, and later the International Standards Organization, ISO, leading to an international standard in 1998 [C++,1998]. During the period of standardization the standards committee acted as an important focus for the C++ community and its draft standards acted as interim definitions of the language. As an active member of the standards committee, I was a key participant in the further evolution of C++. Standard C++ is a better approximation to my ideals for C++ than were earlier versions. The design and evolution of C++ is documented in [Stroustrup,1994] [Stroustrup,1996] and [Stroustrup,1997b]. The language as it is defined at the end of the standardization process and the key design and programming techniques it directly supports are presented in [Stroustrup,1997].

2.1 C++ Design Aims

C++ was designed to deliver the flexibility and efficiency of C for systems programming together with Simula's facilities for program organization (usually referred to as object-oriented programming). Great care was taken that the higher-level programming techniques from Simula could be applied to the systems programming domain. That is, the abstraction mechanisms provided by C++ were specifically designed to be applicable to programming tasks that demanded the highest degree of efficiency and flexibility.

These aims can be summarized:

Aims:
C++ makes programming more enjoyable for serious programmers.
C++ is a general-purpose programming language that
– is a better C
– supports data abstraction
– supports object-oriented programming
– supports generic programming

Support for generic programming emerged late as an explicit goal. During most of the evolution of C++, I presented generic programming styles and the language features that support them (§4.4) under the heading of ''data abstraction.''

2.2 Design Principles

In [Stroustrup,1994], the design rules for C++ are listed under the headings *General rules*, *Design-support rules*, *Language-technical rules*, and *Low-level programming support rules*:

General rules:
C++'s evolution must be driven by real problems.
C++ is a language, not a complete system.
Don't get involved in a sterile quest for perfection.
C++ must be useful *now*.
Every feature must have a reasonably obvious implementation.
Always provide a transition path.
Provide comprehensive support for each supported style.
Don't try to force people.

Note the emphasis on immediate utility in real-world applications and the respect for the skills and preferences of programmers implied by the last three points. From the start, C++ was aimed at programmers engaged in demanding real-world projects. Perfection was considered unattainable because needs, backgrounds, and problems vary too much among C++ users. Also, notions of perfection change significantly over the lifespan of a general-purpose programming language. Thus, feedback from user and implementer experience is essential in the evolution of a language.

Design-support rules:
Support sound design notions.
Provide facilities for program organization.
Say what you mean.
All features must be affordable.
It is more important to allow a useful feature than to prevent every misuse.
Support composition of software from separately developed parts.

The aim of C++ was to improve the quality of programs produced by making better design and programming techniques simpler to use and affordable. Most of these techniques have their root in Simula [Dahl,1970] [Dahl,1972] [Birtwistle,1979] and are usually discussed under the labels of object-oriented programming and object-oriented design. However, the aim was always to support a range of design and programming styles. This contrasts to a view of language design that tries to channel all system building into a single heavily supported and enforced style (paradigm).

Language-technical rules:
No implicit violations of the static type system.
Provide as good support for user-defined types as for built-in types.
Locality is good.
Avoid order dependencies.
If in doubt, pick the variant of a feature that is easiest to teach.
Syntax matters (often in perverse ways).
Preprocessor usage should be eliminated.

These rules must be considered in the context created of the more general aims. In particular, the desire for a high degree of C compatibility, uncompromising efficiency, and immediate real-world utility counteracts desires for complete type safety, complete generality, and abstract beauty.

From Simula, C++ borrowed the notion of user-defined types (classes, §4.1) and hierarchies of classes (§4.3). However, in Simula and many similar languages there are fundamental differences in the support provided for user-defined types and for built-in types. For example, Simula does not allow objects of user-defined types to be allocated on the stack and addressed directly. Instead, all class objects must be allocated in dynamic memory and accessed through pointers (called *references* in Simula). Conversely, built-in types can be genuinely local (stack-frame allocated), cannot be allocated in dynamic memory, and cannot be referred to by pointers. This difference in treatment of built-in types and user-defined types had serious efficiency implications. For example, when represented as a reference to an object allocated in dynamic memory, a user-defined type – such as ***complex*** (§4.1) – incurs overheads in run-time and space that were deemed unacceptable for the kind of applications for which C++ was intended. Also, the difference in style of usage would preclude uniform treatment of semantically similar types in generic programming (§4.4).

When maintaining a large program, a programmer must invariably make changes based of incomplete knowledge and looking at only a small part of the code. Consequently, C++ provides classes (§4), namespaces (§5.2), and access control (§4.1) to help localize design decisions.

Some order dependencies are unavoidable in a language designed for one-pass compilation. For example, in C++ a variable or a function cannot be used before it has been declared. However, the rules for class member names and the rules for overload resolution were made independent of declaration order to minimize confusion and error.

Low-level programming support rules:
Use traditional (dumb) linkers.
No gratuitous incompatibilities with C.
Leave no room for a lower-level language below C++ (except assembler).
What you don't use, you don't pay for (zero-overhead rule).
If in doubt, provide means for manual control.

C++ was designed to be source-and-link compatible with C wherever this did not seriously interfere with C++'s support for strong type checking. Except for minor details, C++ has C [Kernighan,1978] [Kernighan,1988] as a subset. Being C-compatible ensured that C++ programmers immediately had a complete language and toolset available. It was also important that high-quality educational materials were available for C, and that C compatibility gave the C++ programmer direct and efficient access to a multitude of libraries. At the time when the decision to base C++ on C was made, C wasn't as prominent as it later became and language popularity was a minor concern compared to the flexibility and basic efficiency offered by C.

However, C compatibility also leaves C++ with some syntactic and semantic quirks. For example, the C declarator syntax is far from elegant and the rules for implicit conversions among built-in types are chaotic. It is also a problem that many programmers migrate from C to C++ without appreciating that radical improvements in code quality are only achieved by similarly radical changes to programming styles.

3 The C Programming Model

A fundamental property of computers in widespread use has remained remarkably constant: Memory is a sequence of words or bytes, indexed by integers called addresses. Modern machines – say, designed during the last 20 years – have in addition tended to support directly the notion of a function call stack. Furthermore, all popular machines have some important facilities – such as input-output – that do not fit well into the conventional byte- or word-oriented model of memory or computation. These facilities may require special machine instructions or access to ''memory'' locations with peculiar semantics. Either way, from a higher-level language point of view, the use of these facilities is messy and machine-architecture-specific.

C is by far the most successful language providing the programmer with a programming model that closely matches the machine model. C provides language-level and machine-architecture-independent notions that directly map to the key hardware notions: characters for using bytes, integers for using words, pointers for using the addressing mechanisms, functions for program abstraction, and an absence of constraining language features so that the programmer can manipulate the inevitable messy hardware-specific details. The net effect has been that C is relatively easy to learn and use in areas where some knowledge of the real machine is a benefit. Moreover, C is easy enough to implement that it has become almost universally available.

3.1 Arrays and Pointers

A C array is simply a sequence of memory locations. For example:

```
int v[10];      // an array of 10 ints
v[3] = 1;       // assign 1 to v[3]
int x = v[3];   // read from v[3]
```

The subscript notation `[]` is used both in declarations to indicate an array and in expressions referring to elements of an array.

A C pointer is a variable that can hold an address of a memory location. For example:

```
int* p;         // p is a pointer to an int
p = &v[7];      // assign the address of v[7] to p
*p = 4;         // write to v[7] through p
int y = *p;     // read from v[7] through p
```

The pointer dereference (''points to'') notation `*` is used both in declarations to indicate a pointer and in expressions referring to the element pointed to.

This can be represented graphically:

C++ adopted this inherently simple and close-to-the-machine notion of memory from C. It also adopted

C's notion of expressions, control structures, and functions. For example, we can write a function that finds an element in a vector and returns a pointer to the matching element like this:

```
int* find(int v[], int vsize, int val)        // find val in v
{
    for (int i = 0; i<vsize; i++)             // loop through 0..vsize-1
        if (v[i]==val) return &v[i];          // if val is found return pointer to element
    return &v[vsize];                         // if not found return pointer to one-beyond-the-end of v
}
```

The ++ operator means increment. Thus, the name C++ can be read as ''one more than C,'' ''next C,'' or ''successor to C.'' It is pronounced ''See Plus Plus.''

The *find*( ) function might be used like this:

```
int count[] = { 2, 3, 1, 9, 7, 3, 3, 0, 2 };
int count_size = 9;

void f( )
{
    int* p = find(count, count_size, 7);      // find 7 in count
    int* q = find(count, count_size, 0);      // find 0 in count
    *q = 4;
    // ...
}
```

The C++ standard library provides more general versions of functions such as *find*( ); see §6.3. A function declared *void*, as *f*( ) above doesn't return a value.

3.2 Storage

In C and C++, there are three fundamental ways of using memory:

Static memory, in which an object is allocated by the linker for the duration of the program. Global variables, ***static*** class members, and ***static*** variables in functions are allocated in static memory. An object allocated in static memory is constructed once and persists to the end of the program. Its address does not change while the program is running. Static objects can be a problem in programs using threads (shared-address-space concurrency) because they are shared and require locking for proper access.

Automatic memory, in which function arguments and local variables are allocated. Each entry into a function or a block gets its own copy. This kind of memory is automatically created and destroyed; hence the name automatic memory. Automatic memory is also said ''to be on the stack.''

Free store, from which memory for objects is explicitly requested by the program and where a program can free memory again once it is done with it (using the ***new*** and ***delete*** operators). When a program needs more free store, ***new*** requests it from the operating system. Typically, the free store (also called *dynamic memory* or *the heap*) grows throughout the lifetime of a program because no memory is ever returned to the operating system for use by other programs.

For example:

```
int g = 7;                    // global variable, statically allocated

void f( )
{
    int loc = 9;              // local variable, stack allocated
    int* p = new int;         // variable allocated on free store
    // ...
    delete p;                 // return variable pointed to by p for possible re-use
}
```

As far as the programmer is concerned, automatic and static storage are used in simple, obvious, and implicit ways. The interesting question is how to manage the free store. Allocation (using ***new***) is simple, but unless we have a consistent policy for giving memory back to the free store manager, memory will fill up – especially for long-running programs.

The simplest strategy is to use automatic objects to manage corresponding objects in free store.

Consequently, many containers are implemented as handles to elements stored in the free store. For example, a ***string*** (§6.1) variable manages a sequence of characters on the free store:

A ***string*** automatically allocates and frees the memory needed for its elements. For example:

```
void g()
{
    string s = "Time flies when you´re having fun";    // string object created here
    // ...
}                                                      // string object implicitly destroyed here
```

The ***Stack*** example in §4.2.1 shows how constructors and destructors can be used to manage the lifetime of storage for elements. All the standard containers (§6.2), such as ***vector***, ***list***, and ***map***, can be conveniently implemented in this way.

When this simple, regular, and efficient approach isn't sufficient, the programmer might use a memory manager that finds unreferenced objects and reclaims their memory in which to store new objects. This is usually called *automatic garbage collection*, or simply *garbage collection*. Naturally, such a memory manager is called a *garbage collector*. Good commercial and free garbage collectors are available for C++ but a garbage collector is not a standard part of a typical C++ implementation.

3.3 Compile, Link, and Execute

Traditionally, a C or a C++ program consists of a number of source files that are individually compiled into object files. These object files are then linked together to produce the executable form of the program.

Each separately compiled program fragment must contain enough information to allow it to be linked together with other program fragments. Most language rules are checked by the compiler as it compiles an individual source file (translation unit). The linker checks to ensure that names are used consistently in different compilation units and that every name used actually refers to something that has been properly defined. The typical C++ runtime environment performs few checks on the executing code. A programmer who wants run-time checking must provide the tests as part of the source code.

C++ interpreters and dynamic linkers modify this picture only slightly by postponing some checks until the first use of a code fragment.

For example, I might write a simple factorial program and represent it as a separate source file ***fact.c***:

```
// file fact.c:

    #include "fact.h"

    long fact(long f)     // recursive factorial
    {
        if (f>1)
            return f*fact(f-1);
        else
            return 1;
    }
```

A separately compiled program fragment has an interface consisting of the minimal information needed to use it. For this simple ***fact.c*** program fragment, the interface consists of the declaration of ***fact()*** stored in a file ***fact.h***:

```
// file fact.h:

    long fact(long);
```

The interface is ***#include***d in each translation unit that uses it. I also tend to ***#include*** an interface into the translation unit that defines it to give the compiler a chance to diagnose inconsistencies early.

The ***fact()*** function can now be used like this:

```
// file main.c:

    #include "fact.h"
    #include<iostream>

    int main()
    {
        std::cout << "factorial(7) is " << fact(7) << '\n';
        return 0;
    }
```

The function ***main()*** is the starting point for a program, ***iostream*** is the standard C++ I/O library, and ***std::cout*** is the standard character output stream (§6.1). The operator **<<** (''put'') converts values to character strings and outputs those characters. Thus, executing this program will cause

```
factorial(7) is 5040
```

to appear on output followed by a newline (the special character ***\n***).

Graphically, the program fragments can be represented like this:

3.4 Type checking

C++ provides and relies on *static* type checking. That is, most language rules are checked by the compiler before a program starts executing. Each entity in a program has a type and must be used in accordance with its type. For example:

```
int f(double);          // f is a function that takes a double-precision floating point
                        // argument and returns an integer
float x = 2.0;          // x is a single-precision floating point object
string s = "2";         // s is a string of characters
int i = f(x);           // i is an integer
```

The compiler detects inconsistent uses and ensures that conversions defined in the language or by the user are performed. For example:

```
void g()
{
    s = "a string literal";     // ok: convert string literal to string
    s = 7;                      // error: can't convert int to string

    x = "a string literal";     // error: can't convert string literal to float
    x = 7.0;                    // ok
    x = 7;                      // ok: convert int to float

    f(x);                       // ok: convert float to double
    f(i);                       // ok: convert int to double
    f(s);                       // error: can't convert string to double

    double d = f+i;             // ok: add int to float
    string s2 = s+i;            // error: can't add int to string
}
```

For user-defined types, the user has great flexibility in defining which operations and conversions are

acceptable (§6.1). Consequently, the compiler can detect inconsistent use of user-defined types as well as inconsistent use of built-in types.

4 Abstraction

In addition to convenient and efficient mechanisms for expressing computation and allocating objects, we need facilities to manage the complexity of our programs. That is, we need language mechanisms for creating types that are more appropriate to the way we think (to our application domains) than are the low-level built-in features.

4.1 Concrete Types

Small heavily used abstractions are common in many applications. Examples are characters, integers, floating point numbers, complex numbers, points, pointers, coordinates, transforms, (*pointer,offset*) pairs, dates, times, ranges, links, associations, nodes, (*value,unit*) pairs, disc locations, source code locations, ***BCD*** characters, currencies, lines, rectangles, scaled fixed point numbers, numbers with fractions, character strings, vectors, and arrays. Every application uses several of these; a few use them heavily. A typical application uses a few directly and many more indirectly from libraries.

The designer of a general-purpose programming language cannot foresee the detailed needs of every application. Consequently, such as language must provide mechanisms for the user to define such small concrete types. It was an explicit aim of C++ to support the definition and efficient use of such user-defined data types very well. They were seen as the foundation of elegant programming. The simple and mundane is statistically far more significant than the complicated and sophisticated.

Many concrete types are frequently used and subject to a variety of constraints. Consequently, the language facilities supporting their construction emphasizes flexibility and uncompromising time and space efficiency. Where more convenient, higher-level, or safer types are needed, such types can be built on top of simple efficient types. The opposite – building uncompromisingly efficient types on top of more complicated ‘‘higher-level’’ types – cannot be done. Consequently, languages that do not provide facilities for efficient user-defined concrete types need to provide more built-in types, such as lists, strings, and vectors supported by special language rules.

A classical example of a concrete type is a complex number:

```
class complex {
public:          // interface:

    // constructors:
    complex(double r, double i) { re=r; im=i; }   // construct a complex from two scalars
    complex(double r) { re=r; im=0; }              // construct a complex from one scalar
    complex( ) { re = im = 0; }                    // default complex: complex(0,0)

    // access functions:
    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);  // binary minus
    friend complex operator-(complex);           // unary minus
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...

private:
    double re, im; // representation
};
```

This defines a simple ***complex*** number type. Following Simula, the C++ term for user-defined type is *class*. This ***complex*** class specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, ***re*** and ***im*** are accessible only to the functions specified in the declaration of class ***complex***. Restricting the access to the representation to a specific set of functions simplify understanding, eases debugging and testing, and makes it relatively simple to adopt a different implementation if needed.

A member function with the same name as its class is called a *constructor*. Constructors are essential for most user-defined types. They initialize objects, that is, they establish the basic invariants that other

functions accessing the representation can rely on. Class ***complex*** provides three constructors. One makes a ***complex*** from a double-precision floating-point number, another takes a pair of ***double***s, and the third makes a ***complex*** with a default value. For example:

```
complex a = complex(1,2);
complex b = 3;              // initialized by complex(3,0)
complex c;                  // initialized by complex(0,0)
```

A friend declaration grants a function access the representation. Such access functions are defined just like other functions. For example:

```
complex operator+(complex a1, complex a2)   // add two complex numbers
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

This simple ***complex*** type can be used like this:

```
void f()
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b*complex(1,2.3);
    // ...
    c = -(a/b)+2;
}
```

The declaration of ***complex*** specified a representation. This is not necessary for a user-defined type (see §4.2). However, for ***complex***, efficiency and control of data layout are essential. A simple class, such as ***complex***, suffers no space overheads from system-provided ‘‘housekeeping’’ information. Because, the representation of ***complex*** is presented in its declaration, true local variables where all data is stack allocated are trivially implemented. Furthermore, inlining of simple operations is easy for even simple compilers even in the presence of separate compilation. When it comes to supplying acceptable low-level types – such as ***complex***, ***string***, and ***vector*** – for high-performance systems these language aspects are essential [Stroustrup,1994].

Often, notation is an important concern for concrete types. Programmers expect to be able to do complex arithmetic using conventional operators such as `+` and `*`. Similar programmers expect to be able to concatenate strings using some operator (often `+`), to subscript vectors and strings using `[]` or `()`, to invoke objects representing functions using `()`, etc. To meet such expectations, C++ provides the ability to define meanings of operators for user-defined types. Interestingly, the most commonly used operators and the most useful, turns out to be `[]` and `()`, rather than `+` and `-` as most people seem to expect.

The standard C++ library supplies a ***complex*** type defined using the techniques demonstrated here (§6.4.1).

4.2 Abstract Types

Concrete types, as described above, have their representation included in their declaration. This makes it trivial to allocate objects of concrete types on the stack and to inline simple operations on such objects. The resulting efficiency benefits are major. However, the representation of an object cannot be changed without recompiling code taking advantage of such optimizations. This is not always ideal. The obvious alternative is to protect users from any knowledge of and dependency on a representation by excluding it from the class declaration. For example:

```
class Character_device {
public:
    virtual int open(int opt) = 0;          // =0 means ''pure virtual function''
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int . . . ) = 0;
    virtual ~Character_device( ) { }        // destructor (see §4.2.1)
};
```

The word ***virtual*** means ''may be defined later in a class derived from this one'' in Simula and C++. A class derived from ***Character_device*** (see below) provides an implementation of ***Character_device*** interface. The curious ***=0*** syntax says that some class derived from ***Character_device*** *must* define the function.

The ***Character_device*** is an abstract class specifying an interface only. This interface can be implemented in a variety of ways without affecting users. For example, a programmer might use this interface to device drivers on some hypothetical system like this:

```
void user(Character_device* d, char* buffer, int size)
{
    char* p = buffer;

    while (size>chunk_size) {
        if (d->write(p,chunk_size)==chunk_size) {      // whole chunk written
            size -= chunk_size;    // chunk_size characters written
            p += chunk_size;       // move on to next chunk
        }
        else {       // part of chunk written
            // ...
        }
    }
    // ...
}
```

The actual drivers would be specified as classes *derived* from the *base* ***Character_device***. For example:

```
class Dev1 : public Character_device {
    // representation of a Dev1
public:
    int open(int opt);          // open a Dev1
    int close(int opt);         // close a Dev1
    int read(char* p, int n);   // read a Dev1
    // ...
};

class Dev2 : public Character_device {
    // representation of a Dev2
public:
    int open(int opt);          // open a Dev2
    int close(int opt);         // close a Dev2
    int read(char* p, int n);   // read a Dev2
    // ...
};
```

The relationships among the classes can be represented graphically like this:

An arrow represents the *derived from* relationship. The ***user( )*** does not need to know whether a ***Dev1***, or a ***Dev2***, or some other class implementing the ***Character_device*** is actually used.

```
void f(Dev1& d1, Dev2& d2, char* buf, int s)
{
    user(d1,buf,s);      // use a Dev1
    user(d2,buf,s);      // use a Dev2
}
```

A function declared in a derived class is said to *override* a virtual function with the same name and type in a base class. It is the language's job to ensure that calls of ***Character_device***'s virtual functions, such as ***write*( )** invoke the appropriate overriding function for the derived class actually used. The overhead of doing that in C++ is minimal and perfectly predictable. The extra run-time overhead of a ***virtual*** function is a fraction of the cost of an ordinary function call.

We can represent the object layout of a typical implementation like this:

Thus, a virtual function call is simply an indirect function call. No run-time searching for the right function to call is needed.

In many contexts, abstract classes are the ideal way of representing the major internal interfaces of a system. They are simple, efficient, strongly typed, enable the simultaneous use of many different implementations of the concept represented by the interface, and completely insulate users from changes in such implementations.

4.2.1 Destructors

A constructor establishes a context for the member functions of a class to work in for a given object. Often, establishing that context requires the acquisition of resources such as memory, locks, or files. For a program to work correctly, such resources must typically be released when the object is destroyed. Consequently, it is possible to declare a function dedicated to reversing the effect of a constructor. Naturally, such a function is called a *destructor*. The name of a destructor for a class ***X*** is ***~X*( )**; in C++, ~ is the complement operator.

A simple stack of characters can be defined like this:

```
class Stack {
    char* v;
    int max_size;
    int top;
public:
    Stack(int s) { top = 0; v = new T[max_size=s]; }     // constructor: acquire memory
    ~Stack() { delete[] v; }                             // destructor: release memory

    void push(T c) { v[top++] = c; }
    T pop() { return v[--top]; }
};
```

For simplicity, this ***Stack*** has been stripped of all error handling. However, it is complete enough to be used like this:

```
void f(int n)
{
    stack s2(n);    // stack of n characters
```

```
        s2.push('a');
        s2.push('b');
        char c = s2.pop();
        // ...
}
```

Upon entry into *f*( ), *s2* is created and the constructor *Stack*::*Stack*( ) is called. The constructor allocates enough memory for *n* characters. Upon exit from *f*( ), the destructor *Stack*::~*Stack*( ) is implicitly invoked so that the memory acquired by the constructor is freed.

This kind of resource management is often important. For example, an abstract class, such as *Character_device*, will be manipulated through pointers and references and will typically be deleted by a function that has no idea of the exact type of object used to implement the interface. Consequently, a user of a *Character_device* cannot be expected to know what is required to free a device. Conceivably freeing a device would involve nontrivial interactions with an operating system or other guardians of system resources. Declaring *Character_device*'s destructor *virtual* ensures that the removal of the *Character_device* is done using the proper function from the derived class. For example

```
void some_user(Character_device* pd)
{
        // ...
        delete pd; // implicitly invoke object's destructor
}
```

4.3 Object-Oriented Programming

Object-oriented programming is a set of techniques that rely on hierarchies of classes to provide extensibility and flexibility. The basic language facilities used are the user-defined types themselves, the ability to derive a class from another, and *virtual* functions (§4.2). These features allow a programmer to rely on an interface (a class, often an abstract class) without knowing how its operations are implemented. Conversely, they allow new classes to be built directly on older ones without disturbing users of those older classes. As an example, consider the simple task of getting an integer value from a user to an application through some user-interface system. Assuming that we would like to keep the application independent of the details of the user-interface system we could represent the notion of an interaction needed to get an integer as a class *Ival_box*:

```
class Ival_box {
public:
        virtual int get_value( ) = 0;      // get value back to application
        virtual void prompt( ) = 0;        // prompt the user
        // ...
};
```

Naturally, there will be a variety of Ival_boxes:

```
class Ival_dial : public Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
// ...
```

This can be represented graphically like this:

This application hierarchy is independent of the details of an actual user-interface system. The application is written independently of I/O implementation details and then later tied into an implementation hierarchy without affecting the users of the application hierarchy:

A dashed arrow represents a protected base class. A protected base class is one that is part of the implementation of its derived class (only) and is inaccessible general user code. This design makes the application code independent of any change in the implementation hierarchy.

I have used the ***BB*** prefix for realism; suppliers of major libraries traditionally prepend some identifying initials. The superior alternative is to use namespaces (§5.2).

The declaration of a class that ties an application class to the implementation hierarchy will look something like this:

```
class BB_ival_slider : public ival_slider, protected BB_slider {
public:
        // functions overriding Ival_slider functions
        // as needed to implement the application concepts
protected:
        // functions overriding BB_slider and BB_window functions
        // as required to conform to user interface standards
private:
        // representation and other implementation details
};
```

This structure assumes that details of what is to be displayed by a user-interface system is expressed by overriding virtual functions in the ***BB_window***s hierarchy. This may not be the ideal organization of a user interface system, but it is not uncommon.

A derived class inherits properties from its base classes. Thus, derivation is sometimes called *inheritance*. A language, such as C++, that allows a class to have more than one direct base class is said to support *multiple inheritance*.

4.3.1 Run-time Type Identification

A plausible use of the ***Ival_box***es defined in §4.3 would be to hand them to a system that controlled a screen and have that system hand objects back to the application program whenever some activity had occurred. This is how many user interfaces work. However, just as an application using ***Ival_box***es should not know about the user-interface system, the user-interface system will not know about our ***Ival_box***es. The system's interfaces will be specified in terms of the system's own classes and objects rather than our application's classes. This is necessary and proper. However, it does have the unpleasant effect that we lose information about the type of objects passed to the system and later returned to us.

Recovering the ''lost'' type of an object requires us to somehow ask the object to reveal its type. Any operation on an object requires us to have a pointer or reference of a suitable type for the object. Consequently, the most obvious and useful operation for inspecting the type of an object at run time is a type conversion operation that returns a valid pointer if the object is of the expected type and a null pointer if it isn't. The ***dynamic_cast*** operator does exactly that. For example, assume that ''the system'' invokes ***my_event_handler***( ) with a pointer to a ***BBwindow***, where an activity has occurred:

```
void my_event_handler(BBwindow* pw)
{
    if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))    { // does pw point to an Ival_box?
        int i = pb->get_value();
        // ...
    }
    else {
        // Oops! unexpected event
    }
}
```

One way of explaining what is going on is that ***dynamic_cast*** translates from the implementation-oriented language of the user-interface system to the language of the application. It is important to note what is *not* mentioned in this example: the actual type of the object. The object will be a particular kind of ***Ival_box***, say an ***Ival_slider***, implemented by a particular kind of ***BBwindow***, say a ***BBslider***. It is neither necessary nor desirable to make the actual type of the object explicit in this interaction between ''the system'' and the application. An interface exists to represent the essentials of an interaction. In particular, a well-designed interface hides inessential details.

Casting from a base class to a derived class is often called a *downcast* because of the convention of drawing inheritance trees growing from the root down. Similarly, a cast from a derived class to a base is called an *upcast*. A cast that goes from a base to a sibling class, like the cast from ***BBwindow*** to ***Ival_box***, is called a *crosscast*.

4.4 Generic Programming

Given classes and class hierarchies, we can elegantly and efficiently represent individual concepts and also represent concepts that relate to each other in a hierarchical manner. However, some common and important concepts are neither independent of each other nor hierarchically organized. For example, the notions ''vector of integer'' and ''vector of complex number'' are related through the common concept of a vector and differ in the type of the vector elements (only). Such abstractions are best represented through parameterization. For example, the ***vector*** should be parameterized by the element type.

C++ provides parameterization by type through the notion of a template. It was a crucial design criterion that templates should be flexible and efficient enough to be used to define fundamental containers with severe efficiently constraints. In particular, the aim was to be able to provide a ***vector*** template class that did not impose run-time or space overheads compared to a built-in array.

4.5 Containers

We can generalize a stack-of-characters type from §4.2.1 to a stack-of-anything type by making it a *template* and replacing the specific type ***char*** with a template parameter. For example:

```
template<class T> class Stack {
    T* v;
    int max_size;
    int top;
public:
    Stack(int s) { top = 0;  v = new T[max_size=s]; }   // constructor
    ~Stack() { delete[] v; }                            // destructor

    void push(T c) { v[top++] = c; }
    T pop() { v[--top]; }
};
```

The ***template<class T>*** prefix makes ***T*** a parameter of the declaration it prefixes.

We can now use stacks like this:

```
Stack<char> sc(100);              // stack of characters
Stack<complex> scplx(200);        // stack of complex numbers
Stack< list<int> > sli(400);      // stack of list of integers
```

```
void f()
{
    sc.push('c');
    if (sc.pop() != 'c') error("impossible");

    scplx.push(complex(1,2));
    if (scplx.pop() != complex(1,2)) error("can't happen");
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates. A class holding a collection of elements of some type is commonly called a *container class*, or simply a *container*.

Templates are a compile-time mechanism so that their use incurs no run-time overhead compared to ''hand-written code.''

4.5.1 Algorithms

Given a variety of semantically similar types – such as a set of containers that all support similar operations for element insertion and access – we can write code that works for all of those types. For example, we might count the occurrences of a value ***val*** in a sequence of elements delimited by ***first*** and ***last*** like this:

```
template<class In, class T> int count(In first, In last, const T& val)
{
    int res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

This code assumes only that values of type ***T*** can be compared using ==, that an ***In*** can be used to traverse the sequence by using ++ to get to the next element, and that ***p** retrieves the value of the element pointer to by an iterator ***p***. For example:

```
void f(vector<complex>& vc, string s, list<int>& li)
{
    int c1 = count(vc.begin(), vc.end(), complex(0));
    int c2 = count(s.begin(), s.end(), 'x');
    int c3 = count(li.begin(), li.end(), 42);
    // ...
}
```

This counts the occurrences of the ***complex 0*** in the vector, the occurrences of ***x*** in the string, and the occurrences of ***42*** in the list.

A type with the properties specified for ***In*** is called an *iterator*. The simplest example of an iterator is a built-in pointer. Standard library containers – such as ***vector***, ***string***, and ***list*** – all provide the operations ***begin***() and ***end***() that returns iterators for the first element and the one-beyond-the-last element, respectively; thus, ***begin***()..***end***() describes a half-open sequence (§6.3). Naturally, the implementations of ++ and * differ for the different containers, but such implementation details don't affect the way we write the code.

5 Large-scale Programming

The main part of this paper is organized around the key programming styles supported by C++. However, namespaces and exception handling mechanisms are important language features that do not fit this classification because they support large-scale programming in all styles. They are used to ease the construction of programs out of separate parts and they increase in importance as the size of programs increase.

5.1 Exceptions and Error Handling

Exceptions are used to transfer control from a place where an error is detected to some caller that has expressed interest in handling that kind of errors. Clearly, this is a mechanism that should be used only for errors that cannot be handled locally.

One might ask: ''How can something that is (eventually) handled correctly so that the program proceeds as intended be considered an error?'' Consequently, we often refer to exceptional events, or simply to

exceptions, and the language mechanisms provided to deal with these events *exception handling*.

Consider how we might report underflow and overflow errors from a ***Stack***:

```
template<class T> class Stack {
    T* v;
    int max_size;
    int top;
public:
    class Underflow { }; // type used to report underflow
    class Overflow { }; // type used to report overflow

    Stack(int s);          // constructor
    ~Stack();              // destructor

    void push(T c)
    {
        if (top == max_size) throw Overflow();    // check for error
        v[top++] = c;                             // raise top and place c on top
    }

    T pop()
    {
        if (top == 0) throw Underflow();    // check for error
        return v[--top];                    // lower top
    }
};
```

An object thrown can be caught by a caller of the function that threw. For example:

```
void f()
{
    Stack<string> ss(10);

    try {
        ss.push("Quiz");
        string s = ss.pop();          // pop "Quiz"
        ss.pop();                     // try to pop an empty string: will throw Underflow
    }
    catch(Stack<string>::Underflow) {   // exception handler
        cerr << "error: stack underflow";
        return;
    }
    catch(Stack<string>::Overflow) {    // exception handler
        cerr << "error: stack overflow";
        return;
    }
    // ...
}
```

Exceptions thrown within a ***try*** { ... } block or in code called from within a try block will be caught by a catch-clause for their type.

Exceptions can be used to make error handling more stylized and regular. In particular, hierarchies of exception classes can be used to group exceptions so that a piece of code need deal only with errors at a suitable level of detail. For example, assume that ***open_file_error*** is a class derived from ***io_error***:

```
void use_file(const char* fn)
{
    File_ptr f(fn,"r");  // open fn for reading; throw open_file_error if that can't be done
    // use f
}
```

```
void some_fct()
{
    try {
        // ...
        use_file("homedir/magic");
        // ...
    }
    catch (io_error) {
        // ...
    }
}
```

Here, ***some_fct*** need not know the details of what went wrong; that is, it need not know about ***open_file_error***. Instead, it deals with errors at the ***io_error*** level of abstraction.

Note that ***use_file*** didn't deal with exceptions at all. It simply opens a file and uses it. Should the file not be there to open, control is immediately transferred to the caller ***some_fct***. Similarly, should a read error occur during use, the file will be properly closed by the ***File_ptr*** destructor before control returns to ***some_fct***.

5.2 Namespaces

A namespace is a named scope. Namespaces are used to group related declarations and to keep separate items separate. For example, two separately developed libraries may use the same name to refer to different items, but a user can still use both:

```
namespace Mylib {
    template<class T> class Stack { /* ... */ };
    // ...
}

namespace Yourlib {
    class Stack { /* ... */ };
    // ...
}

void f(int max)
{
    Mylib::Stack<int> s1(max);      // use my stack
    Yourlib::Stack s2(max);         // use your stack
    // ...
}
```

Repeating a namespace name can be a distraction for both readers and writers. Consequently, it is possible to state that names from a particular namespace are available without explicit qualification. For example:

```
void f(int max)
{
    using namespace Mylib;   // make names from Mylib accessible

    Stack<int> s1(max);             // use my stack
    Yourlib::Stack s2(max);         // use your stack
    // ...
}
```

Namespaces provide a powerful tool for the management of different libraries and of different versions of code. In particular, they offer the programmer alternatives of how explicit to make a reference to a non-local name.

6 The C++ Standard Library

The standard library provides:

[1] Basic run-time language support (e.g., for allocation and run-time type information).
[2] The C standard library (with very minor modifications to minimize violations of the type system).
[3] Strings and I/O streams (with support for international character sets and localization).

[4] A framework of containers (such as ***vector***, ***list***, and ***map***) and algorithms using containers (such as general traversals, sorts, and merges).
[5] Support for numerical computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization).

The main criterion for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be easy to learn. Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

The framework of containers, algorithms, and iterators is commonly referred to as the STL. It is primarily the work of Alexander Stepanov [Stepanov,1994].

6.1 Strings and I/O

Strings and input/output operations are not provided directly by special language constructs in C++. Instead, the standard library provides string and I/O types. For example:

```
#include<string>      // make standard strings available
#include<iostream>    // make standard I/O available

int main()
{
    using namespace std;

    string name;
    cout << "Please enter your name: ";      // prompt the user
    cin >> name;                             // read a name
    cout << "Hello, " << name << '\n';       // output the name followed by a newline
    return 0;
}
```

This example uses the standard input and output streams, ***cin*** and ***cout*** with their operators **>>** (‘‘get from’’) and **<<** (‘‘put to’’).

The I/O streams support a variety of formatting and buffering facilities, and strings support common string operations such as concatenation, insertion, and extraction of characters and strings. Both streams and strings can be used with characters of any character set.

The standard library facilities can be – and usually are – implemented using only facilities available to all users. Consequently, where the standard facilities happen to be inadequate, a user can provide equally elegant alternatives.

6.2 Containers

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

Standard Container Summary	
vector<T>	A variable-sized vector
list<T>	A doubly-linked list
queue<T>	A queue
stack<T>	A stack
deque<T>	A double-ended queue
priority_queue<T>	A queue sorted by value
set<T>	A set
multiset<T>	A set in which a value can occur many times
map<key,val>	An associative array
multimap<key,val>	A map in which a key can occur many times

These containers are designed to be efficient yet have interfaces designed to ensure that they can be used interchangeably wherever reasonable. For example, like ***list***, ***vector*** provides efficient operations for adding elements to its end (back). This allows data that eventually needs to be efficiently accessed by

subscripting to be constructed incrementally. For example:

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;

    while (cin >> buf) {                    // read points from input
        if (buf == sentinel) return;
        // check new point
        cities.push_back(buf);              // add point to (end of) vector
    }
}
```

The containers are nonintrusive; that is, essentially every type can be used as a container element type. In particular, built-in types – such as ***int*** and ***char**** – and C-style data structures (***struct***s) can be container elements.

6.3 Algorithms

The standard library provides dozens of algorithms. The algorithms are defined in namespace ***std*** and presented in the ***<algorithm>*** header. Here are a few I have found particularly useful:

Selected Standard Algorithms	
for_each()	Invoke function for each element
find()	Find first occurrence of arguments
find_if()	Find first match of predicate
count()	Count occurrences of element
count_if()	Count matches of predicate
replace()	Replace element with new value
replace_if()	Replace element that matches predicate with new value
copy()	Copy elements
unique_copy()	Copy elements that are not adjacent duplicates
sort()	Sort elements
equal_range()	Find all elements with equivalent values
merge()	Merge sorted sequences

These algorithms, and many more, can be applied to of standard containers, ***string***s, and built-in arrays. In fact, they can be applied to any sequence as described in §4.5.1:

As mentioned, operator * is used to mean ‘‘access an element through an iterator’’ and operator ++ to mean ‘‘make the iterator refer to the next element.’’ For example, we can define a general algorithm that finds the first element of a sequence that matches a predicate like this:

```
template <class In, class Predicate> In find_if(In first, In last, Predicate pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

Simple definitions, the ability to chose the right algorithm at compile time based on the type of the input sequence, and the ability to inline simple operations (such as ==, <, and simple user-defined predicates) mean that these very generic and general algorithms outperform most conventional alternatives.

6.4 Numerics

Like C, C++ wasn't designed primarily with numerical computation in mind. However, a lot of numerical work is done in C++, and the standard library reflects that.

6.4.1 Complex Numbers

The standard library supports a family of complex number types along the lines of the ***complex*** class described in §4.1. To support complex numbers where the scalars are single-precision floating-point numbers (***float***s), double precision numbers (***double***s), etc., the standard library ***complex*** is a template:

```
template<class scalar> class complex {
public:
    complex(scalar re, scalar im);
    // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
template<class C> complex<C> pow(const complex<C>&, int);  // exponentiation

void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld = fl+sqrt(db);
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

Thus, complex numbers are supported approximately to the degree that floating-point numbers are.

6.4.2 Vector Arithmetic

The standard ***vector*** from §6.2 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to ***vector*** would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides a vector, called ***valarray***, that is less general and more amenable to optimization for numerical computation:

```
template<class T> class valarray {
    // ...
    T& operator[](size_t);
    // ...
};
```

The type ***size_t*** is the unsigned integer type that the implementation uses for array indices.

The usual arithmetic operations and the most common mathematical functions are supported for ***valarray***s. For example:

```
template<class T> valarray<T> abs(const valarray<T>&);  // absolute value

void f(const valarray<double>& a1, const valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;
    a += a2*3.14;
    valarray<double> aa = abs(a);
    double d = a2[7];
    // ...
}
```

The ***valarray*** type also supports BLAS-style and generalized slicing. More complicated numeric types, such as ***Matrix***, can be constructed from ***valarray***.

7 Use of C++

C++ is used by hundreds of thousands of programmers in essentially every application domain. This use is supported by about a dozen independent implementations, hundreds of libraries, hundreds of textbooks, several technical journals, many conferences, and innumerable consultants. Training and education at a variety of levels are widely available.

Early applications tended to have a strong systems programming flavor. For example, several major operating systems have been written in C++ [Campbell,1987] [Rozier,1988] [Hamilton,1993] [Berg,1995] [Parrington,1995] and many more have key parts done in C++. C++ was designed so that every language feature is usable in code under severe time and space constraints [Stroustrup,1994]. This allows C++ to be used for device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed. Often, so is compactness of the resulting system.

Most applications have sections of code that are critical for acceptable performance. However, the largest amount of code is not in such sections. For most code, maintainability, ease of extension, and ease of testing is key. C++'s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over time. Examples are banking, trading, insurance, telecommunications, and military applications. For years, the central control of the U.S. long-distance telephone system has relied on C++ and every 800 call (that is, a call paid for by the called party) has been routed by a C++ program [Kamath,1993]. Many such applications are large and long-lived. As a result, stability, compatibility, and scalability have been constant concerns in the development of C++. Million-line C++ programs are not uncommon.

Like C, C++ wasn't specifically designed with numerical computation in mind. However, much numerical, scientific, and engineering computation is done in C++. A major reason for this is that traditional numerical work must often be combined with graphics and with computations relying on data structures that don't fit into the traditional Fortran mold [Budge,1992] [Barton,1994]. Graphics and user interfaces are areas in which C++ is heavily used.

All of this points to what may be C++'s greatest strength: its ability to be used effectively for applications that require work in a variety of application areas. It is quite common to find an application that involves local and wide-area networking, numerics, graphics, user interaction, and database access. Traditionally, such application areas have been considered distinct, and they have most often been served by distinct technical communities using a variety of programming languages. However, C++ has been widely used in all of those areas. Furthermore, it is able to coexist with code fragments and programs written in other languages.

C++ is widely used for teaching and research. This has surprised some who – correctly – point out that C++ isn't the smallest or cleanest language ever designed. However, C++ is

- clean enough for successful teaching of basic concepts,
- realistic, efficient, and flexible enough for demanding projects,
- available enough for organizations and collaborations relying on diverse development and execution environments,
- comprehensive enough to be a vehicle for teaching advanced concepts and techniques, and
- commercial enough to be a vehicle for putting what is learned into non-academic use.

Thanks to the ISO standards process (§2), C++ is also well-specified, stable, and supported by a standard library.

8 Further Reading

There is an immense amount of literature on C++, Object-oriented Programming, and Object-Oriented Design. Here is a short list of books that provides information on key aspects of C++ and its use.

[Stroustrup,1997] is a tutorial for experienced programmers and user-level reference for C++ and its standard library; it presents a variety of fundamental and advanced design and programming techniques. [Stroustrup,1994] describes the rationale behind the design choices for C++.

[Koenig,1997] is a collection of essays discussing ways of using C++ effectively. [Barton,1994] focuses on numeric computation and presents some advanced uses of templates. [Cline,1995] gives practical answers to many questions that occur to programmers starting to use C++.

Other books discuss C++ primarily in the context of design. [Booch,1994] presents the general notion of Object-Oriented Design, and [Martin,1995] gives detailed examples of Booch's design method, [Gamma,1994] introduces the notion of design patterns. These three books all provide extensive examples of C++ code.

These books are all aimed at experienced programmers and designers. There is also a host of C++ books aimed at people with weak programming experience, but the selection of those varies so rapidly that a specific recommendation would be inappropriate.

9 Acknowledgements

This paper was written in grateful memory of my CRC Standard Mathematical Tables, 17th edition: an essential tool, status symbol, and security blanket for a young Math student.

10 References

[Barton,1994] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 1-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38 No. 10. October 1995.

[Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Bj rn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.

[Booch,1994] Grady Booch: *Object-Oriented Analysis and Design*. Benjamin/Cummings. Menlo Park, California. 1994. ISBN 0-8053-5340-2.

[Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

[C++,1998] X3 Secretariat: *Standard – The C++ Language*. ISO/IEC:98-14882. Information Technology Council (NSITC). Washington, DC, USA.

[Campbell,1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Cline,1995] Marshall P. Cline and Greg A. Lomow: *C++ FAQs: Frequently Asked Questions*. Addison-Wesley. Reading. Mass. 1995 ISBN 0-20158958-3.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction* in *Structured Programming*. Academic Press, New York. 1972.

[Gamma,1995] Eric Gamma, et al.: *Design Patterns*. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-63361-2.

[Ellis,1990] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-201-51459-1.

[Hamilton,1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Objects*. Proc. 1993 Summer USENIX Conference. USENIX.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72 No. 5. September/October 1993.

[Kernighan,1978] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall 1978. ISBN 0-13-110163-3.

[Kernighan,1981] Brian Kernighan: *Why Pascal is not my Favorite Programming Language*. AT&T Bell Labs Computer Science Technical Report No. 100. July 1981.

[Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall 1988. ISBN 0-13-110362-8.

[Koenig,1997] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Addison Wesley Longman. Reading, Mass. 1997. ISBN 1-201-42339-1.

[Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

[Parrington,1995] Graham Parrington et al.: *The Design and Implementation of Arjuna*. Computer Systems. Vol. 8 No. 3. Summer 1995.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup,1980] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Also Sigplan Notices January, 1982.

[Stroustrup,1982] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution*. Bell Laboratories Computer Science internal document. April 1982. Software Practice & Experience, Vol.13. 1983. pp. 139-61.

[Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1986. ISBN 0-201-12078-X.

[Stroustrup,1986b] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in *IEEE Software Magazine*. May 1988.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[Stroustrup,1996] Bjarne Stroustrup: *A History of C++:1979-1991*. In *The History of Programming Languages* (T. J. Bergin and R. G. Gibson, editors), pp 699-754. Addison-Wesley. 1996. ISBN 0-201-89502-1. Originally published in Proc. ACM History of Programming Languages Conference (HOPL-2). April 1993. ACM SIGPLAN Notices. March 1993.

[Stroustrup,1997] Bjarne Stroustrup: *The C++ Programming language (Third Edition)*. Addison-Wesley. 1997. ISBN 0-201-88954-4.

[Stroustrup,1997b] Bjarne Stroustrup: *A History of C++*. in *Handbook of Programming Languages* Vol. 1. (Editor: Peter H. Salus). MacMillan Technical Publishing. 1997.

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/269031890

A history of C++

Conference Paper *in* ACM SIGPLAN Notices · March 1993

DOI: 10.1145/154766.155375

CITATIONS
7

READS
451

1 author:

Bjarne Stroustrup
Morgan Stanley

157 PUBLICATIONS **8,359** CITATIONS

SEE PROFILE

All content following this page was uploaded by Bjarne Stroustrup on 18 October 2018.

The user has requested enhancement of the downloaded file.

A History of C++: 1979–1991

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper outlines the history of the C++ programming language. The emphasis is on the ideas, constraints, and people that shaped the language, rather than the minutiae of language features. Key design decisions relating to language features are discussed, but the focus is on the overall design goals and practical constraints. The evolution of C++ is traced from C with Classes to the current ANSI and ISO standards work and the explosion of use, interest, commercial activity, compilers, tools, environments, and libraries.

1 Introduction

C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. It was intended to deliver that to real projects within half a year of the idea. It succeeded.

At the time, I realized neither the modesty nor the preposterousness of that goal. The goal was modest in that it did not involve innovation, and preposterous in both its time scale and its Draconian demands on efficiency and flexibility. While a modest amount of innovation did emerge over the years, efficiency and flexibility have been maintained without compromise. While the goals for C++ have been refined, elaborated, and made more explicit over the years, C++ as used today directly reflects its original aims.

This paper is organized in roughly chronological order:

§2 *C with Classes: 1979–1983*. This section describes the fundamental design decisions for C++ as they were made for C++'s immediate predecessor.

§3 *From C with Classes to C++: 1982–1985*. This section describes how C++ evolved from C with Classes up until the first commercial release and the printing of the book that defined C++ in October 1985.

§4 *Release 2.0: 1985–1988*. This section describes how C++ evolved during the early years of commercial availability.

§5 *The Explosion in Interest and Use: 1987–*. This section deals with non-language factors, such as the growth of a C++ tools and library industry. It also tries to estimate the impact of commercial competition on the development of C++.

§6 *Standardization: 1988–*. This section describes the way C++ continues to evolve under the pressures of heavy use in diverse application areas, and how the C++ community handles this challenge through formal ISO and ANSI standardization.

§7 *Retrospective.* This section considers how C++ met its design goals, how it might have been a better language, and how it might become an even more useful tool.

Most effort have been expended on the early years because the design decisions taken early determined the further development of the language. It is also easier to maintain a historical

perspective when one has had many years to observe the consequences of decisions.

Essential language features are presented to make this paper approachable by a non-C++ specialist. However, the emphasis in on the people, ideas, and constraints that shaped C++ rather than on detailed descriptions of those language features or their use. For a description of what C++ is today and how to use it see [Stroustrup,1991]†.

2 C with Classes

C++ evolved from an earlier version called C with Classes. The work and experience with C with Classes from 1979 to 1983 determined the shape of C++.

2.1 Prehistory

The prehistory of C++ – the couple of years before the idea of adding Simula-like features to C occurred to me – is important because during this time the criteria and ideals that later shaped C++ emerged. I was working on my Ph.D. thesis in the Computing Laboratory of Cambridge University in England. My aim was to study alternatives for the organization of system software for a distributed system. The conceptual framework was provided by the capability-based Cambridge CAP computer and its experimental and continuously evolving operating system. The details of this work and its outcome [Stroustrup,1979a] are of little relevance to C++. What is relevant, though, was the focus on composing software out of well-delimited modules and that the main experimental tool was a relatively large and detailed simulator I wrote for simulating software running on a distributed system.

The initial version of this simulator was written in Simula and ran on the Cambridge University computer center's IBM 360/165 mainframe. It was a pleasure to write that simulator. The features of Simula were almost ideal for the purpose and I was particularly impressed by the way the concepts of the language helped me think about the problems in my application. The class concept allowed me to map my application concepts into the language constructs in a direct way that made my code more readable than I had seen in any other language. The way Simula classes can act as co-routines made the inherent concurrency of my application easy to express. For example, an object of class `computer` could trivially be made to work in pseudo-parallel with other objects of class `computer`. Class hierarchies were used to express variants of application level concepts. For example, different types of computers could be expressed as classes derived from class `computer` and different types of inter-module communication mechanisms could be expressed as classes derived from class `IPC`. The use of class hierarchies was not heavy, though; the use of classes to express concurrency was much more important in the organization of my simulator.

During writing and initial debugging I acquired a great respect for the expressiveness of Simula's type system and the ability of its compiler's ability to catch type errors. The observation was that a type error almost invariably reflected either a silly programming error or a conceptual flaw in the design. The latter was by far the most significant and a help that I had not experienced in the use of more primitive ''strong'' type systems. In contrast, I had found Pascal's type system worse than useless – a strait jacket that caused more problems than it solved by forcing me to warp my designs to suit an implementation-oriented artifact. The perceived contrast between the rigidity of Pascal and the flexibility of Simula was essential for the development of C++. Simula's class concept was seen as the key difference and ever since I have seen classes as the proper primary focus of program design.

---

† Author's note: It is now 1995 – 4 years since I first completed this paper. Rather than rewriting major sections of this paper to for the final book, I have added footnotes where needed to reflect recent events. Most of the themes of this paper are explored further in [Stroustrup,1994]. So are many issues related to the design and evolution that couldn't be included here.

I had used Simula before (during my studies at the University of Aarhus, Denmark), but was very pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased. The class and co-routine mechanisms of Simula and the comprehensive type checking mechanisms ensured that problems and errors did not (as I – and I guess most people – would have expected) grow linearly or more than linearly with the size of the program. Instead, the total program acted more like a collection of small (and therefore easy to write, comprehend, and debug) programs rather than a single large program.

The implementation of Simula, however, did not scale in the same way and as a result the whole project came close to disaster. My conclusion at the time was that the Simula implementation (as opposed to the Simula language) was geared to relatively small programs and was inherently unsuited for larger programs [Stroustrup,1979a]. Link times for separately compiled classes were abysmal: It took longer to compile 1/30th of the program and link it to a precompiled version of the rest than it took to compile and link the program as a monolith. This I believe to be more a problem with the mainframe linker than with Simula, but it was still a burden. On top of that, the run-time performance was such that there was no hope of using the simulator to obtain real data. The poor run-time characteristics were a function of the language and its implementation rather than a function of the application. The overhead problems were fundamental to Simula and could not be remedied. The cost arose from several language features and their interactions: run-time type checking, guaranteed initialization of variables, concurrency support, and garbage collection of both user-allocated objects and procedure activation records. For example, measurements showed that more than 80% of the time was spent in the garbage collector despite the fact that resource management was part of the the simulated system so that no garbage was ever produced. Simula implementations are better these days (15 years later), but the order-of-magnitude improvement relative to systems programming languages still has not (to the best of my knowledge) materialized.

To avoid terminating the project I re-wrote the simulator in BCPL and ran it on the experimental CAP computer. The experience of coding and debugging the simulator in BCPL was horrible. BCPL makes C look like a very high level language and provides absolutely no type checking or run-time support. The resulting simulator did, however, run suitably fast and gave a whole range of useful results that clarified many issues for me and provided the basis for several papers on operating system issues [Stroustrup,1978] [Stroustrup,1979b] [Stroustrup,1981a].

Upon leaving Cambridge, I swore never again to attack a problem with tools as unsuitable as those I had suffered while designing and implementing the simulator. The significance of this to C++ was the notion I had evolved of what constituted a ‘‘suitable tool’’ for projects such as the writing of a significant simulator, an operating system, and similar systems programming tasks:

[1] A good tool would have Simula’s support for program organization – that is, classes, some form of class hierarchies, some form of support for concurrency, and strong (that is, static) checking of a type system based on classes. This I saw as support for the process of inventing programs, as support for design rather than just support for implementation.

[2] A good tool would produce programs that ran as fast as BCPL programs and share BCPL’s ability to easily combine separately compiled units into a program. A simple linkage convention is essential for combining units written in languages such as C, Algol68, Fortran, BCPL, assembler, etc., into a single program and thus not to get caught by inherent limitations in a single language.

[3] A good tool should also allow for highly portable implementations. My experience was that the ‘‘good’’ implementation I needed would typically not be available until ‘‘next year’’ and only on a machine I couldn’t afford. This implied that a tool must have multiple sources of implementations (no monopoly would be sufficiently responsive to users of ‘‘unusual’’ machines and to poor graduate students), that there should be no complicated run-time support system to port, and that there should be only very limited integration

between the tool and its host operating system.

Not all of these criteria were fully formed when I left Cambridge, but several were and more matured on further reflection on my experience with the simulator, on programs written over the next couple of years, and on the experiences of others as learned through discussions and reading of code. C++ as defined at the time of release 2.0 strictly fulfills these criteria; the fundamental tensions in the effort to design templates and exception handling mechanisms for C++ arise from the need to depart from some aspects of these criteria. I think the most important aspect of these criteria is that they are only loosely connected with specific programming language features. Rather, they specify constraints on a solution.

My background in operating systems work and my interest in modularization and communication had permanent effects on C++. The C++ model of protection, for example, is based on the notion of granting and transferring access rights, the distinction between initialization and assignment has its root in thoughts about transferring capabilities, and the design of C++'s exception handling mechanism was influenced by work on fault tolerant systems done by Brian Randell's group in Newcastle in the seventies.

2.2 The Birth of C with Classes

The work on what eventually became C++ started with an attempt to analyze the UNIX kernel to determine to what extent it could be distributed over a network of computers connected by a local area network. This work started in April of 1979 in the Computing Science Research Center of Bell Laboratories in Murray Hill, New Jersey, where I have worked ever since. Two sub-problems soon emerged: how to analyze the network traffic that would result from the kernel distribution and how to modularize the kernel. Both required a way to express the module structure of a complex system and the communication pattern of the modules. This was exactly the kind of problem that I had become determined never to attack again without proper tools. Consequently, I set about developing a proper tool according to the criteria I had formed in Cambridge.

In October of 1979 I had a pre-processor, called Cpre, that added Simula-like classes to C running and in March of 1980 this pre-processor had been refined to the point where it supported one real project and several experiments. My records show the pre-processor in use on 16 systems by then. The first key C++ library, the task system supporting a co-routine style of programming [Stroustrup,1980b] [Stroustrup,1987b] [Shopiro,1987], was crucial to the usefulness of ''C with Classes,'' as the language accepted by the pre-processor was called, in these projects.

During the April to October period the transition from thinking about a ''tool'' to thinking about a ''language'' had occurred, but C with Classes was still thought of primarily as an extension to C for expressing modularity and concurrency. A crucial decision had been made, though. Even though support of concurrency and Simula-style simulations was a primary aim of C with Classes, the language contained no primitives for expressing concurrency; rather, a combination of inheritance (class hierarchies) and the ability to define class member functions with special meanings recognized by the pre-processor was used to write the library that supported the desired styles of concurrency. Please note that ''styles'' is plural. I considered it crucial – as I still do – that more than one notion of concurrency should be expressible in the language. This decision has been reconfirmed repeatedly by me and my colleagues, by other C++ users, and by the C++ standards committee. There are many applications for which support for concurrency is essential, but there is no one dominant model for concurrency support; thus when support is needed it should be provided through a library or a special purpose extension so that a particular form of concurrency support does not preclude other forms.

Thus, the language provided general mechanisms for organizing programs rather than support for specific application areas. This was what made C with Classes and later C++ a general-purpose language rather than a C variant with extensions to support specialized applications. Later, the choice between providing support for specialized applications or general abstraction

mechanisms has come up repeatedly. Each time the decision has been to improve the abstraction mechanisms.

An early description of C with Classes was published as a Bell Labs technical report in April 1980 [Stroustrup,1980a] and later in SIGPLAN Notices. The SIGPLAN paper was in April 1982 followed by a more detailed Bell Labs technical report *Adding Classes to the C Language: An Exercise in Language Evolution* [Stroustrup,1982] that was later published in *Software: Practice and Experience*. These papers set a good example by describing only features that were fully implemented and had been used. This was in accordance with a long standing tradition of Bell Labs Computing Science Research Center; that policy has been modified only where more openness about the future of C++ became needed to ensure a free and open debate over the evolution of C++ among its many non-AT&T users.

C with Classes was explicitly designed to allow better organization of programs; ''computation'' was considered a problem solved by C. I was very concerned that improved program structure was not achieved at the expense of run-time overheads compared to C. The explicit aim was to match C in terms of run-time, code compactness, and data compactness. To wit: Someone once demonstrated a 3% systematic decrease in overall run-time efficiency compared with C. This was considered unacceptable and the overhead promptly removed. Similarly, to ensure layout compatibility with C and thereby avoid space overheads, no ''house-keeping data'' was placed in class objects.

Another major concern was to avoid restrictions on the domain where C with Classes could be used. The ideal – which was achieved – was that C with Classes could be used for whatever C could be used for. This implied that in addition to matching C in efficiency, C with Classes could not provide benefits at the expense of removing ''dangerous'' or ''ugly'' features of C. This observation/principle had to be repeated often to people (rarely C with Classes users) who wanted C with Classes made safer by increasing static type checking along the lines of early Pascal. The alternative way of providing ''safety,'' inserting run-time checks for all unsafe operations, was (and is) considered reasonable for debugging environments but the language could not guarantee such checks without leaving C with a large advantage in run-time and space efficiency. Consequently, such checks were not provided for C with Classes, though C++ environments exist that provide such checks for debugging. In addition, users can and do insert run-time checks (assertions [Stroustrup,1991]) where needed and affordable.

C allows quite low-level operations such as bit manipulation and choosing between different sizes of integers. There are also facilities, such as explicit unchecked type conversions, for deliberately breaking the type system. C with Classes and later C++ follow this path by retaining the low-level and unsafe features of C. In contrast to C, C++ systematically eliminates the need to use such features except where they are essential and performs unsafe operations only at the explicit request of the programmer. I strongly felt then, as I still do, that there is no one right way of writing every program and a language designer has no business of trying to *force* programmers to use a particular style. The language designer does, on the other hand, have an obligation to encourage and support a variety of styles and practices that have proven effective and to provide language features and tools to help programmers avoid the well known traps and pitfalls.

2.3 Feature overview

The features provided in the initial 1980 implementation can be summarized:

[1] classes,
[2] derived classes,
[3] public/private access control,
[4] constructors and destructors,
[5] call and return functions (§2.4.8),
[6] `friend` classes,

[7] type checking and conversion of function arguments

During 1981 three more features were added:

[8] inline functions,

[9] default arguments,

[10] overloading of the assignment operator

Since a pre-processor was used for the implementation of C with Classes, only new features, that is features not present in C, needed to be described and the full power of C was directly available to users. Both of these aspects were appreciated at the time. Having C as a subset dramatically reduced the support and documentation work needed. This was most important because for several years I did all of the C with Classes and later C++ documentation and support in addition to doing the experimentation, design, and implementation. Having all C features available further ensured that no limitations introduced through prejudice or lack of foresight on my part would deprive a user of features already available in C. Naturally, portability to machines supporting C was ensured. Initially, C with Classes was implemented and used on a DEC PDP/11 but soon it was ported to machines such as DEC VAX and Motorola 68000 based machines.

C with Classes was still seen as a dialect of C. Furthermore, classes were referred to as ''An Abstract Data Type Facility for the C Language'' [Stroustrup,1980a]. Support for object-oriented programming was not claimed until the provision of virtual functions in C++ in 1983 [Stroustrup,1984a].

2.4 Feature Details

Clearly, the most important aspect of C with Classes – and later of C++ – was the class concept. Many aspects of the C with Classes class concept can be observed by examining a simple example from [Stroustrup,1980a]:

```
class stack {
    char    s[SIZE];  /* array of characters */
    char *  min;      /* pointer to bottom of stack */
    char *  top;      /* pointer to top of stack */
    char *  max;      /* pointer to top of allocated space */
    void    new();    /* initialization function (constructor) */
public:
    void    push(char);
    char    pop();
};
```

A class is a user-defined data type. A class specifies the type of the class members that define the representation of a variable of the type (an object of the class), specifies the set of operations (functions) that manipulate such objects, and specifies the access users have to these members. Member functions are typically defined ''elsewhere:''

```
char stack.pop()
{
    if (top <= min) error("stack underflow");
    return *(--top);
}
```

Objects of class `stack` can now be defined and used:

```
class stack s1, s2;           /* two variables of class 'stack' */
class stack * p1 = &s2;       /* 'p1' points to 's2' */
class stack * p2 = new stack; /* 'p2' points to stack object
                                 allocated on free store */

s1.push('h');   /* use object directly */
p1->push('s');  /* use object through pointer */
```

Several key design decisions are reflected here:

[1] C with Classes follows Simula in letting the programmer specify types from which variables (objects) can be created, rather than, say, the Modula approach of specifying a module as a collection of objects and functions. In C with Classes (as in C++), a class is a type. This is a key notion in C++. When `class` means user-defined type in C++ why didn't I call it `type`? I chose `class` primarily because I dislike inventing new terminology and found Simula's quite adequate in most cases.

[2] The representation of objects of the user-defined type is part of the class declaration. This has far-reaching implications. For example, it means that true local variables can be implemented without the use of free store (heap store, dynamic store) or garbage collection. It also means that a function must be recompiled the representation of an object it uses directly is changed. See §4.3 for C++ facilities for expressing interfaces that avoid such recompilation.

[3] Compile time access control is used to restrict access to the representation. By default, only the functions mentioned in the class declaration can use names of class members. Members (usually function members) specified in the public interface – the declarations after the `public:` label – can be used by other code.

[4] The full type (including both the return type and the argument types) of a function is specified for function members. Static (compile-time) type checking is based on this type specification. This differed from C at the time, where function argument types were neither specified in interfaces nor checked in calls.

[5] Function definitions are typically specified ''elsewhere'' to make a class more like an interface specification than a lexical mechanism for organizing source code. This implies that separate compilation for class member functions and their users is easy and the linker technology traditionally used for C is sufficient to support C++.

[6] The function `new()` is a constructor, a function with a special meaning to the compiler. Such functions provided guarantees about classes. In this case, the guarantee is that the constructor – known somewhat confusingly as a `new`-function at the time – is guaranteed to be called to initialize every object of its class before the first use of the object.

[7] Both pointers and non-pointer types are provided (as in both C and Simula).

Much of the further development of C with Classes and C++ can be seen as exploring the consequences of these design choices, exploiting their good sides, and compensating for the problems caused by their bad sides. Many, but by no means all, of the implications of these design choices were understood at the time; [Stroustrup,1980a] is dated April 3, 1980. This section tries to explain what was understood at the time and give pointers to sections explaining later consequences and realizations.

2.4.1 Run-time Efficiency

In Simula, it is not possible to have local or global variables of class types; that is, every object of a class must be allocated on the free store using the `new` operator. Measurements of my Cambridge simulator had convinced me that this was a major source of inefficiency. Later, Karel Babcisky from the Norwegian Computer Centre presented data on Simula run-time performance that confirmed my conjecture [Babcisky,1984]. For that reason alone, I wanted global and local

variables of class types.

In addition, having different rules for the creation and scope of built-in and user-defined types is inelegant and I felt that on occasion my programming style had been cramped by absence of local and global class variables in Simula. Similarly, I had on occasion missed the ability to have pointers to built-in types in Simula so I wanted the C notion of pointers to apply uniformly over user-defined and built-in types. This is the origin of the notion that over the years grew into a ''principle'' for C++: User-defined and built-in types should behave the same relative to the language rules and receive the same degree of support from the language and its associated tools. When the ideal was formulated built-in types received by far the best support, but C++ has overshot that target so that built-in types now receive slightly inferior support compared to user-defined types.

The initial version of C with Classes did not provide inline functions to take further advantage of the availability of the representation. Inline functions were soon provided, though. The general reason for the introduction of inline functions was worry that the cost of crossing a protection barrier would cause people to refrain from using classes to hide representation. In particular, [Stroustrup,1982] observes that people had made data members public to avoid the function call overhead incurred by a constructor for simple classes where only one or two assignments are needed for initialization. The immediate cause for the inclusion of inline functions into C with Classes was a project that couldn't afford function call overhead for some classes involved in real-time processing.

Over the years considerations along these lines grew into the C++ ''principle'' that it was not sufficient to provide a feature, it had to be provided in an affordable form. Most definitely, ''affordable'' was seen as meaning ''affordable on hardware common among developers'' as opposed to ''affordable to researchers with high-end equipment'' or ''affordable in a couple of years when hardware will be cheaper.'' C with Classes was always considered as something to be used *now* or *next month* rather than simply a research project to deliver something for a couple of years hence.

Inlining was considered important for the utility of classes and therefore the issue was more *how* to provide it than *whether* to provide it. Two arguments won the day for the notion of having the programmer select which functions the compiler should try to inline. Firstly, I had poor experiences with languages that left the job of inlining to compilers ''because clearly the compiler knows best.'' The compiler only knows best if it has been programmed to inline and it has a notion of time/space optimization that agrees with mine. My experience with other languages was that only ''the next release'' would actually inline and it would do so according to an internal logic that a programmer couldn't effectively control. To make matters worse C (and therefore C with Classes and later C++) has genuine separate compilation so that a compiler never has access to more than a small part of the program (§2.4.2). Inlining a function for which you don't know the source appears feasible given advanced linker and optimizer technology, but such technology wasn't available at the time (and still isn't in most environments). Furthermore, extensive global analysis and optimization easily become unaffordable for large systems – where optimizations are most critical. C with Classes was designed to deliver efficient code given a simple portable implementation on traditional systems. Given that, the programmer had to help. Even today, the choice seems right.

2.4.2 The Linkage Model

The issue of how separately compiled programs are linked together is critical for any programming language and to some extent determines the features the language can provide. One of the critical influences on the development of C with Classes and C++ was the decision that

[1] Separate compilation should be possible with traditional C/Fortran UNIX/DOS style linkers.

[2] Linkage should in principle be type safe.
[3] Linkage should not require any form of database (though one could be used to improve a given implementation).
[4] Linkage to program fragments written in other languages such as C, assembler, and Fortran should be easy and efficient.

C uses ‘‘header files’’ to ensure consistent separate compilation. Declarations of data structure layouts, functions, variables, and constants are placed in header files that are typically textually included into every source file that needs the declarations. Consistency is ensured by placing adequate information in the header files and ensuring that the header files are consistently included. C++ follows this model up to a point.

The reason that layout information can be present in a C++ class declaration (though it doesn't *have* to be; see §4.3) is to ensure that the declaration and use of true local variables is easy and efficient. For example:

```
void f()
{
        stack s;
        int c;
        s.push('h');
        c = s.pop();
}
```

Using the `stack` declaration from §2.4, even a simple-minded C with Classes implementation can ensure that no use is made of free store for this example, that the call of `pop()` is inlined so that no function call overhead is incurred and that the non-inlined call of `push()` can invoke a separately compiled function `pop()`. In this, C++ resembles Ada [Ichbiah,1979].

At the time, I felt that there was a tradeoff between having separate interface and implementation declarations (as in Modula2) plus a tool (linker) for matching them up, and having a single class declaration plus a tool (a dependency analyzer) that considered the interface part separately from the implementation details for the purposes of re-compilation. It appears that I underestimated the complexity of the latter and also that the proponents of the former approach underestimate the cost (in terms of porting problems and run time overheads) of the former.

The concern for simple-minded implementations was partly a necessity caused by the lack of resources for developing C with Classes and partly a distrust of languages and mechanisms that required ‘‘clever’’ techniques. An early formulation of a design goal was that C with Classes ‘‘should be implementable without using an algorithm more complicated than a linear search.’’ Wherever that rule of thumb was violated – as in the case of function overloading (§3.3.3) – it led to semantics that were more complicated than anyone felt comfortable with and typically also to implementation complications.

The aim – based on my Simula experience – was to design a language that would be easy enough to understand to attract users and easy enough to implement to attract implementers. Only if a relatively simple implementation could be used by a relatively novice user in a relatively unsupportive programming environment to deliver code that compared favorably with C code in development time, correctness, run-time speed, and code size could C with Classes and later C++ expect to survive in competition with C.

This was part of a philosophy of fostering self-sufficiency among users. The aim was always – and explicitly – to develop local expertise in all aspects of using C++. Most organizations must follow the exact opposite strategy. They keep users dependent on services that generates revenue for a central support organization and/or consultants. In my opinion, this contrast is a deep reason for some of the differences between C++ and many other languages.

The decision to work in the relatively primitive – and almost universally available – framework of the C linking facilities caused the fundamental problem that a C++ compiler must always

work with only partial information about a program. An assumption made about a program could possibly be violated by a program written tomorrow in some other language (such as C, Fortran, or assembler) and linked in – possibly after the program has started executing. This problem surfaces in many contexts. It is hard for an implementation to guarantee

[1] that something is unique,
[2] that (type) information is consistent,
[3] that something is initialized.

In addition, C provides only the feeblest support for the notion of separate name spaces so that avoiding name space pollution by separately written program segments becomes a problem. Over the years, C++ has tried to face all of these challenges without departing from the fundamental model and technology that gives portability, but in the C with Classes days we simply relied on the C technique of header files.

Through the acceptance of the C linker came another ''principle'' for the development of C++: C++ is just one language in a system and not a complete system. In other words, C++ accepts the role of a traditional programming language with a fundamental distinction between the language, the operating system, and other important parts of the programmer's world. This delimits the role of the language in a way that is hard to do for a language, such as Smalltalk or Lisp, that is conceived as a complete system or environment. It makes it essential that a C++ program fragment can call program fragments written in other languages and that a C++ program fragment can itself be called by program fragments written in other languages. Being ''just a language'' also allows C++ implementations to benefit directly from tools written for other languages.

The need for a programming language and the code written in it to be just a cog in a much larger machine is of utmost importance to most industrial users yet such co-existence with other languages and systems was apparently not a major concern to most theoreticians, would-be perfectionists, and academic users. I believe this to be one of the main reasons for C++'s success.

2.4.3 Static Type Checking

I have no recollection of discussions, no design notes, and no recollection of any implementation problems about the introduction of static (''strong'') type checking into C with Classes. The C with Classes syntax and rules, the ones subsequently adopted for the ANSI C standard, simply appeared fully formed in the first C with Classes implementation. After that, a minor series of experiments led to the current (stricter) C++ rules. Static type checking was to me, after my experience with Simula and Algol68, a simple *must* and the only question was exactly how it was to be added.

To avoid breaking C code, it was decided to allow the call of an undeclared function and not perform type checking on such undeclared functions. This was of course a major hole in the type system and several attempts were made to decrease its importance as the major source of programming errors before finally – in C++ – the hole was closed by making a call of an undeclared function illegal. One simple observation defeated all attempts to compromise, and thus maintain a greater degree of C compatibility: As programmers learned C with Classes they lost the ability to find run-time errors caused by simple type errors. Having come to rely on the type checking and type conversion provided by C with Classes or C++, they lost the ability to quickly find the ''silly errors'' that creep into C programs through the lack of checking. Further, they failed to take the precautions against such silly errors that good C programmers take as a matter of course. After all, ''such errors don't happen in C with Classes.'' Thus, as the frequency of run-time errors caused by uncaught argument type errors goes down their seriousness and the time needed to find them goes up. The result was seriously annoyed programmers demanding further tightening of the type system.

The most interesting experiment with ''incomplete static checking'' was the technique of allowing calls of undeclared functions, but noting the type of the arguments used so that a

consistency check could be done when further calls were seen. When Walter Bright many years later independently discovered this trick he named it ‘‘autoprototyping,’’ using the ANSI C term *prototype* for a function declaration. The experience was that autoprototyping caught many errors and initially increased a programmer’s confidence in the type system. However, since consistent errors and errors in a function called only once in a compilation were not caught, autoprototyping ultimately destroyed programmer confidence in the type checker and induced a sense of paranoia even worse than I have seen in C or BCPL programmers.

C with Classes introduced the notation `f(void)` for a function `f` that takes no arguments as a contrast to `f()` that in C declares a function that can take any number of arguments of any type without any type check. My users soon convinced me, however, that the `f(void)` notation wasn’t very elegant, and that having functions declared `f()` accept arguments wasn’t very intuitive. Consequently, the result of the experiment was to have `f()` mean a function `f` that takes no arguments, as any novice would expect. It took support from both Doug McIlroy and Dennis Ritchie for me to build up courage to make this break from C. Only after they used the word *abomination* about `f(void)` did I dare give `f()` the obvious meaning. However, to this day C’s type rules are much laxer than C++’s and any use of `f()` as a function declaration is incompatible between the two languages.

Another early attempt to tighten C with Classes’ type rules was to disallow ‘‘information destroying’’ implicit conversions. Like others, I had been badly bitten by implicit `long` to `int` and `int` to `char` conversions. I decided to try to ban all implicit conversions that were not value preserving; that is, to require an explicit conversion operator wherever a larger object was stored into a smaller. The experiment failed miserably. Every C program I looked at contained large numbers of assignments of `int`s to `char` variables. Naturally, since these were working programs, most of these assignments were perfectly safe. That is, either the value was small enough not to become truncated or the truncation was expected or at least harmless in that particular context. There was no willingness in the C with Classes community to make such a break from C. I’m still looking for ways to compensate for these problems.

2.4.4 Why C?

A common question at C with Classes presentations was ‘‘Why use C? Why didn’t you build on, say, Pascal?’’ One version of my answer can be found in [Stroustrup,1986b]:

> ‘‘C is clearly not the cleanest language ever designed nor the easiest to use so why do so many people use it?
>
> [1] C is *flexible*: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.
>
> [2] C is *efficient*: The semantics of C are ‘‘low level’’; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.
>
> [3] C is *available*: Given a computer, whether the tiniest micro or the largest super−computer, the chance is that there is an acceptable quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so that a programmer rarely needs to design a new system from scratch.
>
> [4] C is *portable*: A C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible.
>
> Compared with these ‘‘first order’’ advantages, the ‘‘second order’’ drawbacks like the

> curious C declarator syntax and the lack of safety of some language constructs become less important. Designing ''a better C'' implies compensating for the major problems involved in writing, debugging, and maintaining C programs *without compromising the advantages of C*. C++ preserves all these advantages and compatibility with C at the cost of abandoning claims to perfection and of some compiler and language complexity. However, designing a language ''from scratch'' does not ensure perfection and the C++ compilers compare favorably in run-time, have better error detection and reporting, and equal the C compilers in code quality.''

This formulation is more polished than I could have managed in the early C with Classes days, but it does capture the essence of what I considered important about C and that I did not want to lose in C with Classes. Pascal was considered a toy language [Kernighan,1981], so it seemed easier and safer to add type checking to C than to add the features considered necessary for systems programming to Pascal. At the time, I had a positive dread of making mistakes of the sort where the designer out of misguided paternalism or plain ignorance makes the language unusable for real work in important areas. The ten years that followed clearly showed that choosing C as a base left me in the mainstream of systems programming where I intended to be. The cost in language complexity has been considerable, but manageable.

At the time, I considered Modula-2, Ada, Smalltalk, Mesa, and Clu as alternatives to C and as sources for ideas for C++ [Stroustrup,1984b] so there was no shortage of inspiration. However, only C, Simula, Algol68, and in one case BCPL left noticeable traces in C++ as released in 1985. Simula gave classes, Algol68 operator overloading (§3.3.3), references (§3.3.4), and the ability to declare variables anywhere in a block (§3.3.1), and BCPL gave // comments (§3.3.1).

There were several reasons for avoiding major departures from C style. I saw the merging of C's strengths as a systems programming language with Simula's strengths for organizing programs as a significant challenge in itself. Adding significant features from other languages could easily lead to a ''shopping list'' language and destroy the integrity of the resulting language. To quote from [Stroustrup,1986b]:

> ''A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of concepts for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is ''close to the machine'', so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is ''close to the problem to be solved'' so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.''

Again this formulation is more polished than I could have managed during the early stages of the design of C with Classes, but the general idea was clear. Departures from the known and proven techniques of C and Simula would have to wait for further experience with C with Classes and C++ and further experiments. I firmly believe – and believed then – that language design is not just design from first principles but also an art that requires experience, experiments, and sound engineering tradeoffs. Adding a major feature or concept to a language should not be a leap of faith but a deliberate action based on experience and fitting into a framework of other features and ideas of how the resulting language can be used. The post-1985 evolution of C++ shows the influence of ideas from Ada, Clu, and ML.

2.4.5 Syntax Problems

Could I have ''fixed'' the most annoying deficiencies of the C syntax and semantics at some point before C++ was made generally available? Could I have done so without removing useful features (to C with Classes' users in their environments – as opposed to an ideal world) or introducing incompatibilities that were unacceptable to C programmers wanting to migrate to C with Classes? I think not. In some cases, I tried, but I backed out my changes after complaints from

outraged users. The part of the C syntax I disliked most was the the declaration syntax. Having both prefix and postfix declarator operators cause a fair amount of confusion. So does allowing the type specifier to be left out (meaning `int` by default)†.

My eventual rationale for leaving things as they were was that any new syntax would (temporarily at least) add complexity to a known mess. Also, even though the old style is a boon to teachers of trivia and to people wanting to ridicule C, it is not a significant problem for C programmers. In this case, I'm not sure if I did the right thing, though. The agony to me and other C++ implementers, documenters, and tool builders caused by the perversities of syntax has been significant. Users can – and do – of course insulate themselves from such problems by writing in a small and easily understood subset of the C/C++ declaration syntax.

A significant syntactic simplification for the benefit of users was introduced into C++ at the cost of some extra work to implementers and some C compatibility problems. In C, the name of a structure, a ''structure tag,'' must always be preceded by the keyword `struct`. For example

```
struct buffer a;   /* 'struct' is necessary in C */
```

In the context of C with Classes, this had annoyed me for some time because it made user-defined types second class citizens syntactically. Given my lack of success with other attempts to clean up the syntax, I was reluctant and only made the change – at the time where C with Classes was mutated into C++ – at the urging of Tom Cargill. The name of a `struct` or a `class` is now a type name and requires no special syntactic identification:

```
buffer a;       // C++
```

The resulting fights over C compatibility lasted for years (see also §3.4).

2.4.6 Derived Classes

The derived class concept is C++'s version of Simula's prefixed class notion and thus a sibling of Smalltalk's subclass concept. The names *derived* class and *base* class were chosen because I never could remember what was *sub* and what was *super* and observed that I was not the only one with this particular problem. It was also noted that many people found it counterintuitive that a subclass typically has *more* information than its superclass. In inventing the terms derived class and base class, I departed from my usual principle of not inventing new names where old ones exist. In my defense, I note that I have never observed any confusion about what is base and what is derived among C++ programmers and that the terms are trivially easy to learn even for people without a grounding in mathematics.

The C with Classes concept was provided without any form of run-time support. In particular, the Simula (and C++) concept of a virtual function was missing. The reason for this was that I – with reason I think – doubted my ability to teach people how to use them and even more my ability to convince people that a virtual function is as efficient in time and space as an ordinary function as typically used. Often, people with Simula and Smalltalk experience still don't quite believe that until they have had the C++ implementation explained to them in detail – and many still harbor irrational doubts after that.

Even without virtual functions, derived classes in C with Classes were useful for building new data structures out of old ones and for associating operations with the resulting types. In particular, as explained in [Stroustrup,1980] and [Stroustrup,1982], they allowed list classes to be defined and also task classes.

In the absence of virtual functions, a user could use objects of a derived class and treat base classes as implementation details (only). Alternatively, an explicit type field could be introduced in a base class and used together with explicit type casts. The former strategy was used for tasks

† In 1995, the C++ standards committee finally banned ''implicit `int`'' in declarations.

where the user only sees specific derived task classes and ‘‘the system’’ sees only the task base classes. The latter strategy was used for various application classes where, in effect, a base class was used to implement a variant record for a set of derived classes. Much of the effort in C with Classes and later C++ has been to ensure that programmers needn’t write such code. Most important in my thinking at the time and in my own code was the combination of base classes, explicit type conversions, and (occasionally) macros to provide generic container classes. Eventually, these techniques matured into C++’s template facility and the techniques for using templates together with base classes to express commonality among instantiated templates (§6.3).

2.4.7 The Protection Model

Before starting work on C with Classes, I worked with operating systems. The notions of protection from the Cambridge CAP computer and similar systems – rather than any work in programming languages – inspired the C++ protection mechanisms. The class is the unit of protection and the fundamental rule is that you cannot grant yourself access to a class; only the declarations placed in the class declaration (supposedly by its owner) can grant access. By default, all information is private.

Access is granted by declaring a function in the public part of a class declaration, or by specifying a function or a class as a `friend`. Initially, only classes could be friends, thus granting access to all member functions of the friend class, but later it was found convenient to be able to grant access (friendship) to individual functions. In particular, it was found useful to be able to grant access to global functions.

A friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another.

Even in the first version of C with Classes, the protection model applied to base classes as well as members. Thus a class could be either publicly or privately derived from another. The private/public distinction for base classes predates the debate on implementation inheritance vs interface inheritance by about 5 years [Snyder,1986] [Liskov,1987]. If you want to inherit an implementation only, you use private derivation in C++. Public derivation gives users of the derived class access to the interface provided by the base class. Private derivation leaves the base as an implementation detail; even the public members of the private base class are inaccessible except through the interface explicitly provided for the derived class.

To provide ‘‘semi-transparent scopes’’ a mechanism was provided to allow individual public names from a private base class to be made public [Stroustrup,1982].

2.4.8 Run-time Guarantees

The access control mechanisms described above simply prevent unauthorized access. A second kind of guarantee was provided by ‘‘special member functions,’’ such as constructors, that were recognized and implicitly invoked by the compiler. The idea was to allow the programmer to establish guarantees, sometimes called ‘‘invariants,’’ that other member function could rely on. Curiously enough, the initial implementation contained a feature that is not provided by C++ but is often requested. In C with Classes, it was possible to define a function that would implicitly be called before every call of every member function (except the constructor) and another that would be implicitly called before every return from every member function. They were called `call` and `return` functions. They were used to provide synchronization for the monitor class in the original task library [Stroustrup,1980b]:

```
class monitor : object {
        /* ... */
        call() { /* grab lock */ }
        return() { /* release lock */ }
};
```

These are similar in intent to the CLOS `:before` and `:after` methods. Call and return functions were removed from the language because nobody (but me) used them and because I seemed to have completely failed to convince people that `call()` and `return()` had important uses. In 1987 Mike Tiemann suggested an alternative solution called ''wrappers'' [Tiemann,1987], but at the USENIX implementors' workshop in Estes Park this idea was determined to have too many problems to be accepted into C++.

2.4.9 Features Considered, but not Provided

In the early days, many features were considered that later appeared in C++ or are still discussed. These included virtual functions, `static` members, templates, and multiple inheritance. However,

> ''All of these generalizations have their uses, but every ''feature'' of a language takes time and effort to design, implement, document, and learn.'' ... ''The base class concept is an engineering compromise, like the C class concept [Stroustrup,1982].''

I just wish I had explicitly mentioned the need for experience. With that, the case against featurism and for a pragmatic approach would have been complete.

The possibility of automatic garbage collection was considered on several occasions before 1985 and deemed unsuitable for a language already in use for real-time processing and hard-core systems tasks such as device drivers. In those days, garbage collectors were less sophisticated than they are today and the processing power and memory capacity of the average computer were small fractions of what today's systems offer. My personal experience with Simula and reports of other GC-based systems convinced me that GC was unaffordable by me and my colleagues for the kind of applications we were writing. Had C with Classes (or even C++) been defined to require automatic garbage collection it would have been more elegant, but stillborn.

Direct support for concurrency was also considered but rejected in favor of a library based approach (§2.2).

2.5 Work Environment

C with Classes was designed and implemented by me as a research project in the Computing Science Research Center of Bell Labs. This center provided – and still provides – a possibly unique environment for such work. When I joined I was basically told to ''do something interesting,'' given suitable computer resources, encouraged to talk to interesting and competent people, and given a year before having to formally present my work for evaluation.

There was a cultural bias against ''grand projects'' requiring many people, against ''grand plans'' like untested paper designs for others to implement, and against a class distinction between designers and implementers. If you liked such things, Bell Labs and others have many places where you could indulge such preferences. However, in the Computing Science Research Center it was almost a requirement that you – if you were not into theory – (personally) implemented something embodying your ideas and found users that could benefit from what you built. The environment was very supportive for such work and the Labs provided a large pool of people with ideas and problems to challenge and test anything built. Thus I could write in [Stroustrup,1986b]:

> ''There never was a C++ paper design; design, documentation, and implementation went on simultaneously. Naturally, the C++ front-end is written in C++. There never was a ''C++ project'' either, or a ''C++ design committee''. Throughout, C++ evolved, and continues to

evolve, to cope with problems encountered by users, and through discussions between the author and his friends and colleagues.''

Only after C++ was an established language did more conventional organizational structures emerge and even then I was officially in charge of the reference manual and had the final say over what went into it until that task was handed over to the ANSI C++ committee in early 1990. On the other hand, after the first few months I never had the freedom to design just for the sake of designing something beautiful or to make arbitrary changes in the language as it stood at any given time. Whatever I considered a language feature required an implementation to make it real, and any change or extension required the concurrence and usually enthusiasm of key C with Classes and later C++ users.

Since there was no guaranteed user population, the language and its implementations could only survive by serving the needs of its users well enough to counteract the organizational pull of established languages and the marketing hype of newer languages.

C with Classes grew through discussions with people in the Computing Science Research Center and early users there and elsewhere in the Labs. Most of C with Classes and later C++ was designed on somebody else's blackboard and the rest on mine. Most such ideas were rejected as being too elaborate, too limited in usefulness, too hard to implement, too hard to teach for use in real projects, not efficient enough in time or space, too incompatible with C, or simply too weird. The few ideas that made it through this filter – invariably involving discussions among at least two people – I then implemented. Typically, the idea mutated through the effort of implementation, testing, and early use by me and one or two others. The resulting version was tried on a larger audience and would often mutate a bit further before finding its way into the ''official'' version of C with Classes as shipped by me. Usually, a tutorial was written somewhere along the way. Writing a tutorial was considered an essential design tool, because if a feature cannot be explained simply the burden of supporting it will be too great. This point was never far from my mind because during the early years I *was* the support organization.

In the early days Sandy Fraser, my department head at the time, was very influential. For example, I believe he was the one to encourage me to break from the Simula style of class definition where the complete function definition is included and adopt the style where function definitions are typically elsewhere thus emphasizing the class declaration's role as an interface. Much of C with Classes was designed to allow simulators to be built that could be used in Sandy Fraser's work in network design. The first real application of C with Classes was such network simulators. Sudhir Agrawal was another early user who influenced the development of C with Classes through his work with network simulations. Jonathan Shopiro provided much feedback of the C with Classes design and implementation based on his simulation of a ''dataflow database machine.''

For more general discussions on programming language issues, as opposed to looking at applications to determine which problems needed to be solved, I turned to Dennis Ritchie, Steve Johnson, and in particular Doug McIlroy. Doug McIlroy's influence on the development of both C and C++ cannot be overestimated. I cannot remember a single critical design decision in C++ that I have not discussed at length with Doug. Naturally, we didn't always agree, but I still have a strong reluctance to make a decision that goes against Doug's opinion. He has a knack for being right and an apparently infinite amount of experience and patience.

Since the main design work for C with Classes and C++ was done on blackboards the thinking tended to focus on solutions to ''archetypical'' problems: Small examples that are considered characteristic for a large class of problems. Thus, a good solution to the small example will provide significant help in writing programs dealing with real problems of that class. Many of these problems have entered the C++ literature and folklore through my use of them as examples in my papers, books, and talks. For C with Classes, the example considered most critical was the `task` class that was the basis of the task-library supporting Simula-style simulation. Other key classes

were `queue`, `list`, and `histogram` classes. The `queue` and `list` classes were based on the idea – borrowed from Simula – of providing a `link` class from which users derived their own classes.

The danger inherent in this approach is to create a language and tools that provide elegant solutions to small selected examples yet don't scale to building complete systems or large programs. This was counteracted by the simple fact that C with Classes (and later C++) had to pay for itself during its early years. This ensured that C with Classes couldn't evolve into something that was elegant but useless.

3 From C with Classes to C++

During 1982 it became clear to me that C with Classes was a ''medium success'' and would remain so until it died. I defined a medium success as something so useful that it easily paid for itself and its developer, but not so attractive and useful that it would pay for a support and development organization. Thus, continuing with C with Classes and its C pre-processor implementation would condemn me to support C with Classes use indefinitely. I was convinced that there were only two ways out of this dilemma:

[1] Stop supporting C with Classes so that the users would have to go elsewhere (freeing me to do something else).

[2] Develop a new and better language based on my experience with C with Classes that would serve a large enough set of users to pay for a support and development organization (thus freeing me to do something else). At the time I estimated that 5,000 industrial users was the necessary minimum.

The third alternative, increasing the user population through marketing (hype), never occurred to me. What actually happened was that the explosive growth of C++, as the new language was eventually named, kept me so busy that to this day I haven't managed to get sufficiently detached to do something else of significance.

The success of C with Classes was, I think, a simple consequence of meeting its design aim: C with Classes did help organize a large class of programs significantly better than C without the loss of run-time efficiency and without requiring enough cultural changes to make its use infeasible in organizations that were unwilling to undergo major changes. The factors limiting its success were partly the limited set of new facilities offered over C and partly the pre-processor technology used to implement C with Classes. There simply wasn't enough support in C with Classes for people who *were* willing to invest significant efforts to reap matching benefits: C with Classes was an important step in the right direction, but only one small step. As a result of this analysis I began designing a cleaned-up and extended successor to C with Classes and implementing it using traditional compiler technology.

The resulting language was at first still called C with Classes but after a polite request from management it was given the name C84. The reason for the naming was that people had taken to calling C with Classes ''new C,'' and then C. This last abbreviation led to C being called ''plain C,'' ''straight C,'' and ''old C.'' The name C84 was used only for a few months, partly because it was ugly and institutional, partly because there would still be confusion if people dropped the '84.' I asked for ideas for a new name and picked C++ because it was short, had nice interpretations, and wasn't of the form ''adjective C.'' In C, `++` can, depending on context, be read as ''next,'' ''successor,'' or ''increment'' though it is always pronounced ''plus plus.'' The name C++ and its runner up ++C are fertile sources for jokes and puns – almost all of which were known and appreciated before the name was chosen. The name C++ was suggested by Rick Mascitti. It was first used in [Stroustrup,1984b] where it was edited into the final copy in December of 1983.

3.1 Aims

During the 1982–1984 period the aims for C++ gradually became more ambitious and more definite. I had come to see C++ as a language separate from C, and libraries and tools had emerged as areas of work. Because of that, because tools developers within Bell Labs were beginning to show interest in C++, and because I had embarked on a completely new implementation that would become the C++ compiler front-end, Cfront, I had to answer key questions:

[1] Who will the users be?
[2] What kind of systems will they use?
[3] How will I get out of the business of providing tools?
[4] How should the answers to [1], [2], and [3] affect the language definition?

My answer to [1], ''Who will the users be?,'' was that first my friends within Bell Labs and I would use it, then more widespread use within AT&T would provide more experience, then some universities would pick up the ideas and the tools, and finally AT&T and others would be able to make some money by selling the set of tools that had evolved. At some point, the initial and somewhat experimental implementation done by me would be faded out in favor of more ''industrial strength'' implementations by AT&T and others.

This made practical and economic sense; the initial (Cfront) implementation would be tool-poor, portable, and cheap because that was what I, my colleagues, and many university users needed and could afford. Later, there would be ample scope for AT&T and others to provide better tools for more specialized environments. Such better tools aimed primarily at industrial users needn't be cheap either, and would thus be able to pay for the support organizations necessary for large-scale use of the language. That was my answer to [3] ''How will I get out of the business of providing tools?'' Basically, the strategy worked. However, just about every detail actually happened in an unforeseen way.

To get an answer to [2] ''What kind of systems will they use?,'' I simply looked around to see what kind of systems the C with Classes users actually did use. They used everything from boxes that were so small that they couldn't run a compiler to mainframes. They used more operating systems than I had heard of. Consequently, I concluded that extreme portability and the ability to do cross compilation were necessities and that I could make no assumption about the size and speed of the machines running generated code. To build a compiler, however, I would have to make assumptions about the kind of system people would develop their programs on. I assumed that one MIPS plus one Mbyte would be available. That assumption, I considered a bit risky because most of my prospective users at the time had at most part of a PDP11 or some other relatively low-powered and/or timeshared system available.

I did not predict the PC revolution, but by over-shooting my performance target for Cfront I happened to build a compiler that (barely) could run on an IBM PC/AT, thus providing an existence proof that C++ could be an effective language on a PC and thereby spurring commercial software developers to beat it.

As the answer to [4] ''How does all this affect the language definition?'' I concluded that no feature must require really sophisticated compiler or run-time support, that available linkers must be used, and that the code generated would have to be efficient (comparable to C) even initially.

3.2 Cfront

The Cfront compiler front-end for the C84 language was designed and implemented by me between the spring of 1982 and the summer of 1983. The first user outside the computer science research center, Jim Coplien, received his copy in July of 1983. Jim was in a group that had been doing experimental switching work with C with Classes in Bell Labs in Naperville, Illinois for some time.

In that same time period I designed C84, drafted the reference manual published January 1, 1984 [Stroustrup,1984a], designed the `complex` number library and implemented it together

with Leonie Rose [Rose,1984], designed and implemented the first `string` class together with Jonathan Shopiro, maintained and ported the C with Classes implementation, supported the C with Classes users and helped them become C84 users. That was a busy year and a half.

Cfront was (and is) a traditional compiler front-end performing a complete check of the syntax and semantics of the language, building an internal representation of its input, analyzing and rearranging that representation, and finally producing output suitable for some code generator. The internal representation was (is) a graph with one symbol table per scope. The general strategy is to read a source file one global declaration at a time and produce output only when a complete global declaration has been completely analyzed.

The organization of Cfront is fairly traditional except maybe for the use of many symbol tables instead of just one. Cfront was originally written in C with Classes (what else?) and soon transcribed into C84 so that the very first working C++ compiler was done in C++. Even the first version of Cfront used classes heavily, but no virtual functions because they were not available at the start of the project.

The most unusual – for its time – aspect of Cfront was that it generated C code. This has caused no end of confusion. Cfront generated C because I needed extreme portability for an initial implementation and I considered C the most portable assembler around. I could easily have generated some internal back-end format or assembler from Cfront, but that was not what my users needed. No assembler or compiler back-end served more than maybe a quarter of my user community and there was no way that I could produce the, say, six backends needed to serve just 90% of that community. In response to this need, I concluded that using C as a common input format to a large number of code generators was the only reasonable choice. The strategy of building a compiler as a C generator has later become quite popular so that languages such as Ada, CLOS, Eiffel, Modula-3, and Smalltalk have been implemented that way. I got a high degree of portability at a modest cost in compile time overhead. Over the years I have measured this overhead on various systems and found it to be between 25% and 100% of the ''necessary'' parts of a compilation.

Please note that the C compiler is used as a code generator *only*. Any error message from the C compiler reflects an error in the C compiler or in Cfront, but not in the C++ source text. Every syntactic and semantic error is in principle caught by Cfront, the C++ compiler front-end. I stress this because there has been a long history of confusion about what Cfront was/is. It has been called a preprocessor because it generates C, and for people in the C community (and elsewhere) that has been taken as proof that Cfront was a rather simple program – something like a macro preprocessor. People have thus ''deduced'' (wrongly) that a line-for-line translation from C++ to C is possible, that symbolic debugging at the C++ level is impossible when Cfront is used, that code generated by Cfront must be inferior to code generated by ''real compilers,'' that C++ wasn't a ''real language,'' etc. Naturally, I have found such unfounded claims most annoying – especially when they were leveled as criticisms of the C++ language. There are now several C++ compilers that use Cfront together with local code generators without going through a C front end. To the user, the only obvious difference is faster compile times.

Cfront is only a compiler front-end and can never be used for real programming by itself. It needs a driver to run the source file through the C preprocessor, Cpp, then run the output of Cpp through Cfront and the output from Cfront through a C compiler:

In addition, the driver must ensure that dynamic (run-time) initialization is done. In Cfront 3.0, the driver becomes yet more elaborate as automatic template instantiation (§6.3) is implemented [McClusky,1992].

As mentioned, I decided to live within the constraints of traditional linkers. However, there was one constraint that I felt was too difficult to live with, yet so silly that I had a chance of fighting it if I had sufficient patience: Most traditional linkers had a very low limit on the number of characters that can be used in external names. A limit of 8 characters was common, and 6 characters and one case only are guaranteed to work as external names in Classical C; ANSI/ISO C accepts that limit also. Given that the name of a member function includes the name of its class and that the type of an overloaded function has to be reflected in the linkage process somehow or other, I had little choice. Consequently I started (in 1982) lobbying for longer names in linkers. I don't know if my efforts actually had any effect, but these days most linkers do give me the much larger number of characters I need. Cfront uses encodings to implement type-safe linkage in a way that makes a limit of 32 characters too low for comfort and even 256 is a bit tight at times (see §3.3.3). In the interim, systems of hash coding of long identifiers have been used with archaic linkers, but that was never completely satisfactory.

Versions of C++ are often named by Cfront release numbers. Release 1.0 was the language as defined in ''*The C++ Programming Language*'' [Stroustrup,1986b].

Releases 1.1 (June 1986) and 1.2 (February 1987) were primarily bug fix releases but also added pointers to members and protected members (§4.1). Release 2.0 was a major cleanup that also introduced multiple inheritance (§4.2) in June 1989. Release 2.1 (April 1990) was primarily a bug fix release that brought Cfront (almost) into line with the definition in the ARM, Ellis & Stroustrup: *The Annotated C++ Reference Manual* [Ellis,1990] (§6.1). Release 3.0 (September 1991) added templates (§6.3) as specified in the ARM. Release 4.0 (due in 1993) is expected to add exception handling (§6.4) as specified in the ARM.

I wrote the first versions of Cfront (1.0, 1.1, 1.2) and maintained them; Steve Dewhurst worked on it with me for a few months before release 1.0 in 1985. Laura Eaves did much of the work on the Cfront parser for release 1.0, 1.1, 2.1, and 3.0. I also did the lion's share of the programming for releases 1.2 and 2.0, but starting with release 1.2, Stan Lippman also spent most of his time on Cfront. George Logothetis, Judy Ward, and Nancy Wilkinson, and Stan Lippman did most of the work for releases 2.1 and 3.0. The work on 2.0 was coordinated by Barbara Moo, and Andrew Koenig organized Cfront testing. Barbara also coordinated releases 1.2, 2.1, and 3.0. Sam Haradhvala from Object Design Inc. did an initial implementation of templates in 1989 that Stan Lippman extended and completed for release 3.0 in 1991. The initial implementation of exception handling in Cfront was done by Hewlett-Packard in 1992. In addition to these people

who have produced code that has found its way into the main version of Cfront, many people have built local C++ compilers from it. Apple, Centerline (formerly Saber), ParcPlace, Sun, HP, and others ship products that contain locally modified versions of Cfront.

3.3 Language Feature Details

The major additions to C with Classes introduced to produce C++ were:

[1] Virtual functions.
[2] Function name and operator overloading.
[3] References.
[4] Constants (`const`).
[5] User-controlled free-store memory control.
[6] Improved type checking

In addition, the notion of call and return functions (§2.4.8) was dropped due to lack of use and many minor details were changed to produce a cleaner language.

3.3.1 Minor Changes

The most visible minor change was the introduction of BCPL-style comments:

```
int a; /* C-style explicitly terminated comment */
int b; // BCPL-style comment terminated by end-of-line
```

Since both styles of comments are allowed people can simply use the style they like best.

The name ''`new`-function'' for constructors had been a source of confusion so the name constructor was introduced.

In C with Classes, a dot was used to express membership of a class as well as expressing selection of a member of a particular object. This had been the cause of some minor confusion and could also be used to construct ambiguous examples. To alleviate this, `::` was introduced to mean membership of class and `.` was retained exclusively for membership of object.

I borrowed the Algol68 notion that a declaration can be introduced wherever it is needed (and not just at the top of some block). Thus, I enabled a ''initialize-only'' or ''single-assignment'' style of programming that is less error-prone than traditional styles. This style is essential for references and constants that cannot be assigned and inherently more efficient for types where default initialization is expensive.

3.3.2 Virtual Functions

The most obvious new feature in C++ – and certainly the one that had the greatest impact on the style of programming one could use for the language – was virtual functions. The idea was borrowed from Simula and presented in a form that was intended to make a simple and efficient implementation easy. The rationale for virtual functions was presented in [Stroustrup,1986b] and [Stroustrup,1986c]. To emphasize the central role of virtual functions in C++ programming I will quote it in detail here [Stroustrup,1986c]:

> ''An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type `shape` for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:
>
> ```
> class point{ /* ... */ };
> class color{ /* ... */ };
> ```
>
> You might define a shape like this:

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where()       { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};
```

The ‘‘type field’’ `k` is necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag `k`). The function `draw()` might be defined like this:

```
void shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
    }
}
```

This is a mess. Functions such as `draw()` must ‘‘know about’’ all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves ‘‘touching’’ the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type `shape`.

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and used supports object-oriented programming. Other languages don’t.

The Simula inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked ‘‘virtual’’ (the Simula and C++ term for ‘‘may be re-defined later in a class derived from this one’’). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all(shape** v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {}    // yes, the null function
};
```

In C++, class `circle` is said to be *derived* from class `shape`, and class `shape` is said to be a *base* of class `circle`. An alternative terminology calls `circle` and `shape` subclass and superclass, respectively.’’

For further discussion of virtual functions and object-oriented programming see §3.7 and §4.3.

The key implementation idea was that the set of virtual functions in a class defines a array of pointers to functions so that a call of a virtual function is simply an indirect function call through that array. There is one array per class and one pointer to such an array in each object of a class that has virtual functions.

I don’t remember much interest in virtual functions at the time. Probably I didn’t explain the concepts involved well, but the main reaction I received from people in my immediate vicinity was one of indifference and skepticism. A common opinion was that virtual functions were simply a kind of crippled pointer to function and thus redundant. Worse, it was sometimes argued that a well-designed program wouldn’t need the extensibility and openness provided by virtual functions so that proper analysis would show which non-virtual functions could be called directly. Therefore, the argument went, virtual functions were simply a form of inefficiency. Clearly I disagreed, and added virtual functions anyway.

3.3.3 Overloading

Several people had asked for the ability to overload operators. Operator overloading ‘‘looked neat’’ and I knew from experience with Algol68 how the idea could be made to work. However, I was reluctant to introduce the notion of overloading into C++:

[1] Overloading was reputed to be hard to implement so that compilers would grow to

monstrous size.

[2] Overloading was reputed to be hard to teach and hard to define precisely so that manuals and tutorials would grow to monstrous size.

[3] Code written using operator overloading was reputed to be inherently inefficient.

[4] Overloading was reputed to make code incomprehensible.

If [3] or [4] were true then C++ would be better off without overloading. If [1] or [2] were true then I didn't have the resources to provide overloading.

However, if all of these conjectures were false then overloading would solve some real problems for C++ users. There were people who would like to have complex numbers, matrices, and APL-like vectors in C++. There were people who would like range-checked arrays, multi-dimensional arrays and strings in C++. There were at least two separate applications for which people wanted to overload logical operators such as `|` (or), `&` (and), and `^` (exclusive or). The way I saw it, the list was long and would grow with the size and the diversity of the C++ user population. My answer to [4] ''overloading makes code obscure'' was that several of my friends whose opinion I valued and whose experience was measured in decades claimed that their code would become cleaner if they had overloading. So what if one can write obscure code with overloading? It is possible to write obscure code in any language. It matters more how a feature can be used well than how it can be misused.

Next, I convinced myself that overloading wasn't inherently inefficient [Stroustrup,1984c] [Ellis,1990]. The details of the overloading mechanism were mostly worked out on my blackboard and those of Stu Feldman, Doug McIlroy, and Jonathan Shopiro.

Thus, having worked out an answer to [3], ''code written using overloading is efficient,'' I needed to concern myself with [1] and [2], the issue of compiler and language complexity. I first observed that use of classes with overloaded operators, such as `complex` and `string`, was quite easy and didn't put a major burden on the programmer. Next I wrote the manual sections to prove that the added complexity wasn't a serious issue; the manual needed less than a page and a half extra (out of a 42-page manual). Finally, I did the first implementation in two hours using only 18 lines of extra code in Cfront, and I felt I had demonstrated that the fears about definition and implementation complexity were somewhat exaggerated.

Naturally, all these issues were not really tackled in this strict sequential order. However, the emphasis of the work did start with utility issues and slowly drifted to implementation issues. The overloading mechanisms were described in detail in [Stroustrup,1984c] and examples of classes using the mechanisms were written up [Rose,1984] [Shopiro,1985].

In retrospect, I underestimated the complexity of the definition and implementation issues and compounded these problems by trying to isolate overloading mechanisms from the rest of the language semantics. The latter was done out of misguided fear of confusing users. In particular, I required that a declaration

```
overload print;
```

should precede declarations of an overloaded function `print`, such as

```
void print(int);
void print(const char*);
```

I also insisted that ambiguity control should happen in two stages so that resolutions involving built-in operators and conversions would always take precedence over resolutions involving user-defined operations. Maybe this latter was inevitable given the concern for C compatibility and the chaotic nature of the C conversion rules for built-in types. These conversions do *not* constitute a lattice; for example, implicit conversions are allowed both from `int` to `float` and from `float` to `int`. However, the rules for ambiguity resolution were too complicated, caused surprises, and had to be revised for release 2.0. I still consider these rules too complex, but do not see scope for more than minor adjustments.

Requiring explicit `overload` declarations was plain wrong and the requirement was dropped in release 2.0.

3.3.4 References

References were introduced primarily to support operator overloading. C passes every function argument by value, and where passing an object by value would be inefficient or inappropriate the user can pass a pointer. This strategy doesn't work where operator overloading is used. In that case, notational convenience is essential so that a user cannot be expected to insert address-of operators if the objects are large.

Problems with debugging Algol68 convinced me that having references that didn't change what object they referred to after initialization was a good thing. If you wanted to do more complicated pointer manipulation in C++ you can use pointers. Because C++ has both pointers and references it does not need operations for distinguishing operations on the reference itself from operations on the object referred to (like Simula) or the kind of deductive mechanism employed by Algol68.

It is important that `const` references can be initialized by non-lvalues and lvalues of types that require conversion. In particular, this is what allows a Fortran function to be called with a constant:

```
extern "Fortran" float sqrt(const float&); // '&' means reference

sqrt(2); // call by reference
```

Jonathan Shopiro was deeply involved in the discussions that led to the introduction of references. In addition to the obvious uses of references, such as argument, we considered the ability to use references as return types important. This allowed us to have a very simple index operator for a string class:

```
class String {
  // ...
  char& operator[](int index);  // subscript operator
                                // return a reference
};

void f(String& s)
{
  char c1 = ...
  s[i] = c1;      // assign to operator[]'s result
  // ...
  char c2 = s[i]; // assign operator[]'s result
}
```

We considered allowing separate functions for left-hand and right-hand side use of a function but considered using references the simpler alternative even though this implies that we need to introduce additional ''helper classes'' to solve some problems where returning a simple reference isn't enough.

3.3.5 Constants (`const`)

In operating systems, it is common to have access to some piece of memory controlled directly or indirectly by two bits: one that indicates whether a user can write to it and one that indicates whether a user can read it. This idea seemed to me directly applicable to C++ and I considered allowing every type to be specified `readonly` or `writeonly` [Stroustrup,1981b]. The proposal is focused on specifying interfaces rather than on providing symbolic constants for C. Clearly, a `readonly` value is a symbolic constant, but the scope of the proposal is far greater.

Initially, I proposed pointers to `readonly` but not `readonly` pointers. A brief discussion with Dennis Ritchie evolved the idea into the `readonly/writeonly` mechanism that I implemented and proposed to an internal Bell Labs C standards group chaired by Larry Rosler. There, I had my first experience with standards work. I came away from a meeting with an agreement (that is, a vote) that `readonly` would be introduced into C – yes C, not C with Classes or C++ – provided it was renamed `const`. Unfortunately, a vote isn't executable so nothing happened to our C compilers. A while later, the ANSI C committee (X3J11) was formed and the `const` proposal resurfaced there and became part of ANSI/ISO C.

However, in the meantime I had experimented further with `const` in C with Classes and found that `const` was a useful alternative to macros for representing constants only if a global `consts` were implicitly local to their compilation unit. Only in that case could the compiler easily deduce that their value really didn't change and allow simple `consts` in constant evaluations and thus avoid allocating space for such constants and use them in constant expressions. C did not adopt this rule. This makes `consts` far less useful in C than in C++ and leaves C dependent on the preprocessor where C++ programmers can use properly typed and scoped `consts`.

3.3.6 Memory Management

Long before the first C with Classes program was written, I knew that free store (dynamic memory) would be used more heavily in a language with classes than in traditional C programs. This was the reason for the introduction of the `new` and `delete` operators in C with Classes. The `new` operator that both allocates memory from the free store and invokes a constructor to ensure initialization was borrowed from Simula. The `delete` operator was a necessary complement because I did not want C with Classes to depend on a garbage collector. The argument for the `new` operator can be summarized like this. Would you rather write:

```
X* p = new X(2);
```

or

```
struct X * p = (struct X *) malloc(sizeof(struct X));
if (p == 0) error("memory exhausted");
p->init(2);
```

and which version are you most likely to make a mistake in? The arguments against – which were voiced quite a lot at the time – were ''but we don't *really* need it,'' and ''but someone will have used `new` as an identifier.'' Both observations are correct, of course.

Introducing operator `new` thus made the use of free store more convenient and less error–prone. This increased its use even further so that the C free store allocation routine `malloc()` used to implement `new` became the most common performance bottleneck in real systems. This was no real surprise either; the only problem was what to do about it. Having real programs spend 50% or more of their time in `malloc()` wasn't acceptable.

I found per–class allocators and deallocators very effective. The fundamental idea is that free store memory usage is dominated by the allocation and deallocation of lots of small objects from very few classes. Take over the allocation of those objects in a separate allocator and you can save both time and space for those objects and also reduce the amount of fragmentation of the general free store. The mechanism provided for 1.0, ''assignment to `this`,'' was too low level and error–prone and was replaced by a cleaner solution in 2.0 (§4.1).

Note that static and automatic (stack allocated) objects were always possible and that the most effective memory management techniques relied heavily on such objects. The string class was a typical example, here `String` objects are typically on the stack so that they require no explicit memory management and the free store they rely on is managed exclusively and invisibly to the user by the String member functions.

3.3.7 Type Checking

The C++ type checking rules were the result of experiments with the C with Classes. All function calls are checked at compile time. The checking of trailing arguments can be suppressed by explicit specification in a function declaration. This is essential to allow C's `printf()`:

```
int printf(const char* ...); // accept any argument after
                             // the initial character string

// ...

printf("date: %s %d 19%d\n",month,day,year); // maybe right
```

Several mechanisms were provided to alleviate the withdrawal symptoms that many C programmers feel when they first experience strict checking. Overriding type checking using the ellipsis was the most drastic and least recommended of those. Function name overloading (§3.3.3) and default arguments [Stroustrup,1986b] made it possible to give the appearance of a single function taking a variety of argument lists without compromising type safety. The stream I/O system demonstrates that the weak checking wasn't necessary even for I/O (see §5.3.1).

3.4 Relationship to Classic C

With the introduction of a separate name, C++, and the writing of a C++ reference manual [Stroustrup,1984a] compatibility with C became an issue of major importance and a point of controversy.

Also, in late 1983 the branch of Bell Labs that developed and supported UNIX and produced AT&T's 3B series of computers became interested in C++ to the point where they were willing to put resources into the development of C++ tools. Such development was necessary for the evolution of C++ from a one man show to a language that a corporation could base critical projects on. Unfortunately, it also implied that development management needed to consider C++.

The first demand to emerge from development management was that of 100% compatibility with C. The ideal of C compatibility is quite obvious and reasonable, but the reality of programming isn't that simple. For starters, with which C should C++ be compatible? C dialects abounded, and though ANSI C was emerging, it was still years from having a stable definition and its definition allowed many dialects. Naturally, the average user who wanted C compatibility insisted that C++ should be compatible with the local C dialect. This was an important practical problem and a great concern to me and my friends. It seemed far less of a concern to business−oriented managers and salesmen who either didn't quite understand the technical details or would like to use C++ to tie users into their software and/or hardware.

Another side of the compatibility issue was more critical: ''In which ways must C++ differ from C to meet its fundamental goals?'' and also ''In which ways must C++ be compatible with C to meet its fundamental goals?'' Both sides of the issue are important and revisions were made in both directions during the transition from C with Classes to C++ as shipped as release 1.0. Slowly and painfully an agreement emerged that there would be no gratuitous incompatibilities between C++ and ANSI C (when it became a standard) [Stroustrup,1986b] but that there was such a thing as an incompatibility that was not gratuitous. Naturally, the concept of ''gratuitous incompatibilities'' was a topic of much debate and took up a disproportional part of my time and effort. This principle has lately been known as ''C++: As close to C as possible – but no closer.'' after the title of a paper by Andrew Koenig and me [Koenig,1989].

Some conclusions about modularity and how a program is composed out of separately compiled parts were explicitly reflected in the original C++ reference manual [Stroustrup,1984a]:

[a] Names are private unless they are explicitly declared public.

[b] Names are local to their file unless explicitly exported from it.

[c] Static type is checked unless explicitly suppressed.

[d] A class is a scope (implying that classes nest properly).

Point [a] doesn't affect C compatibility but [b], [c], [d] imply incompatibilities:

[1] The name of a non-local C function or object is by default accessible from other compilation units,

[2] C functions need not be declared before use and calls are by default not type checked, and

[3] C structure names don't nest (even when they are lexically nested).

In addition,

[4] C++ has a single name space whereas C had a separate name space for ''structure tags'' (§2.4.5).

The ''compatibility wars'' now seem petty and boring, but some of the underlying issues are still unresolved and we are still struggling with them in the ANSI/ISO committee. I strongly suspect that the reason the compatibility wars were drawn out and curiously inconclusive was that we never quite faced the deeper issues related to the differing goals of C and C++ and saw compatibility as a set of separate issues to be resolved individually.

Typically, the least fundamental issue [4] ''name spaces'' took up the most effort, but was eventually resolved [Ellis,1990].

I had to compromise the notion of a class as a scope [3] and accept the C ''solution'' to be allowed to ship release 1.0. One practical problem was that I had never realized that a C struct didn't constitute a scope so that examples like this

```
struct outer {
        struct inner {
                int i;
        };
        int j;
};

struct inner a = { 1 };
```

are legal C. When the issue came up towards the end of the compatibility wars I didn't have time to fathom the implications of the C ''solution'' and it was much easier to agree than to fight the issue. Later, after many technical problems and much discontent from users, nested class scopes were re-introduced into C++ in 1989 [Ellis,1990].

After much hassle, C++'s stronger type checking of function calls was accepted (unmodified). An implicit violation of the static type system is the original example of a C/C++ incompatibility that is not gratuitous. As it happens, the ANSI C committee adopted a slightly weaker version of C++'s rules and notation on this point and declared uses that don't conform to the C++ rules obsolete.

On issue [1], I had to accept the C rule that global names are by default accessible from other compilation units. There simply wasn't any support for the more restrictive C++ rule. This meant that C++, like C, lacks an effective mechanism for expressing modularity above the level of the class and the file. This has led to a series of complaints and the ANSI/ISO committee is now looking into several proposals for mechanisms to avoid name space pollution†. However, people – such as Doug McIlroy – who argued that C programmers would not accept a language where every object and function meant to be accessible from another compilation unit had to be explicitly declared as such were probably right at the time and saved me from making a serious mistake. I am now convinced that the original C++ solution wasn't elegant enough anyway.

† In 1994, the C++ standard committee accepted a proposal for a namespace mechanism; see [Stroustrup,1994].

3.5 Tools for Language Design

Theory and tools more advanced than a blackboard have not been given much space in the description of the history of C++. I tried to use YACC (an LALR(1) parser generator) for the grammar work, and was defeated by C's syntax (§2.4.5). I looked at denotational semantics, but was again defeated by quirks in C. Ravi Sethi had looked into that problem and found that he couldn't express the C semantics that way [Sethi,1980]. The main problem was the irregularity of C and the number of implementation-dependent and undefined aspects of a C implementation. Much later, the ANSI/ISO C++ committee had a stream of formal definition experts explain their techniques and tools and give their opinion of the extent to which a genuine formal approach to the definition of C++ would help us in the standards effort. My conclusion is that with the current state of the art, and certainly with the state of the art in the early 1980s, a formal definition of a language that is not designed together with a formal definition method is beyond the ability of all but a handful of experts in formal definition.

This confirms my conclusion at the time. However, that left us at the mercy of imprecise and insufficient terminology. Given that, what could I do to compensate? I tried to reason about new features both on my own and with others to check my logic. However, I soon developed a healthy disrespect for arguments (definitely including my own) because I found that it is possible to construct a plausible logical argument for just about any feature. On the other hand, you simply don't get a useful language by accepting every feature that makes life better for someone. There are far too many reasonable features and no language could provide them all and stay coherent. Consequently, wherever possible, I tried to experiment.

My impression was and is that many programming languages and tools represent solutions looking for problems, and I was determined that my work should not fall into that category. Thus, I follow the literature on programming language and the debates about programming languages primarily looking for ideas for solutions to problems my colleagues and I have encountered in real applications. Other programming languages constitute a mountain of ideas and inspiration – but it has to be mined carefully to avoid featurism and inconsistencies. The main sources for ideas for C++ were Simula, Algol68, and later Clu, Ada, and ML. The key to good design is insight into problems, not the provision of the most advanced features.

3.6 The C++ Programming Language (1st edition)

In the fall of 1983 my next door neighbor at work, Al Aho, suggested that I write a book on C++ structured along the lines of Brian Kernighan and Dennis Ritchie's ''The C Programming Language'' based on my published papers, internal memoranda, and the C++ reference manual. Completing the book took nine months.

The preface mentions the people who had by then contributed most to C++: Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, and Jon Shopiro. My criteria for adding a person to that list was that I was able to identify a specific C++ feature that the person has caused to be added.

The book's opening line ''C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer'' was deleted twice by reviewers who refused to believe that the purpose of programming language design could be anything but some serious mutterings about productivity, management, and software engineering. However,

> ''C++ was designed primarily so that the author and his friends would not have to program in assembler, C, or various modern high-level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer.''

This was the case whether those reviewers were willing to believe it or not. The focus of my work is the person, the individual (whether part of a group or not), the programmer. This line of reasoning has been strengthened over the years and is even more prominent in the 2nd edition

[Stroustrup,1991].

''The C++ Programming Language'' was the definition of C++ and the introduction to C++ for an unknown number of programmers and its presentation techniques and organization (borrowed with acknowledgements if not always sufficient skill from ''The C Programming Language'') have become the basis for an almost embarrassing number of articles and books. It was written with a fierce determination not to preach any particular programming technique. In the same way as I feared to build limitations into the language out of ignorance and misguided paternalism, I didn't want the book to turn into a ''manifesto'' for my personal preferences.

3.7 The ''whatis?'' Paper

Having shipped release 1.0 and sent the camera ready copy of the book to the printers I finally found time to re-consider larger issues and to document overall design issues. Just then Karel Babcisky (the chairman of the Association of Simula Users) phoned from Oslo with an invitation to give a talk on C++ at the 1986 ASU conference in Stockholm. Naturally, I wanted to go but I was worried that presenting C++ at a Simula conference would be seen as a vulgar example of self-advertisement and an attempt to steal users away from Simula. After all, I said, C++ is not Simula so why would Simula-users want to hear about it. Karel replied ''Ah, we are not hung up on syntax.'' This provided me with an opportunity to write not only about what C++ was but also what it was supposed to be and where it didn't measure up to those ideals. The result was the paper *What is ''Object-Oriented Programming''?* [Stroustrup86] that I presented to the ASU conference in Stockholm.

The significance of this paper is that it is the first exposition of the set of techniques that C++ was aiming to provide support for. All previous presentations, to avoid dishonesty and hype, had been restricted to describe what features were already implemented and in use. The ''whatis paper'' defined the set of problems I thought a language supporting data abstraction and object-oriented programming ought to solve and gave examples of language features needed.

The result was a re-affirmation of the importance of the ''multi-paradigm'' nature of C++:

> ''Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment. To be general purpose, a language supporting data abstraction or object-oriented programming must enable effective use of traditional hardware.''

The importance of static type checking was also strongly emphasized. In other words, C++ follows the Simula rather than the Smalltalk model of inheritance and type checking:

> ''a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class) whereas a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.''

This has deep implications on the way one design systems and on what language facilities are needed. A dynamically typed language such as Smalltalk simplifies the design and implementation of libraries by postponing type checking to run time. For example (using C++ syntax):

```
stack cs;
cs.push(new Saab900);
cs.pop()->takeoff();  // Oops! Run time error:
                      // a car does not have a takeoff method.
```

This delayed type error detection was considered unacceptable for C++, yet there had to be a way of matching the notational convenience and the standard libraries of a dynamically typed

language. The notion of parameterized types was presented as the (future) solution for that problem in C++:

```
stack(plane*) cs;

cs.push(new Saab37b);  // ok a Saab37b is a plane
cs.push(new Saab900);  // error, type mismatch:
                       // car passed, plane* expected

cs.pop()->takeoff();   // no run-time check needed
cs.pop()->takeoff();   // no run-time check needed
```

The key reason for considering compile time detection of such problems essential was the observation that C++ is often used for programs executing where no programmer is present. Fundamentally, the notion of static type checking was seen as the best way of providing as strong guarantees as possible for a program rather than merely a way of gaining run-time efficiency.

The ''whatis'' paper lists three aspects in which C++ was deficient:

[1] ''Ada, Clu, and ML support parameterized types. C++ does not; the syntax used here is simply devised as an illustration. Where needed, parameterized classes are ''faked'' using macros. Parameterized classes would clearly be extremely useful in C++. They could easily be handled by the compiler, but the current C++ programming environment is not sophisticated enough to support them without significant overhead and/or inconvenience. There need not be any run-time overheads compared with a type specified directly.''

[2] ''As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: ''exceptional circumstances'') become important. Ada, Algol68, and Clu each support a standard way of handling exceptions. Unfortunately, C++ does not. Where needed exceptions are ''faked'' using pointers to functions, ''exception objects,'' ''error states'', and the C library `signal` and `longjmp` facilities. This is not satisfactory in general and fails even to provide a standard framework for error handling.''

[3] ''Given this explanation it seems obvious that it might be useful to have a class `B` inherit from two base classes `A1` and `A2`. This is called multiple inheritance''

All three facilities were linked to the need to provide better (that is, more general, more flexible) libraries. All are now available in C++. Note that adding multiple inheritance and templates was considered as early as 1982 [Stroustrup,1982].

4 C++ Release 2.0

Now (Mid 1986) the course for C++ was set for all who cared to see. The key design decisions were made. The direction of the future evolution was set with the aim for parameterized types, multiple inheritance, and exception handling. Much experimentation and adjustment based on experience was needed, but the glory days were over. C++ had never been silly putty, but there was now no real possibility for radical change. For good and bad, what was done was done. What was left was an incredible amount of solid work. At this point C++ had about 2,000 users worldwide.

This was the point where the plan – as originally conceived by Steve Johnson and me – was for a development and support organization to take over the day-to-day work on the tools (primarily Cfront), thus freeing me to work on the new features and the libraries that was expected to depend on them. This was also the point where I expected first AT&T and then others would start to build compilers and other tools to eventually make Cfront redundant.

Actually, they had already started, but the good plan was soon derailed due to management indecisiveness, ineptness, and lack of focus. A project to develop a brand new C++ compiler diverted attention and resources from Cfront maintenance and development. A plan to ship a release 1.3 in early 1988 completely fell through the cracks. The net effect was that we had to

wait until June 1989 for release 2.0, and that even though 2.0 was significantly better than release 1.2 in almost all ways, 2.0 did not provide the language features outlined in the ''whatis paper'' and (consequently) a significantly improved and extended library wasn't part of it.

Many of the people who influenced C with Classes and the original C++ continued to help with the evolution in various ways. Phil Brown, Tom Cargill, Jim Coplien, Steve Dewhurst, Keith Gorlen, Laura Eaves, Bob Kelley, Brian Kernighan, Andy Koenig, Archie Lachner, Stan Lippman, Larry Mayka, Doug McIlroy, Pat Philip, Dave Prosser, Peggy Quinn, Roger Scott, Jerry Schwarz, Jonathan Shopiro, and Kathy Stark were explicitly acknowledged in [Stroustrup,1989b].

Stability of the language definition and its implementation was considered essential. The features of 2.0 were fairly simple modifications of the language based on experience with the 1.* releases. The most important aspect of release 2.0 was that it increased the generality of the individual language features and improved their integration into the language.

4.1 Feature Overview

The main features of 2.0 were first presented in [Stroustrup,1987c] and summarized in the revised version of that paper [Stroustrup,1989b] that accompanied 2.0 as part of its documentation:

[1] multiple inheritance,
[2] type-safe linkage,
[3] better resolution of overloaded functions,
[4] recursive definition of assignment and initialization,
[5] better facilities for user-defined memory management,
[6] abstract classes,
[7] static member functions,
[8] `const` member functions,
[9] `protected` members (first provided in release 1.2),
[10] overloading of operator `->`, and
[11] pointers to members (first provided in release 1.2).

Most of these extensions and refinements represented experience gained with C++ and couldn't have been added earlier without more foresight than I possessed. Naturally, integrating these features involved significant work, but it was most unfortunate that this was allowed to take priority over the completion of the language as outlined in the ''whatis'' paper.

Most features enhanced the safety of the language in some way or other. Cfront 2.0 checked the consistency of function types across separate compilation units (type-safe linkage), made the overload resolution rules order independent, and also ensured that more calls were considered ambiguous. The notion of `const` was made more comprehensive, pointers to members closed a loophole in the type system, and provided explicit class-specific memory allocation and deallocation operations to make the error-prone ''assignment to `this`'' technique redundant.

To some people, the most important ''feature'' of release 2.0 wasn't a feature at all but a simple space optimization. From the beginning, the code generated by Cfront tended to be pretty good. As late as 1992, Cfront generated the fastest running code in a benchmark used to evaluate C++ compilers on a Sparc. There have been no significant improvements in Cfront's code generation since Release 1.0. However, release 1.* was wasteful because each compilation unit generated its own set of virtual function tables for all the classes used in that unit. This could lead to megabytes of waste. At the time (about 1984), I considered the waste necessary in the absence of linker support and asked for such linker support. By 1987 that linker support hadn't materialized. Consequently, I re-thought the problem and solved it by the simple heuristic of laying down the virtual function table of a class right next to its first non-virtual non-inline function.

4.2 Multiple Inheritance

In most people's minds multiple inheritance, the ability to have two or more direct base classes, is *the* feature of 2.0. I disagreed at the time because I felt that the sum of the improvements to the type system were of far greater practical importance. Also, adding multiple inheritance in 2.0 was a mistake. Multiple inheritance belongs in C++ but is far less important than parameterized types. As it happened, parameterized types in the form of templates only appeared in release 3.0. There were a couple of reasons for choosing to work on multiple inheritance at the time: The design was further advanced and the implementation could be done within Cfront. Another factor was purely irrational. Nobody doubted that I could implement templates efficiently. Multiple inheritance, on the other hand, was widely supposed to be very difficult to implement efficiently. Thus multiple inheritance seemed more of a challenge and since I had considered it as early as 1982 and found a simple and efficient implementation technique in 1984 I couldn't resist the challenge. I suspect that this is the only case where fashion affected the sequence of events.

In September 1984, I presented the C++ operator overloading mechanism at the IFIP WG2.4 conference in Canterbury [Stroustrup,1984c]. There, I met Stein Krogdahl from the University of Oslo who was just finishing a proposal for adding multiple inheritance to Simula [Krogdahl,1984]. His ideas became the basis for the implementation of ordinary multiple base classes in C++. He and I later found out that the proposal was almost identical to an idea for providing multiple inheritance in Simula that had been considered by Ole-Johan Dahl in 1966 and rejected because it would have complicated the Simula garbage collector [Dahl,1988].

The original and fundamental reason for considering multiple inheritance was simply to allow two classes to be combined into one in such a way that objects of the resulting class would behave as objects of either base class [Stroustrup,1986c]:

> ''A fairly standard example of the use of multiple inheritance would be to provide two library classes `displayed` and `task` for representing objects under the control of a display manager and co-routines under the control of a scheduler, respectively. A programmer could then create classes such as
>
> ```
> class my_displayed_task : public displayed, public task {
>  // ...
> };
>
> class my_task : public task {  // not displayed
>  // ...
> };
>
> class my_displayed : public displayed  {   // not a task
>  // ...
> };
> ```
>
> Using (only) single inheritance only two of these three choices would be open to the programmer.''

The implementation requires little more than remembering the relative offsets of the `task` and `displayed` objects in a `my_displayed_task` object. All the gory implementation details were explained in [Stroustrup,1987a]. In addition, the language design must specify how ambiguities are handled and what to do if a class is specified as a base class more than once in a derived class:

> ''Ambiguities are handled at compile time:

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B {  /* no f() ... */ };

void g() {
        C* p;
        p->f(); // error: ambiguous
}
```

In this, C++ differs from the object-oriented Lisp dialects that support multiple inheritance.'' Basically, I rejected all forms of dynamic resolution beyond the use of virtual functions as unsuitable for a statically typed language under severe efficiency constraints. Maybe, I should at this point have revived the notion of `call` and `return` functions (§2.4.8) to mimic the CLOS `:before` and `:after` methods. However, people were already worrying about the complexity of the multiple inheritance mechanisms and I am always reluctant to re-open old wounds.

Multiple inheritance in C++ became controversial [Cargill,1991] [Carroll,1991] [Waldo,1991] [Sakkinen,1992] for several reasons. The arguments against it centered around the real and imaginary complexity of the concept, the utility of the concept, and the impact of multiple inheritance on other extensions and tool building. In addition, proponents of multiple inheritance can and do argue over exactly what multiple inheritance is supposed to be and how it is best supported in a language. I think – as I did then – that the fundamental flaw in these arguments is that they take multiple inheritance far too seriously. Multiple inheritance doesn't solve all of your problems, but it doesn't need to because it is quite cheap, and sometimes it is very convenient to have. Grady Booch [Booch,1991] expresses a slightly stronger sentiment: ''Multiple inheritance is like a parachute, you don't need it very often, but when you do it is essential.''

4.3 Abstract Classes

The very last feature added to 2.0 before it shipped was abstract classes. Late modification to releases are never popular and late changes to the definition of what will be shipped are even less so. I remember that several members of management thought I had lost contact with the real world when I insisted on this feature.

A common complaint about C++ was (and is) that private data is visible and that when private data is changed then code using that class must be recompiled. Often, this complaint is expressed as ''abstract types in C++ aren't really abstract.'' What I hadn't realized was that many people thought that because they *could* put the representation of an object in the private section of a class declaration then they actually *had to* put it there. This is clearly wrong (and that is how I failed to spot the problem for years). If you don't want a representation in a class, thus making the class an interface only, then you simply delay the specification of the representation to some derived class and define only virtual functions. For example, one can define a `set` of `T` pointers like this:

```
class set {
public:
    virtual void insert(T*);
    virtual void remove(T*);

    virtual int is_member(T*);

    virtual T* first();
    virtual T* next();

    virtual ~set() { }
};
```

This provides all the information that people need to use a `set` except that whoever actually

creates a `set` must know something about how some particular kind of `set` is represented. For example, given:

```
class slist_set : public set, private slist {
    slink* current_elem;
public:
    void insert(T*);
    void remove(T*);

    int is_member(T*);

    virtual T* first();
    virtual T* next();

    slist_set() : slist(), current_elem(0) { }
};
```

we can create `slist_set` objects that can be used as `sets` by users that have never heard of a `slist_set`.

The only problem was that in C++ as defined before 2.0 there was no explicit way of saying ‘‘The `set` class is just an interface: its functions need not be defined, it is an error to create objects of class `set`, and anyone who derives a class from `set` must define the virtual functions specified in `set`.’’ Release 2.0 allowed a class to be declared explicitly *abstract* by declaring one or more of its virtual functions ‘‘pure’’ using the syntax `=0`:

```
class set {                          // abstract class
public:
    virtual void insert(T*) = 0;  // pure virtual function
    virtual void remove(T*) = 0;

    // ...
};
```

The `=0` syntax wasn’t exactly brilliant, but it expresses the desired notion of a pure virtual function in a way that is terse and fits the use of `0` to mean ‘‘nothing’’ or ‘‘not there’’ in C and C++. The alternative, introducing a new keyword, say `pure`, wasn’t an option. Given the opposition to abstract classes as a ‘‘late and unimportant change,’’ I would never simultaneously have overcome the traditional, strong, widespread, and emotional opposition to new keywords in parts of the C and C++ community.

The importance of the abstract class concept is that it allows a cleaner separation between a user and an implementor than is possible without it. This limits the amount of recompilation necessary after a change and also the amount of information necessary to compile an average piece of code. By decreasing the coupling between a user and an implementor, abstract classes provide an answer to people complaining about long compile times and also serve library providers who must worry about the impact to users of changes to a library implementation. I had unsuccessfully tried to explain these notions in [Stroustrup,1986b]. With an explicit language feature supporting abstract classes I was much more successful [Stroustrup,1991].

5 The Explosion in Interest and Use

C++ was designed to serve users. It was not an academic experiment to design the perfect programming language. Nor was it a commercial product meant to enrich its developers. Thus to fulfill its purpose C++ had to have users – and it had:

C++ use	
Date	estimated number of users
Oct 1979	1
Oct 1980	16
Oct 1981	38
Oct 1982	85
Oct 1983	??+2 (no Cpre count)
Oct 1984	??+50 (no Cpre count)
Oct 1985	500
Oct 1986	2,000
Oct 1987	4,000
Oct 1988	15,000
Oct 1989	50,000
Oct 1990	150,000
Oct 1991	400,000

In other words, the C++ user population doubled every 7.5 months or so. These are conservative figures. The actual number of C++ users has never been easy to count. Firstly, there are implementations such as GNU's G++ and Cfront shipped to universities for which no meaningful records can be kept. Secondly, many companies, both tools suppliers and end-users, treat the number of their users and the kind of work they do like state secrets. However, I always had many friends, colleagues, contacts, and many compiler suppliers who were willing to trust me with figures as long as I used them in a responsible manner. This enabled me to estimate the number of C++ users. These estimates are created by taking the number of users reported to me or estimated based on personal experience, rounding them all down, adding them, and then rounding down again. These number are the estimates made at the time and not adjusted in any way. To support the claim that these figures are conservative, I can mention that Borland, the largest single C++ compiler supplier, publicly stated that it had shipped 500,000 compilers by October 1991.

Early users had to be gained without the benefit of traditional marketing. Various forms of electronic communication played a crucial role in this. In the early years most distribution and all support was done using email and relatively early on newsgroups dedicated to C++ were created (*not* at the initiative of Bell Labs employees) that allowed a wider dissemination of information about the language, techniques, and the current state of tools. These days this is fairly ordinary, but in 1981 it was relatively new. I think that only the spread of Interlisp over the Arpanet provides a contemporary parallel.

Later, more conventional forms of communication and marketing arose. After AT&T released Cfront 1.0 some resellers, notably Glockenspiel in Ireland and their US distributor Oasys (later part of Green Hills) started some minimal advertising in 1986, and when independently developed C++ compilers such as Oregon Software's C++ Compiler (developed by Mike Ball at Tau-Metric Software in San Diego) and Zortech's C++ Compiler (developed by Walter Bright in Seattle) appeared 'C++' became a common sight in ads (from about 1988).

5.1 Conferences

In 1987 USENIX, the UNIX Users' association, took the initiative to hold the first conference specifically devoted to C++. Thirty papers were presented to 214 people in Santa Fe, NM in November of 1987.

The Santa Fe conference set a good example for future conferences with a mix of papers on applications, programming and teaching techniques, ideas for improvements to the language, libraries, and implementation techniques. Notably for a USENIX conference, there were papers

on C++ on the Apple MAC, OS/2, the Connection machine, and for implementing non-UNIX operating systems (for example, CLAM [Call,1987] and Choices [Campbell,1987]). The NIH library [Gorlen,1987] and the Inverviews library [Linton,1987] also made their public debut in Santa Fe. An early version of what became Cfront 2.0 was demonstrated and I gave the first public presentation of its features. The USENIX C++ conferences continue as the primary technically and academically oriented C++ conference. The proceedings from these conferences are among the best reading about C++ and its use.

In addition to the USENIX C++ conferences, there are now many commercial and semi-commercial conferences devoted to C++, to C including C++, and to Object-Oriented Programming.

5.2 Journals and Books

By 1991 there were more than 60 books on C++ available in English alone and both translations and locally written books available in languages such as Chinese, Danish, French, German, and Japanese. Naturally, the quality varies enormously.

The first journal devoted to C++, ''*The C++ Report*'' from SIGS publications, started publishing in January 1989 with Rob Murray as its editor, A larger and glossier quarterly ''*The C++ Journal*'' appeared in the spring of 1991. In addition there are several newsletters controlled by C++ tools suppliers and many journals such as *Computer Language*, *The Journal of Object-Oriented Programming*, *Dr. Dobbs Journal*, *The C Users' Journal*, run regular columns or features on C++. Andrew Koenig's column in JOOP is particularly consistent in its quality and lack of hype.

Newsgroup and bulletin boards such as comp.lang.c++ on usenet and c.plus.plus on BIX also produced tens of thousands of messages over the years to the delight and despair of the readers. Keeping up with what is written about C++ is currently more than a full time job.

5.3 Libraries

The very first real code to be written in C with Classes was the task library [Stroustrup,1980b] providing Simula-like concurrency for simulation. The first real programs were simulations of network traffic, circuit board layout, etc. using the task library. The task library is still heavily used today. The standard C library was available from C++ – without overhead or complication compared with C – from day one. So are all other C libraries. Classical data types such as character strings, range checked arrays, dynamic arrays, and lists were among the examples used to design C++ and test its early implementations.

The early work with container classes such as list and array were severely hampered by the lack of support for a way of expressing parameterized types in C with Classes and in C++ up until version 3.0. In the absence of proper language support, (later provided in the form of templates (§6.3) we had to made do with macros. The best that can be said for the C preprocessor's macro facilities is that it allowed us to gain experience with parameterized types and support individual and small group use.

Much of the work on designing classes was done in cooperation with Jonathan Shopiro who in 1983 produced list and string classes that saw wide use within AT&T and are the basis for the classes currently found in the ''Standard Components'' library that was developed in Bell labs and is now sold by USL. The design of these early libraries interacted directly with the design of the language and in particular with the design of the overloading mechanisms.

5.3.1 The Stream I/O Library

C's `printf` family of functions is an effective and often convenient I/O mechanism. It is not, however, type safe or extensible to user-defined types (classes). Consequently, I started looking for a type safe, terse, extensible, and efficient alternative to the `printf` family. Part of the inspiration came from the last page and a half of the Ada Rationale [Ichbiah,1979], which is an argument that you cannot have a terse and type-safe I/O library without special language features to support it. I took that as a challenge. The result was the stream I/O library that was first implemented in 1984 and presented in [Stroustrup,1985]. Soon after, Dave Presotto reimplemented the stream library without changing the interfaces.

To introduce stream I/O this example was considered:

```
fprintf(stderr,"x = %s\n",x);
```

Because `fprintf()` relies on unchecked arguments that are handled according to the format string at run time this is not type safe and

> ''had `x` been a user-defined type like `complex` there would have been no way of specifying the output format of `x` in the convenient way used for types ''known to `printf()`'' (for example, `%s` and `%d`). The programmer would typically have defined a separate function for printing complex numbers and then written something like this:
>
> ```
> fprintf(stderr,"x = ");
> put_complex(stderr,x);
> fprintf(stderr,"\n");
> ```
>
> This is inelegant. It would have been be a major annoyance in C++ programs that use many user-defined types to represent entities that are interesting/critical to an application.
>
> Type-security and uniform treatment can be achieved by using a single overloaded function name for a set of output functions. For example:
>
> ```
> put(stderr,"x = ");
> put(stderr,x);
> put(stderr,"\n");
> ```
>
> The type of the argument determines which ''put function'' will be invoked for each argument. However, this is too verbose. The C++ solution, using an output stream for which `<<` has been defined as a ''put to'' operator, looks like this:
>
> ```
> cerr << "x = " << x << "\n";
> ```
>
> where `cerr` is the standard error output stream (equivalent to the C `stderr`). So, if `x` is an `int` with the value `123`, this statement would print
>
> ```
> x = 123
> ```
>
> followed by a newline onto the standard error output stream.
>
> This style can be used as long as `x` is of a type for which operator `<<` is defined, and a user can trivially define operator `<<` for a new type. So, if `x` is of the user-defined type `complex` with the value `(1,2.4)`, the statement above will print
>
> ```
> x = (1,2.4)
> ```
>
> on `cerr`.
>
> The stream I/O facility is implemented exclusively using language features available to every C++ programmer. Like C, C++ does not have any I/O facilities built into the language. The stream I/O facility is provided in a library and contains no ''extra-linguistic magic''.''

The idea of providing an output operator rather than a named output function was suggested by Doug McIlroy. This requires operators that return their left-hand operand for use by further operations.

In connection with release 2.0, Jerry Schwarz reimplemented and partially redesigned the streams library to serve a larger class of applications and to be more efficient for file I/O. A significant improvement was the use of Andrew Koenig's idea of manipulators [Stroustrup,1991] to control formatting details such as the precision used for floating point output. Experience with streams was a major reason for the change to the basic type system and to the overloading rules to allow `char` values to be treated as characters rather than small integers the way they are in C. For example:

```
char ch = 'b';
cout << 'a' << ch;
```

would in Release 1.* output a string of digits reflecting the integer values of the characters `a` and `b` whereas Release 2.* outputs `ab` as one would expect.

5.3.2 Other Libraries

There were and are many other significant C++ libraries. These will be mentioned only briefly here because even though they were essential to their users they did not affect the development of C++ significantly. They are, however, most significant to their users and most users' view of C++ is strongly affected or even dominated by a library.

The most significant early libraries was Keith Gorlen's [Gorlen,1990] NIH class library that provides a Smalltalk-like set of classes and Mark Linton's Interviews library [Linton,1987] that makes use of the X windows system convenient from C++. GNU C++ (G++) comes with a library designed by Doug Lea that is distinguished by heavy use of abstract base classes. Rogue Wave and Dyad supply large sets of libraries primarily aimed at scientific uses. Glockenspiel has for years supplied libraries for various commercial uses. Rational ships a C++ version of ''The Booch Components'' that was originally designed for and implemented in Ada by Grady Booch. Grady Booch and Mike Vilot designed and implemented the C++ version. The Ada version is 150,000 non-commented source lines compared to the C++ version's 10,000 lines – inheritance combined with templates can be a very powerful mechanism for organizing libraries without loss of performance or clarity.

This is only a very short list of early libraries to indicate the diversity of C++ libraries. Many more libraries exists. In particular, most tools suppliers provide foundation libraries for their users. It seems that the ''software components'' industry that pundits have promised for years – and bemoaned the lack of – has finally come into existence.

5.4 Compilers

The Santa Fe conference (§5.1) marked the announcement of the second wave of C++ implementations. Steve Dewhurst described the architecture of a compiler he and others were building in AT&T's Summit facility, Mike Ball presented some ideas for what became the TauMetric C++ compiler (more often known as the Oregon Software C++ compiler), and Mike Tiemann gave a most animated and interesting presentation of how the GNU G++ he was building would do just about everything and put all other C++ compiler writers out of business. The new AT&T C++ compiler never materialized; GNU C++ version 1.13 was first released in December 1987; and TauMetric C++ first shipped in January 1988.

Until June 1988 all C++ compiler on PCs were Cfront ports. Then Zortech started shipping their compiler. The appearance of Walter Bright's compiler made C++ ''real'' for many PC-oriented people for the first time. More conservative people reserved their judgement until the Borland C++ compiler in May 1990 or even Microsoft's C++ compiler in March 1992. DEC released their first independently developed C++ compiler in February 1992 and IBM released their first independently developed C++ compiler in May 1992. In all there are now more than a dozen independently developed C++ compilers.

In addition to these compilers, Cfront ports seemed to be everywhere. In particular, Sun, HP,

Centerline, ParcPlace, Glockenspiel, and Comeau Computing ship Cfront-based products on just about any platform.

5.5 Tools and Environments

C++ was designed to be a viable language in a tool-poor environment. This was partly a necessity because of the almost complete lack of resources in the early years and the relative poverty later on. It was also a conscious decision to allow simple implementations and in particular simple porting of implementations.

C++ programming environments are now emerging that are a match for the environments habitually supplied with other object-oriented languages. For example, ObjectWorks for C++ from ParcPlace is essentially the best Smalltalk program development environment adapted for C++, and Centerline C++ (formerly Saber C++) is an interpreter-based C++ environment inspired by the interlisp environment. This gives C++ programmers the option of using the more whizzy, more expensive, and often more productive environments that have previously only been available for other languages and/or as research toys. An environment is a framework in which tools can cooperate. There is now a host of such environments for C++: Most C++ implementations on PCs are compilers embedded in a framework of editors, tools, file systems, standard libraries, etc. MacApp and the Mac MPW is the Apple Mac version of that, ET++ is a public domain version in the style of the MacApp. Lucid's Energize and HP's Softbench are yet other examples.

5.6 Commercial Competition

Commercial competitors were largely ignored and the C++ language was developed according to the original plan, its own internal logic, and the experience of its users. There was (and is) always much discussion among programmers, in the press, at conferences, and on the electronic bulletin boards about which language ''is best'' and which language will ''win'' in some sort of competition for users. Personally, I consider much of that debate misguided and uninformed, but that doesn't make the issues less real to a programmer, manager, or professor who has to choose a programming language for his or her next project. For good and bad, people debate programming languages with an almost religious fevor and often consider the choice of programming language the most important choice of a project or organization.

In the early years, Modula-2 [Wirth,1982] was by many considered a competitor to C++. However, until the commercial release of C++ in 1985, C++ could hardly be considered a competitor to any language, and by then Modula-2 seemed to me to have been largely outcompeted by C. Later it was popular to speculate about whether C++ or Objective C [Cox,1986] was to be ''*the* Object-Oriented C.'' Ada [Ichbiah,1979] was often a possible choice of organizations who might use C++. In addition, Smalltalk [Goldberg,1983] and some object-oriented variant of Lisp [Kiczales,1992] would often be considered for applications that did not require hard-core systems work or maximum performance. Lately, some people have been comparing C++ with Eiffel [Meyer,1988] and Modula-3 [Nelson,1991] for some uses.

My personal view is different. The main competitor to C++ was C. The reason that C++ is the most widely used object-oriented language today is that it was/is the only one that could consistently match C on C's own turf and that allows a transition path from C to a style of system design and implementation based on a more direct mapping between application level concepts and language concepts (usually called ''data abstraction'' or ''object-oriented programming.''). Secondarily many organizations that consider a new programming language have a tradition for the use of an in-house language (usually a Pascal variant) or Fortran. Except for serious scientific computation these languages can be considered roughly equivalent to C when compared with C++.

In the secondary competition between C++ and other newer languages supporting abstraction mechanisms (object-oriented programming languages, languages supporting data abstraction)

C++ was during the early years (1984 to 1989) consistently the underdog as far as marketing was concerned. In particular, AT&T's marketing budget during that period was usually empty and AT&T's total spending on C++ advertising was about $3,000. To this day, most of AT&T's visibility in the C++ arena relies on Bell Labs' traditional policy of encouraging developers and researchers to give talks, write papers, and attend conferences rather than on any deliberate policy to promote C++. Within AT&T, C++ was also a grass-roots movement without money or management clout. Naturally, coming from AT&T Bell Labs helps C++, but that help is earned the hard way by surviving in a large-company environment.

In competition, C++'s fundamental strength is its ability to operate in a traditional environment (social and computer-wise), its run-time and space efficiency, the flexibility of its class concept, its low price, and its non-proprietary nature. Its weaknesses compared to newer languages are some of the uglier parts inherited from C, its lack of spectacular new features (such as built-in data base support), its lack of spectacular program development environments (only lately have C++ environments of the sort people have taken for granted for Smalltalk and Lisp become available for C++), its lack of standard libraries (only lately have major libraries become widely available for C++ – and they are not ''standard''†), and its lack of salesmen to balance the efforts of richer competitors. With C++'s recent dominance in the market the last factor has disappeared. Some C++ salesmen will undoubtedly embarrass the C++ community by emulating some of the sleazy tricks and unscrupulous practices that salesmen and admen have used to attempt to derail C++'s progress.

An important factor, both for and against C++, was the willingness of the C++ community to acknowledge C++'s many imperfections. This openness is reassuring to many that have become cynics from years of experience with the people and products of the software tools industry, but also infuriating to perfectionists and a fertile source for fair and not-so-fair criticism of C++. On balance, I think that tradition of throwing rocks at C++ within the C++ community has been a major advantage. It kept us honest, kept us busy improving the language and its tools, and kept the expectations of C++ users and would-be users realistic.

In competition with traditional languages, C++'s inheritance mechanism was a major plus. In competition with languages with inheritance, C++'s static type checking was a major plus. Of the languages mentioned, only Eiffel and Modula-3 combines the two in a way similar to C++. It is widely assumed that Ada will be revised to include inheritance [Tucker,1992].

C++ was designed to be a systems programming language and a language for applications that had a large ''systems-like'' component. This was the area that my friends and I knew well. The decision not to compromise C++'s strengths in this area to broaden its appeal has been crucial in its success. Only time will tell if this has also compromised its ability to appeal to an even larger audience. I would not consider that a tragedy because I am not among those that think that a single language should be all things to all people and C++ already serves the community it was designed for well. However, I suspect that through the design of libraries C++'s appeal will be very wide.

---

† This problem has now been remedied. The standard C++ library provides standard containers, such as list, vector, map, set, etc. and a library of sorting, searching, etc. algorithms operating on these containers. These containers and algorithms are all templates. This part of the standard library is based on the work of Alex Stepanov [Stepanov,1994]. In addition, the standard library provides a vector type with associated operations to support numeric calculations based on the work of Ken Budge.

6 Standardization

Sometime in 1988 it became clear that C++ would eventually have to be standardized [Stroustrup,1989]. There were now a handful of independent implementations in use or being produced and clearly an effort had to be made to write a more precise and comprehensive definition of the language and also to gain wide acceptance for that definition. At first, formal standardization wasn't considered an option. Many people involved with C++ considered – and still consider – standardization before genuine experience has been gained abhorrent. However, making an improved reference manual wasn't something that could be done by one person (me) in private. Input and feedback from the C++ community was needed. Thus I came upon the idea of re−writing the C++ reference manual and circulating its draft among important and insightful members of the C++ community worldwide.

6.1 The Annotated Reference Manual

At about the same time, the part of AT&T that sold C++ commercially wanted a new and improved C++ reference manual and gave Margaret Ellis the task of writing it. It seemed only reasonable to combine the efforts and produce a single, externally reviewed reference manual. It also seemed obvious to me that publishing this manual with some additional information would help the acceptance of the new definition and make C++ more widely understood. Thus, the Annotated C++ Reference Manual was written ''to provide a firm basis for the further evolution of C++ ... (and) to serve as a starting point for the formal standardization of C++ [Ellis,1990].''

> ''The C++ reference manual alone provides a complete definition of C++, but the terse reference manual style leaves many reasonable questions unanswered. Discussions of what is *not* in the language, *why* certain features are defined as they are, and *how* one might implement some particular feature have no place in a reference manual but are nevertheless of interest to most users. Such discussions are presented as annotations and in the commentary sections.
>
> The commentary also helps the reader appreciate the relationships among different parts of the language and emphasizes points and implications that might have been overlooked in the reference manual itself. Examples and comparisons with C also make this book more approachable than the bare reference manual [Ellis,1990].''

After some minor squabbling with the product people it was agreed that we'd write the ARM (as ''The Annotated C++ Reference Manual'' came to be popularly called) describing the whole of C++, that is with templates and exception handling, rather than as a manual for the subset implemented by the most recent AT&T release. This was important because it clearly established the language itself as different from any one implementation of it. This principle had been present from the very beginning, but needs to be restated often because users and implementors seem to have difficulties remembering it.

Of the ARM, I wrote every word of the reference manual proper except the section on the preprocessor that Margaret Ellis adopted from the C Standard. The annotations were jointly written and partly based on my earlier papers [Stroustrup,1987a] [Stroustrup,1987c] [Stroustrup,1988a] [Stroustrup,1988b].

The reference manual proper of the ARM was reviewed by about a hundred people from two dozen organizations. Most are named in the acknowledgement section of the ARM. In addition, many contributed to the whole of the ARM. The contributions of Brian Kernighan, Andrew Koenig, and Doug McIlroy were specifically noted. The reference manual proper from the ARM was accepted as the basis for the ANSI standardization of C++ in March 1990.

The ARM doesn't attempt to explain the techniques that the language features support. That job was left for the second edition of *The C++ Programming Language* [Stroustrup,1991].

6.2 Minor Features

The ARM presented a few minor features that were not implemented until 2.1 releases from AT&T and other C++ compiler vendors. The most obvious of these were nested classes. I was strongly encouraged to revert to the original definition of nested class scopes by comments from external reviewers of the reference manual. I also despaired over ever getting the scope rules of C++ coherent while the C rule was in place (§3.4).

The ARM allowed people to overload prefix and postfix increment (++) independently. The main impetus for that came from people who wanted ‘‘smart pointers’’ that behaved exactly like ordinary pointers except for some added work done ‘‘behind the scenes.’’

6.3 Templates

In the original design of C++, parameterized types (templates) were considered but postponed because there wasn’t time to do a thorough job of exploring the design and implementation issues. I first presented templates at the 1988 USENIX C++ conference in Denver [Stroustrup,1988b]:

> ‘‘For many people, the largest single problem using C++ is the lack of an extensive standard library. A major problem in producing such a library is that C++ does not provide a sufficiently general facility for defining ‘‘container classes’’ such as lists, vectors, and associative arrays.’’

There are two approaches for providing such classes/types: One can either rely on dynamic typing and inheritance like Smalltalk does, or one can rely on static typing and a facility for arguments of type *type*. The former is very flexible, but carries a high run-time cost, and more importantly defies attempts to use static type checking to catch interface errors. Therefore, the latter approach was chosen.

A C++ parameterized type is called a class template. A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. A vector class template might be declared like this:

```
template<class T> class vector {
        T* v;
        int sz;
public:
        vector(int);
        T& operator[](int);
        T& elem(int i) { return v[i]; }
        // ...
};
```

The `template <class T>` prefix specifies that a template is being declared and that an argument `T` of type *type* will be used in the declaration. After its introduction, `T` is used exactly like other type names within the scope of the template declaration. Vectors can then be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec;   // make cvec a synonym for
                                // vector<complex>

cvec v3(40);                    // v2 and v3 are of the same type

v1[3] = 7;
v2[3] = v3.elem(4) = complex(7,8);
```

C++ does not require the user to explicitly ‘‘instantiate’’ a template; that is, the user need not specify which versions of a template needs to be generated for particular sets of template

arguments. The reason is that only when the program is complete can it be known what templates need to be instantiated. Many templates will be defined in libraries and many instantiations will be directly and indirectly caused by users that don't even know of the existence of those templates. It therefore seemed unreasonable to require the user to request instantiations (say, by using something like Ada's 'new' operator).

Avoiding unnecessary space overheads caused by too many instantiations of template functions was considered a first order – that is, language level – problem rather than an implementation detail. I considered it unlikely that early (or even late) implementations would be able to look at instantiations of a class for different template arguments and deduce that all or part of the instantiated code could be shared. The solution to this problem was to use the derived class mechanism to ensure code sharing among derived template instances.

The template mechanism is completely a compile and link time mechanism. No part of the template mechanism needs run-time support. This leaves the problem of how to get the classes and functions generated (instantiated) from templates to depend on information known only at run time. The answer was, as ever in C++, to use virtual functions and abstract classes. Abstract classes used in connection with templates also have the effect of providing better information hiding and better separation of programs into independently compiled units.

6.4 Exception Handling

Exceptions were considered in the original design of C++, but were postponed because there wasn't time to do a thorough job of exploring the design and implementation issues. Exceptions were considered essential for error handling in programs composed out of separately designed libraries.

The actual design of the C++ exception mechanism stretched from 1984 to 1989. Andrew Koenig was closely involved in the later iterations and is the co-author (with me) on the published papers [Koenig,1989a] [Koenig,1990] [Koenig,1990]. I also had meetings at Apple, DEC, Microsoft, IBM, Sun, and other places where I presented draft versions of the design and received valuable input. In particular, I searched out people with actual experience with systems providing exception handling to compensate for my personal inexperience in that area. Throughout the design effort there was an increasing influence of systems designers of all sorts and a decrease of input from the language design community. In retrospect, the greatest influence on the C++ exception handling design was the work on fault-tolerant systems started at the University of Newcastle in England by Brian Randell and his colleagues and continued in many places since.

The following assumptions were made for the design:

- Exceptions are used primarily for error handling.
- Exception handlers are rare compared to function definitions.
- Exceptions occur infrequently compared to function calls.

These assumptions, together with the requirement that C++ with exceptions should cooperate smoothly with languages without exceptions, such as C and Fortran, led to a design with multi-level propagation. The view is that not every function should be a fire-wall and that the best error-handling strategies are those where only designated major interfaces are concerned with non-local error handling issues. By allowing multi-level propagation of exceptions C++ loses one aspect of static checking. One cannot simply by looking at a function determine which exceptions it may throw. C++ compensates by providing a mechanism for specifying a list of exceptions that a function may throw.

I concluded that the ability to define groups of exceptions is essential. For example, a user must be able to catch ''any I/O library exception'' without knowing exactly which exceptions those are. Many people, including Ted Goldstein and Peter Deutsch, noticed that most such groups were equivalent to class hierarchies. We therefore adopted a scheme inspired by ML where you throw an object and catch it by a handler declared to accepts objects of that type. This

scheme naturally provides for type-safe transmission of arbitrary amounts of information from a throw point to a handler. For example:

```
class Matherr { /* ... */ };
class Overflow : public Matherr { /* ... */ };
class Underflow : public Matherr { /* ... */ };
class Zerodivide : public Matherr { /* ... */ };
class Int_add_overflow : public Overflow { /* ... */ };
// ...

try {
        f();
}
catch (Overflow& over) {
        // handle Overflow or anything derived from Overflow
}
catch (Matherr& math) {
        // handle any Matherr
}
```

Thus `f()` might be written like this

```
void f() throw(Matherr&) // f() can throw Matherr& exceptions
                         // (and only Matherr& exceptions)
{
        // ...
        if (d == 0) throw Zerodivide();
        // ...
        if ( check(x,y) ) throw Int_add_overflow(x,y);
        // ...
}
```

The `Zerodivide` will be caught by the `Matherr&` handler above and `Int_add_overflow` will be caught by the `Overflow&` handler that might access the operand values `x` and `y` passed by `f()` in the object thrown.

The central point in the exception handling design was the management of resources. In particular, if a function grabs a resource how can the language help the user to ensure that the resource is correctly released upon exit even if an exception occurs? The problem with most solutions are that they are verbose, tedious, potentially expensive and therefore error-prone. However, I noticed that many resources are released in the reverse order of their acquisition. This strongly resembles the behavior of local objects created by constructors and destroyed by destructors. Thus we can handle such resource acquisition and release problems by a suitable use of objects of classes with constructors and destructors. This technique extends to partially constructed objects and thus addresses the otherwise difficult issue of what to do when an error is encountered in a constructor.

During the design the most contentious issue turned out to be whether the exception handling mechanism should support termination semantics or resumption semantics; that is, whether it should be possible for an exception handler to require execution to resume from the point where the exception was thrown. The main resumption vs termination debate took place in the ANSI C++ committee. After a discussion that lasted for about a year, the exception handling proposal as presented in the ARM (that is, with termination semantics) was voted into C++ by an overwhelming majority. The key to that consensus was presentations of experience data based on decades of use of systems that supported both resumption and termination semantics by representatives of DEC, Sun, Texas Instruments, IBM, and others. Basically, every use of resumption had represented a failure to keep separate levels of abstraction disjoint.

The C++ exception handling mechanism is explicitly *not* for handling asynchronous events directly. This view precludes the direct use of exceptions to represent something like hitting a DEL key and the replacement of UNIX signals with exceptions. In such cases, a low-level interrupt routine must somehow do its minimal job and possibly map into something that could trigger an exception at a well-defined point in a programs execution.

As ever, efficiency was a major concern. The C++ exception handling mechanism can be implemented without any run-time overhead to a program that doesn't throw an exception [Stroustrup,1988b]. It is also possible to limit space overhead, but it is hard simultaneously to avoid run-time overhead and code size increases. The first implementations of exception handling as defined in the ARM are just appearing (Spring 1992).

6.5 ANSI and ISO

The initiative to formal (ANSI) standardization of C++ was taken by HP in conjunction with AT&T, DEC, and IBM. Larry Rosler from HP was important in this initiative. The proposal for ANSI standardization was written by Dmitry Lenkov [Lenkov,1989]. Dmitry's proposal cites several reasons for immediate standardization of C++:

- C++ is going through a much faster public acceptance than most other languages.
- Delay ... will lead to dialects.
- Requires a careful and detailed definition providing full semantics ... for each language feature.
- C++ lacks some important features ... exception handling, aspects of multiple inheritance, features supporting parametric polymorphism, and standard libraries.

The proposal also stressed the need for compatibility with ANSI C. The organizational meeting of the ANSI C++ committee, X3J16 took place in December of 1989 in Washington, D.C. and was attended by about 40 people including people who took part in the C standardization, people who by now were ''old time C++ programmers,'' and others. Dmitry Lenkov became its chairman and Jonathan Shopiro became its editor.

The committee now has more than 250 members out of which something like 70 turn up at meetings. The aim of the committee was and is a draft standard for public review in late 1993 or early 1994 with the hope of an official standard about two years later†. This is an ambitious schedule for the standardization of a general-purpose programming language. To compare, the standardization of C took 7 years.

Naturally, standardization of C++ isn't just an American concern. From the start, representatives from other countries attended the ANSI C++ meetings; and in Lund, Sweden, in June 1991 the ISO C++ committee WG21 was convened and the two C++ standards committees decided to hold joint meetings – starting immediately in Lund. Representatives from Canada, Denmark, France, Japan, Sweden, the UK, and USA were present. Notably, the vast majority of these national representatives were actually long-time C++ programmers. The C++ committee had a difficult charter:

[1] The definition of the language must be precise and comprehensive.
[2] C/C++ compatibility had to be addressed.
[3] Extensions beyond current C++ practice had to be considered.
[4] Libraries had to be considered.

On top of that, the C++ community was *very* diverse and totally unorganized so that the standards committee naturally became an important focal point of that community. In the short run, that is actually the most important role for the committee.

C compatibility was the first major controversial issue we had to face. After some – occasionally heated – debate it was decided that 100% C/C++ compatibility wasn't an option. Neither

† We still expect to see a C++ ISO standard in late-1995 to mid-1996.

was significantly decreasing C compatibility. C++ was a separate language and not a strict superset of ANSI C and couldn't be changed to be such a superset without breaking the C++ type system and without breaking millions of lines of C++ code. This decision, often referred to as ''As close to C, but no closer'' after a paper written by Andrew Koenig and me [Koenig1989a], is the same that has been reached over and over again by individuals and groups considering C++ and the direction of its evolution (§3.4).

6.6 Rampant Featurism?

A critical issue was – and is – how to handle the constant stream of proposals for language changes and extensions. The focus of that effort is the extensions working group of which I'm the chairman. It is much easier to accept a proposal than to reject it. You win friends this way and people praise the language for having so many ''neat features.'' Unfortunately, a language made as a shopping list of features without coherence will die so there is no way we could accept even most of the features that would be of genuine help to some section of the C++ community.

So how is the committee doing? We won't really know until the standard appears because there is no way of knowing which, if any, of the backlog of proposals will be accepted. There is some hope of restraint and that accepted features will be properly integrated into the language. Only three new features have been accepted so far: the ''mandated'' extensions (exception handling and templates), and a proposal for relaxing the requirements for return types for overriding functions†.

7 Retrospective

It is often claimed that hindsight is an exact science. It is not. The claim is based on the false assumptions that we know all relevant facts about what happened in the past, that we know the current state of affairs, and that we have a suitably detached point of view from which to judge the past. Typically none of these conditions hold. This makes a retrospective on something as large, complex, and dynamic as a programming language in large scale use hazardous. Anyway, let me try to stand back and answer some hard questions:

[1] Did C++ succeed at what it was designed for?
[2] Is C++ a coherent language?
[3] What was the biggest mistake?

Naturally, the replies to these questions are related. The basic answers are, 'yes,' 'yes,' and 'not shipping a larger library with release 1.0.'

7.1 Did C++ succeed at what it was designed for?

''C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer [Stroustrup1986b].'' In this, it clearly succeeded, especially in the more specific aim of letting reasonably educated and experienced programmers write programs at a higher level of abstraction (''as in Simula'') without loss of efficiency compared to C for applications that were demanding in time, space, inherent complexity, and constraints from the execution environment.

More generally, C++ made object-oriented programming and data abstraction available to the community of software developers that until then had considered such techniques and the languages that supported them such as Smalltalk, Clu, Simula, Ada, OO Lisp dialects, etc., with disdain and even scorn: ''expensive toys unfit for real problems.'' C++ did three things to overcome

† As I write this, the chance of any significant additions to the C++ language by the standard committee is about zero. Many minor extensions was added, but I think that C++ is now a significantly more powerful, pleasent, and coherent language that is was when the standards process started. The major ''new'' extensions was run-time type information, namespaces, and the sum of many minor improvements to templates; see [Stroustrup,1994].

this formidable barrier:

[1] It produced code with run-time and space characteristics that competed head-on with the perceived leader in that field: C. Anything that matches or beats C *must* be fast enough. Anything that doesn't can and will – out of need or mere prejudice – be ignored.

[2] It allowed such code to be integrated into conventional systems and produced on traditional systems. A conventional degree of portability, the ability to coexist with existing code, and the ability to coexist with traditional tools, such as debuggers and editors, was essential.

[3] It allowed a gradual transition to these new programming techniques. It takes time to learn new techniques. Companies simply cannot afford to have significant numbers of programmers unproductive while they are learning. Nor can they afford the cost of failed projects caused by programmers poorly trained and inexperienced in the new techniques failing by overenthusiastically misapplying ideas.

In other words, C++ made object-oriented programming and data abstraction cheap and accessible.

In succeeding, C++ didn't just help ''itself'' and the C++ programmers. It also provided a major impetus to languages that provided different aspects of object-oriented programming and data abstraction. C++ isn't everything to all people and doesn't deliver on every promise ever made about some language or other. It does deliver on its own promises often enough to break down the wall of disbelief that stood in the way of all languages that allowed programmers to work at a higher level of abstraction.

7.2 Is C++ a Coherent Language?

C++ was successful in its own terms and is an effective vehicle for systems development, but is it a good language? Does C++ have an ecological niche now that the barriers of ignorance and prejudice against abstraction techniques have been broken?

Basically, I am happy with the language and quite a few users agree. There are many details I'd like to improve if I could, but the fundamental concept of a statically typed language using classes with virtual functions as the inheritance mechanism and facilities for low-level programming is sound.

7.2.1 What Should and Could Have Been Different?

Given a clean slate, what would be a better language than C++ for the things C++ is meant for? Consider the first order decisions: use of static type checking, clean separation between language and environment, no direct support for concurrency, ability to match the layout of objects and call sequences for languages such as C and Fortran, C compatibility.

Firstly, I considered and still consider static type checking essential both as a support for good design and secondarily for delivering acceptable run-time efficiency. Were I to design a new language for what C++ is used for today, I would again follow the Simula model of type checking and inheritance, *not* the Smalltalk or Lisp models. As I have said many times ''Had I wanted an imitation Smalltalk, I would have built a much better imitation. Smalltalk is the best Smalltalk around. If you want Smalltalk, use it [Stroustrup,1990].'' Having both static type checking and dynamic type identification (for example, in the form of virtual function calls) implies some difficult tradeoffs compared to language with only static or only dynamic type checking. The static type model and the dynamic type model cannot be identical and thus there will be some complexity and inelegance that can be avoided by supporting only one type model. However, I wouldn't want to write programs with only one model.

I also still consider a separation between the environment and the language essential. I do not want to use only one language, only one set of tools, and only one operating system. To offer a choice, separation is necessary. However, once the separation exists one can provide different

environments to suit different tastes and different requirements for supportiveness, resource consumption, and portability.

We never have a clean slate. Whatever new we do we must also make it possible for people to make a transition from old tools and ideas to new. Thus, if C hadn't been there for C++ to be almost compatible with then I would have chosen to be almost compatible with some other language.

Should a new language support garbage collection directly, say, like Modula-3 does? If so, could C++ have met its goals had it provided garbage collection? Garbage collection is great when you can afford it. Therefore, the option of having garbage collection is clearly desirable. However, garbage collection can be costly in terms of run time, real-time response, and porting effort (exactly how costly is the topic of much confused debate). Therefore, being forced to pay for garbage collection at all times isn't necessarily a blessing. C++ allows *optional* garbage collection [Ellis,1990], and I expect to see many experiments with garbage collecting C++ implementations in the near future. However, I am convinced (after reviewing the issue many times over the years) that had C++ depended on garbage collection it would have been stillborn.

Should a language have reference semantics for variables (that is a name is really a pointer to an object allocated elsewhere) like Smalltalk or Clu or true local variables like C and Pascal? This question relates to several issues such as co-existence with other languages, compatibility, and garbage collection. Simula dodged the question by having references to class objects (only) and true variables for objects of built-in types (only). Again, I consider it an open issue whether a language could be designed that provided the benefits of both references and true local variables without ugliness. Given a choice between elegance and the benefits of having both references and true local variables, I'll take the two kinds of variables.

7.2.2 What Should Have Been Left Out?

Even [Stroustrup,1980a] voiced concern that C with Classes might have become too large. I think ''a smaller language'' is number one on any wish list for C++; yet people deluge me and the standards committee with extension proposals. The fundamental reason for the size of C++ is that it supports more than one way of writing programs, more than one programming paradigm. From one point of view C++ is really three languages in one: A C-like language plus an Ada-like language, plus a Simula-like language, plus what it takes to integrate those features into a coherent whole.

Brian Kernighan observes that in C there is usually about one way of solving a given problem whereas, in C++ there are more. I conjecture that there typically are more than one way in C but that people don't see them. In C++, there are typically at least three alternatives and experienced people have quite a hard time not seeing them. There always is a design choice but in most languages the language designer has made the choice for you. For C++ I did not; the choice is yours. This is naturally abhorrent to people who believe that there is exactly one right way of doing things. It can also scare beginners and teachers that feel that a good language is one that you can completely understand in a week. C++ is not such a language. It was designed to provide a tool set for a professional and complaining that there are too many features is like the ''layman'' looking into an upholsterer's tool chest and exclaiming that there couldn't possibly be a need for all of those little hammers.

7.2.3 What Should Have Been Added?

As ever, the principle is to add as little as possible. A letter published on behalf of the extensions working group of the C++ standards committee puts it this way [Stroustrup,1992b]:

> ''First, let us try to dissuade you from proposing an extension to the C++ language. C++ is already too large and complicated for our taste and there are millions of lines of C++ code ''out there'' that we endeavor not to break. All changes to the language must undergo

tremendous consideration. Additions to it are undertaken with great trepidation. Wherever possible we prefer to see programming techniques and library functions used as alternatives to language extensions.

Many communities of programmers want to see their favorite language construct or library class propagated into C++. Unfortunately, adding useful features from diverse communities could turn C++ into a set of incoherent features. C++ is not perfect, but adding features could easily make it worse instead of better.''

So, given that, what features have caused trouble by their absence and which are under debate so that they might make it into C++ over the next couple of years? The feature I regret not having put in much earlier when it could be done without formal debate and experimentation was easy is some form of name space control†. C++ follows C in having a single global name space. C++ alleviates the problems that causes with class scopes and overloading mechanisms, but as programs grow and especially as independent libraries are developed and later used together in a single program the problem of name space clashes increases. My only defense is that I didn't like the resolution mechanisms I looked at, such as Ada packages and Modula-2 modules, because they had too great overlap with the C++ class mechanism and thus didn't fit.

In the original C++ design, I deliberately didn't include the Simula mechanisms for run-time type identification (`QUA` and `INSPECT`). My experience was that they were almost always misused, so that the benefits from having the mechanism would be outweighed by the disadvantages. Several proposals for providing some form of run-time type identification have arisen over the years and the focus is now on a proposal for dynamic casts – that is, type conversions that are checked at run time – by Dmitry Lenkov and me [Stroustrup,1992a]. Like other extension proposals it is going through extensive discussion and experimentation and is unlikely to be accepted in the form presented in that paper†.

7.3 What Was The Biggest Mistake?

To my mind there really is only one contender for the title of ''worst mistake.'' Release 1.0 and my first edition [Stroustrup,1986b] should have been delayed until a larger library including some simple classes such as singly and doubly linked lists, an associative array class, a range checked array class, and a simple string class could have been included. The absence of those led to everybody re-inventing the wheel and to an unnecessary diversity in the most fundamental classes. However, could I have done that? In a sense, I obviously could. The original plan for my book included three library chapters, one on the stream library, one on the container classes, and one on the task library, so I knew roughly what I wanted. Unfortunately, I was too tired and couldn't do container classes without some form of templates.

7.4 Hopes for the Future

May C++ serve its user community well. For that to happen, the language itself must be stable and well-specified. The C++ standards group and the C++ compiler vendors have a great responsibility here.

In addition to the language itself, we need libraries. Actually, we need a libraries industry to produce, distribute, maintain, and educate people. This is emerging. The challenge is to allow programs to be composed out of libraries from different vendors. This is hard and might need some support from the standards committee in the form of standard classes and mechanisms that ease the use of independently developed libraries†.

---

† A facility for defining and using name spaces was introduced into C++ in 1994; see [Stroustrup,1994].

† After extensive discussion and revision, a mechanism for run-time type information based on this proposal was accepted in 1993; see [Stroustrup,1994].

† The standard library now provides containers and algorithms serving this need.

The language itself plus the libraries define the language that a user de facto writes in. However, only through good understanding of the application areas and design techniques will the language and library features be put to good use. Thus there must be an emphasis on teaching people effective design techniques and good programming practices. Many of the techniques we need have still to be developed and many of the best techniques we do have still compete with plain ignorance and snake oil. I hope for far better textbooks for the C++ language and for programming and design techniques, and especially for textbooks that emphasize the connection between language features, good programming practices, and good design approaches.

Techniques, languages, and libraries must be supported by tools. The days of C++ programming supported by simply a ''naked'' compiler are almost over and the best C++ tools and environments are beginning to approach the power and convenience of the best tools and environments for any language. We can do much better. The best has yet to come.

8 Acknowledgements

It is clear that given the number of people who have contributed to C++ in some form or other over the last 12 years or so, I must have left most unmentioned. Some have been mentioned in this paper, more can be found in §1.2c of the ARM, and other acknowledgement sections of my books and papers. Here I'll just mention Doug McIlroy, Brian Kernighan, Andrew Koenig, and Jonathan Shopiro who have provided constant help, ideas, and encouragement to me and others for a decade.

I'd also like to mention the people who have done much of the hard, but usually unnoticed and unacknowledged work of developing and supporting Cfront through the years when we were always short of resources: Steve Dewhurst, Laura Eaves, Andrew Koenig, Stan Lippman, George Logothetis, Glen McClusky, Judy Ward, Nancy Wilkinson, and Barbara Moo who managed AT&T Bell Lab's C++ development and support group during the years when the work got done.

Also thanks to C++ compiler writers, library writers, etc. who sent me information about their compilers and tools – most of which went beyond what could be presented in a paper: Mike Ball, Walter Bright, Keith Gorlen, Steve Johnson, Kim Knuttilla, Archie Lachner, Doug Lea, Mark Linton, Aron Insinga, Doug Lea, Dmitri Lenkov, Jerry Schwarz, Michael Tiemann, and Mike Vilot.

Also thanks to people who read drafts of this paper and provided many constructive comments: Dag Brück, Steve Buroff, Peter Juhl, Brian Kernighan, Andrew Koenig, Stan Lippmann, Barbara Moo, Jerry Schwarz, and Jonathan Shopiro.

Most comments on the various versions of this paper were of the form ''This paper is too long ... please add information about X, Y, and Z ... also be more detailed about A, B, and C.'' I have tried to follow the first part of that advice, though the constraints of the second part have ensured that this did not become a short paper. Without Brian Kernighan's help this paper would have been much longer.

9 References

[Babcisky,1984] Karel Babcisky: *Simula Performance Assessment*. Proc. IFIP WG2.4 conference on System Implementation Languages: Experience and Assessment. Canterbury, Kent, UK. September 1984.

[Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.

[Booch,1991] Grady Booch: *Object-Oriented Design*. Benjamin Cummings. 1991. ISBN 0-8053-0091-0.

[Booch,1990] Grady Booch and Michael M. Vilot: *The Design of the C++ Booch*

Components. Proc. OOPSLA'90.

[Call,1987] Lisa A. Call, et al.: – *An Open System for Graphical User Interfaces*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Campbell,1987] Roy Campbell, et. al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Cargill,1991] Tom A. Cargill: *The Case Against Multiple Inheritance in C++*. USENIX Computer Systems Vol 4, no 1,1991.

[Carroll,1991] Martin Carroll: *Using Multiple Inheritance to Implement Abstract Data Types*. The C++ Report. April 1991.

[Cristian,1989] Flaviu Cristian: Exception Handling. in *Dependability of Resilient Computers*, T. Andersen Editor, BSP Professional Books, Blackwell Scientific Publications, 1989.

[Cox,1986] Brad Cox: *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley. Reading, Massachusetts. 1986.

[Dahl,1988] Ole-Johan Dahl: Personal communication.

[Ellis,1990] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-201-51459-1.

[Goldberg,1983] Adele Goldberg and David Robson: *Smalltalk-80, The language and its Implementation*. Addison-Wesley. 1983. ISBN 0-201-11371-6.

[Goodenough,1975] John Goodenough: *Exception Handling: Issues and a Proposed Notation*. CACM December 1975.

[Gorlen,1987] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Gorlen,1990] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: *Data Abstraction and Object-Oriented Programming in C++*. Wiley. West Sussex. England. 1990. ISBN 0-471-92346-X.

[Ichbiah,1979] Jean D. Ichbiah, et.al.: *Rationale for the Design of the ADA Programming Language*. Sigplan Notices Vol 14, no 6, June 1979 Part B.

[Johnson,1989] Ralph E. Johnson: *The Importance of Being Abstract*. The C++ Report, Vol 1 No 3, March 1989.

[Kernighan,1981] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall 1978. ISBN 0-13-110163-3.

[Kernighan,1981] Brian Kernighan: *Why Pascal is not my Favorite Programming Language*. AT&T Bell Labs Computer Science Technical Report No. 100. July 1981.

[Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall 1988. ISBN 0-13-110362-8.

[Kiczales,1992] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow: *The Art of the Metaobject Protocol*. The MIT Press. Cambridge, Massachusetts. 1991. ISBN 0-262-11158-6.

[Koenig,1988] Andrew Koenig: *Associative arrays in C++*. Proc. USENIX Conference. San Francisco. June 1988.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol.1 No.7. July 1989.

[Koenig,1989b] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++*. Proc. ''C++ at Work'' Conference. November 1989.

[Koenig,1990] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference, April 1990. Also, Journal of Object Oriented Programming, Vol.3 No.2. July/Aug 1990. pp. 16-33.

[Krogdahl,1984] Stein Krogdahl: *An Efficient Implementation of Simula Classes with*

Multiple Prefixing. Research Report No. 83 June 1984, University of Oslo, Institute of Informatics.

[Lenkov,1989] Dmitry Lenkov: *C++ Standardization Proposal*. #X3J11/89-016.

[Linton,1987] Mark A. Linton and Paul .R. Calder: *The Design and Implementation of InterViews*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Liskov,1987] Barbara Liskov: *Data Abstraction and Hierarchy*. Addendum to Proceedings of OOPSLA'87. October 1987.

[Meyer,1988] Bertrand Meyer: *Object-Oriented Software Construction*. Prentice-Hall. 1988. ISBN 0-13-629049.

[McCluskey,1992] Glen McCluskey: *An Environment for Template Instantiation*. The C++ Report. February 1992.

[Nelson,1991] Nelson, G. (editor): *Systems Programming with Modula-3*. Prentice-Hall. 1991. ISBN 0-13-590464-1.

[Rose,1984] Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in C++* Internal AT&T Bell Labs Technical Memorandum. January 1984. Reprinted in AT&T C++ Translator Release Notes November, 1985.

[Sakkinen,1992] Markku Sakkinen: *A Critique of the Inheritance Principles of C++*. USENIX Computer Systems, vol 5, no 1, Winter 1992.

[Sethi,1980] Ravi Sethi: A case study in specifying the semantics of a programming language. Seventh Annual ACM Symposium on Principles of Programming Languages. January 1980. pp 117-130.

[Shopiro,1985] Jonathan E. Shopiro: *Strings and lists for C++*. AT&T Bell Labs Internal Technical Memorandum. July 1985.

[Shopiro,1987] Jonathan E. Shopiro: Extending the C++ Task System for Real-Time Control. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[Snyder,1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Proc. OOPSLA'86. September 1986.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Techinical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup,1978] Bjarne Stroustrup: *On Unifying Module Interfaces*. ACM Operating Systems Review Vol.12 No.1. pp 90-98. January 1978.

[Stroustrup,1979a] Bjarne Stroustrup: *Communication and Control in Distributed Computer Systems*. Ph.D. thesis, Cambridge University, 1979.

[Stroustrup,1979b] Bjarne Stroustrup: *An Inter-Module Communication System for a Distributed Computer System*. Proc. 1st Int'l Conf. on Distributed Computing Systems. October 1979. pp 412-418.

[Stroustrup,1980a] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Also Sigplan Notices January, 1982.

[Stroustrup,1980b] Bjarne Stroustrup: *A Set of C Classes for Co-routine Style Programming*. Bell Laboratories Computer Science Technical Report CSTR-90. November 1980.

[Stroustrup,1981a] Bjarne Stroustrup: *Long Return: A technique for Improving The Efficiency of Inter-Module Communication*. Software Practice and Experience. January 1981. pp 131-143.

[Stroustrup,1981b] Bjarne Stroustrup: *Extensions of the C Language Type Concept* Bell Labs Internal Memorandum. January 1981.

[Stroustrup,1982] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution*. Bell Laboratories Computer Science internal document. April 1982.

Software Practice & Experience, Vol.13. 1983. pp. 139-61.

[Stroustrup,1984a] Bjarne Stroustrup: *The C++ Reference Manual*. AT&T Bell Labs Computer Science Technical Report No. 108 January 1984. (Written in the summer of 1983). Revised version November 1984.

[Stroustrup,1984b] Bjarne Stroustrup: *Data Abstraction in C*. Bell Labs technical Journal. Vol 63, No. 8. October 1984. pp 1701-1732. (Written in the summer of 1983)

[Stroustrup,1984c] Bjarne Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment September 1984.

[Stroustrup,1985] Bjarne Stroustrup: *An Extensible I/O Facility for C++* Proc. Summer 1985 USENIX Conference. June 1985. pp 57-70.

[Stroustrup,1986a] Bjarne Stroustrup *An Overview of C++*. ACM Sigplan Notices, Special Issue pp 7-18 October, 1986.

[Stroustrup,1986b] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1986. ISBN 0-201-12078-X.

[Stroustrup,1986c] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. pp 69-84. August, 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276, pp 51-70. Revised version in *IEEE Software Magazine*, May 1988, pp 10-20.

[Stroustrup,1987a] Bjarne Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference, May 1987. Also, USENIX Computer Systems, Vol 2 No 4, Fall 1989.

[Stroustrup,1987b] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C classes for Co-Routine Style Programming*. Proc. USENIX C++ conference, Santa Fe. November 1987. pp 417-439.

[Stroustrup,1987c] Bjarne Stroustrup: *The Evolution of C++: 1985-1987*. Proc. USENIX C++ conference, Santa Fe. November 1987. pp 1-22.

[Stroustrup,1988a] Bjarne Stroustrup: *Type-safe Linkage for C++*. USENIX Computer Systems, Vol.1 No.4. Fall 1988.

[Stroustrup,1988b] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. October 1988. pp. 1-18. Also, USENIX Computer Systems, Vol.2 No.1. Winter 1989.

[Stroustrup,1989a] Bjarne Stroustrup: *Standardizing C++*. The C++ Report. Vol.1 No.1. January 1989.

[Stroustrup,1989b] Bjarne Stroustrup: *The Evolution of C++: 1985-1989*. USENIX Computer Systems, Vol.2 No.3. Summer 1989. Revised version of [Stroustrup,1987c].

[Stroustrup,1990] Bjarne Stroustrup: *On Language Wars*. Hotline on Object-Oriented Technology. Vol 1, No 3. January 1990.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd edition)*. Addison-Wesley. 1991. ISBN 0-201-53992-6.

[Stroustrup,1992a] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992.

[Stroustrup,1992b] Bjarne Stroustrup: *How to Write a C++ Language Extension Proposal*. The C++ Report. May 1992.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[Taft,1992] S. Tucker Taft: *Ada 9X: A Technical Summary*. CACM. November 1992.

[Tiemann,1987] Michael Tiemann: *Wrappers* Proc. USENIX C++ conference, Santa Fe.

November 1987.

[Waldo,1991] Jim Waldo: *Controversy: The Case for Multiple Inheritance in C++*. USENIX Computer Systems, vol 4, no 2, Spring 1991.

[Wirth,1982] Niclaus Wirth: *Programming in Modula-2* Springer-Verlag. 1982.

View publication stats

Doc. no.: P0939R1
Date: 2018-10-08
Programming Language C++
Audience: All WG21
Reply to: Bjarne Stroustrup (bs@ms.com)

Direction for ISO C++

H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong

Revision History

- R0
  - This is the initial version.
- R1
  - Many clarifications, especially related to education (4.1), graphics (5), and error-handling (4.3)
  - Added a reference to the Vasa story to warn against featurism (3)

Main sections

- History
- Long-term Aims (decades)
- Medium-term Aims (3-10 years)
- Priorities for C++20
- Process Issues
- The C++ Programmer's Bill of Rights

Abstract

This is intended as a document updated as times go by and new facts and opinions emerge. It tries to articulate our aims for C++, the challenges faced, and some concrete suggestions for priorities.

This document differs from most papers by aiming for a global view and placing individual features in context, trying to avoid delving into technical details of individual proposals except where those "details" could affect the language as a whole or its tool environments.

To prevent this document from becoming toothless "pure philosophy," we propose giving priority to specific areas of development and to specific proposals in those areas.

Suggested improvements are welcome.

1. History

The Direction Group (DG) was created in response to calls for more thought about the direction of C++'s evolution (language and standard library) and comments about our self-established processes in WG21. The specific "call to action" was Operating principles for C++ [Winkel,2017] by several Heads of National Delegations.

We see C++ in danger of losing coherency due to proposals based on differing and sometimes mutually contradictory design philosophies and differing stylistic tastes. For that reason, we recommend that you (re)read [Winkel,2017] before proposing a new feature (language or library).

[Winkel,2017] quoted heavily from D&E. However, during the presentation of C++ Stability, Velocity, and Deployment Plans [Winters,2017] at the June 2017 Toronto meeting to an almost complete WG21, Titus Winters asked for a show of hands of who had read "The Design and Evolution of C++" [Stroustrup,1994]. Only about a quarter of the hands went up. Assuming that another quarter was shy or distracted, that indicates that half of the committee have never read the articulation of key design principles and decisions for C++. Part of the reason for that Is that copies of D&E can be hard to find. Consequently, we offered every member of the committee a free copy and ensured that Addison Wesley made D&E available again (in paper and electronic versions).

We strongly recommend that someone who wants to push or oppose a proposal read [Stroustrup,1994] and its related HOPL papers ([Stroustrup,1993] and [Stroustrup,2007]). Our impression is that many members of WG21 have an inaccurate view of how and why C++ succeeded and some are pushing popular approaches that in the past have led to failure (e.g., by languages following those approaches and losing out to C++). Obviously aims, techniques, and ideals change over decades, progress happens, but it is dangerous to operate in ignorance of past successes and failures.

2. The Direction Group

The direction group's (DG) charter is [Sutter,2018]:

> Direction group. The direction group is a small by-invitation group of experienced participants who are asked to recommend priorities for WG21. Currently, that group consists of: Howard Hinnant, Roger Orr, Bjarne Stroustrup, Daveed Vandevoorde, and Michael Wong. Their charter includes setting forth a group opinion on:
>
> - Evolution direction (language and library): This includes both language and library topics, and includes both proposals in hand and proposals we do not have but should solicit. The direction group maintains a list of the proposals it considers the most important for the next version of C++ or to otherwise make progress such as in a TS, and the design group chairs use that list to prioritize work at meetings. Typically, work on other topics will occur after there's nothing further left to do at this meeting to advance the listed items.

- Providing an opinion on any specific proposal: This includes whether the proposal should be pursued or not, and if pursued any changes that should be considered. Design group participants are strongly encouraged to give weight to an opinion that the direction group feel strongly enough about to suggest.

We are concerned with both the long-term and the shorter-term direction and handling of proposals, especially where they may intersect multiple WGs. When looking at direction and at individual proposals, we try to consider the interests of the larger C++ community, rather than just the narrower interests of WG21 members.

The direction group mailing list is direction@lists.isocpp.org. Opinions welcome.

We operate similar to a Board of Directors, with an annually rotating chairman. Bjarne lost a random number draw (written by Howard) and is therefore our chair for the first year:

- B. Stroustrup (2018)
- H. Hinnant (2019)
- M. Wong (2020)
- D. Vandevoorde (2021)
- R. Orr (2022)

3. Long-term Aims (decades)

Assuming we succeed, many of us will be writing and maintaining C++ in 10 years' time and 20 years' time. What kind of aims would we think reasonable for such a time scale? Obviously, long-term aims cannot change every year.

As a community, we want lots of things for C++, but

- Some we can't get because they are impossible.
- Some we can't get because they are incompatible with something else we have or want to have.
- Some we don't yet know how to do.
- Some we don't have the resources to do.
- Some we don't agree to be good for C++.
- Some we don't agree to be worthwhile for C++.

We understand that we can't have everything we want and that some of the things we want take a lot of time and work. Given that, we need to clarify our current long-term goals.

No language can be everything for everybody. Trying to achieve that has killed many efforts. We have to decide what C++ is supposed to be and to become.

What is C++?

- C++ is a language for defining and using light-weight abstractions
- C++ supports building resource constrained applications and software infrastructure
- C++ supports large-scale software development

How do we want C++ to develop?

- Improve support for large-scale dependable software
- Improve support for high-level concurrency models
- Simplify language use
- Address major sources of dissatisfaction
- Address major sources of errors

We fundamentally need:

- *Stability*: Useful code "lives" for decades
- *Evolution*: The world changes and C++ must change to face new challenges.

There is an inherent tension here. We should be very reluctant to break compatibility. People always want a simpler language, a language without complicating "legacy features", and get seriously angry if we break code that they depend on.

C++ is complicated, too complicated, yet we cannot remove significant facilities and changing them is very hard. Changing parts deemed insignificant can be risky (we need better analysis tools [Winter,2016]) and the potential gains would be insignificant. However, we badly need to simplify use of C++ and that leaves us with three alternatives:

- Provide simpler alternatives for simple uses
- Provide simplifying generalizations
- Provide alternatives to complicated and/or error-prone features

Often, a significant improvement involves a combination of those three.

Calling C++ a "multi-paradigm language" is no excuse for adding support for incompatible programming styles. If features don't smoothly interoperate, we get de facto dialects. Today, some of the most powerful design techniques combine aspects of traditional object-oriented programming, aspects of generic programming, aspects of functional programming, and some traditional imperative techniques. Such combinations, rather than theoretical purity, are the ideal.

- Provide features that are coherent in style (syntax and semantics) and style of use

This applies to libraries, to language features, and combinations of the two.

Technically, C++ rests on two pillars:

- A direct map to hardware (initially from C)

- Zero-overhead abstraction in production code (initially from Simula, where it wasn't zero-overhead)

Depart from those and the language is no longer C++. We should not depart from these principles and fall into the trap of:

- Abandoning the past which can seriously jeopardize compatibility. C++ is and will continue to be heavily used in long-lasting systems.
- Not addressing new challenges such as higher-level concurrency models. Should we fail, developers will need to switch to some other framework to gain the best performance.

Over the long term, we must strengthen those two pillars

- Better support for modern hardware (e.g., concurrency, GPUs, FPGAs, NUMA architectures, distributed systems, new memory systems)
- More expressive, simpler, and safer abstraction mechanisms (without added overhead)

Many useful abstractions should find their way into the standard library

- In principle, the C++ standard library can be implemented in C++ plus a few "intrinsic operations" for accessing low level machine facilities.

This is a variant of the "Don't leave room for a language below C++ (except assembler)" rule of thumb from D&E. Where we depart from this principle, we get long-term problems because the semantics easily drifts apart from the rest of the language.

C++ relies critically on static type safety for expressiveness, performance, and safety. The ideal is

- Complete type-safety and resource-safety (no memory corruption and no leaks)

This is achievable without added overhead, in particular without adding a garbage collector, and without restricting expressibility (see A brief introduction to C++'s model for type- and resource-safety [Stroustrup,2015b] and C++ Core Guidelines [CG,2015]). We don't consider a program littered with casts type-safe and we recognize that the ideal of complete type-safety and complete resource-safety cannot be achieved while accepting every legal C++ program (as we must for compatibility), so external tools will be needed (e.g., static analysis of lifetimes), but improved language and library support can help a lot.

C++ is now used in more domains than ever. We do not specifically recommend any specific domains, as every domain is important to somebody in the larger C++ community – even if their domain isn't well-represented in the committee. But we do intend to broaden our support for domains that are well-represented by the C++ community, even non-traditional ones, as a high-level aim. However here are some areas of general concerns – often cutting across application domains – that we should not ignore:

- *Safety and security*: reduce vulnerability to intrusion and the ability to exploit intrusions (e.g., prevent unsafe input and eliminate type violations from misuse of free store).
- *Simplification*: make simple things simple to do (and thereby making C++ easier to learn).

- *Interoperability*: improve interoperability with important languages and systems (e.g., Python bindings). Modern C++ facilitates that with Standard Layout types, variadic template, lambdas, and more.
- *Support for demanding applications*: offer domain-specific support for important application areas, such as medical, finance, automotive, and games (e.g., key libraries, such as JSON, flat containers, and pool and stack allocators).
- *Embedded systems*: make C++ easier to use and more effective for large and small embedded systems, (that does not mean just simplified C-like C++; e.g., see [Quora1] [Quora2] [Saks,2016] [Stroustrup,2018]). In particular, embedded systems programming could be supported by more specific libraries, emphasizing compactness of data and predictability of operations (e.g., "no free store allocation").
- Alternatives for error-prone and unsafe facilities (like **std::variant** as an alternative to unions or pattern matching as an alternative for **std::variant**).

This is not an exhaustive list, but demonstrates high level aims that brackets our priority. The medium-term aims will bracket our priorities further with specific proposals that are currently in flight. Many of these goals cannot be met through changes in the standard alone. Some, such as bindings and avoidance of error-prone facilities, require improvements beyond the standard, but the standard can make such improvements easier.

We would like to see the software development tools used for C++ (such as compilers, static analyzers, refactoring tools) significantly improved. Most of this is beyond the scope of WG21, but we should try hard not to make things more difficult, e.g. by significantly increasing compile times or the cost of tool building by adding significant complexity barriers (e.g., by encouraging coding styles increasing the use of macros or brittle SFINAE).

To maximize the likelihood of delivering significant improvements,

- We discourage isolated “cute” proposals.

There is a limit to what the community can absorb in a given period of time and while a large number of new features can excite C++ enthusiasts, their addition invariably leads to an impression of instability among many developers and among people observing the evolution of C++ from a distance. Also, the work on small isolated features distracts from the work on fundamental improvements (e.g., to the type system or the foundations of concurrency support), diverts resources, may complicate further improvements, and requires additions to references and teaching materials. In the years leading up to the 1998 standard, the committee members often reminded themselves of the story of the Vasa, the beautiful 17th century battleship that sank on its maiden voyage because of (among other things) insufficient work on its foundation and excessive late additions [Stroustrup,2018c].

4. Medium-term Aims (3-10 years)

Naturally, whatever we focus on in the medium term should reflect the aims, principles, and problems identified for the long term. What can we do to get closer to those ideals over the next few years? What are promising areas of development? What is urgent?

4.1. Learning and Teaching C++

C++ has a major problem with learning and teaching. This could send the C++ community into decline over the next few years. C++ is expert-friendly, but it cannot be just expert-friendly without losing important streams of new talent. This was the case before C++11 and C++11 reversed that negative trend. Currently, the C++ community is more active than it ever was, but this improvement must be sustained with solid improvements in the language and the standard library. It takes more still to sustain the enthusiasm, but here we are concerned with what WG21 can do. As individuals, we can encourage better tools and libraries. We suggest

- Establish a "Learning and Teaching" SG

We say "Learning and Teaching" because many (most?) new C++ programmers learn on their own. We must help them. Teachers (incl. professors) are often short of C++ experience, software development experience, and preparation time. We must help them. To paraphrase D&E: C++ needs to be like a weed, not like a garden rose, in order to flourish. We can't assume a reliable support system.

C++ is seriously under-represented in academia and often very poorly taught. It has been conventional to start teaching C++ by first introducing the lowest level and most error-prone facilities. Naturally, that discourages students and increases the time needed to get to what students consider meaningful computing (graphics, networking, mathematics, data analysis, etc.). Often, teachers even go to the extreme of insisting on using a C compiler. If the ultimate aim is to teach C++, that's like insisting people start learning English by reading Beowulf or the Canterbury Tales in their original early-English language versions. Those are great books, but Early English is incomprehensible to most native Modern-English speakers. In addition to the linguistic difficulties, such ancient sources present cultural conventions and idioms that seem very peculiar today. Instead of C, someone could teach Simula to prepare for learning C++. Why don't people do that? Because the historical approach to teaching language (natural or programming language) complicates and detracts from the end goal: good code.

Why do teachers use the C-first approach to teach C++? Part is tradition, curriculum inertia, and ignorance, but part of the reason is that C++ doesn't offer a smooth path to idiomatic, proper, modern use of C++. It is hard to bypass both the traps of low-level constructs and the complexities of advanced features and teach programming and proper C++ usage from the start.

C++ teaching is mostly stuck in a pre-graphics, pre-web world. This is a serious weakness, potentially fatal over the longer term. There are beginners' books trying to alleviate that (e.g., Programming -- Principles and Practice Using C++ (Second Edition) [Stroustrup,2014]) but they are a small minority and only partially successful. Teaching C++ to modern students will not be really effective until we can offer

- Simple standard graphics, simple interaction, and simple use of browsers (or GUI).
- Simple mechanisms for packaging, distribution, and installation for libraries and programs

The latter is beyond the current scope of WG21, but it is essential for the continuing success of C++; maybe the new Tools SG can help. If WG21 cannot do something official, maybe its members can help establish widely accepted de facto standards (note that languages and systems competing with C++ tend not to have formal standards).

What in the language and libraries helps learning and teaching?

- The ability to start teaching using only what is supplied by every implementation.

Students cannot be expected to download and install libraries on day #1. Even a requirement to **#include** an appropriate set of standard headers can be a burden. Thus, the technique of providing a training-wheels library in the form of a header or precompiled libraries is at most second best. This is a major technical reason that courses start with pointers, arrays, C-style strings, unions, etc., rather than **std::string**, **std::vector**, **std::variant**, and algorithms.

- The ability to sequence teaching from the simple and fundamental to the advanced.

The snag is that the simple and fundamental typically involves abstractions. This implies that

- It should be trivial to include what is needed for teaching; modules should help here and we need an “all” module and/or a “foundation” module (see also [Clow,2018]) so that novices don’t have to search a textbook or the web to find out which header to **#include** to use a foundational feature. Modules should help here.
- Errors coming from misuse of standard library facilities should be comprehensible; concepts should help here. All templates in the standard library intended for non-expert use should be “conceptualized.”
- There should be support for preventing use of non-approved and or yet-to-be-taught features; this cannot be standardized, but tools for catching counterproductive styles should be encouraged; the Core Guidelines [CG,2015] is a start.

To avoid such start-up problems, teachers sometimes restrict students to C as the simpler-to-start-using subset of C++. Unfortunately, this typically leads to unnecessary learning problems and delays the introduction of good C++ programming styles.

Students just starting to learn programming are not the only ones who need help with learning and initial use. They are most likely not even the majority of new C++ programmers. Other significant groups needing support are programmers coming to C++ from other languages and people who are professionals in some field other than software development (e.g., physicists or biologists), and just need to use C++ as a tool in their work. We recommend that the Teaching and Learning SG (should it be formed) consider:

- Help/support for programmers from non-C++ backgrounds, e.g., "C++ for X programmers" guides and specialized curricula.
- In the context of graphics support, consider what can be done to allow results of scientific computations, simulations, performance measures, and the like to be simply displayed

"Simply displayed" could be direct support for histograms (as in the very first 1980 C++ library), tables, and graphs in a standard-library and/or support for writing such data out in standard formats for display using other tools.

4.2. Library and Language

During the early stages of the development of the first standard, we articulated some principles for what to include in the standard library, including

- Language support
- Facilities that everybody needs
- Facilities needed for communicating among separately developed libraries

We think these are reasonable criteria, but in practice they didn't have much effect. People worked on and overelaborated individual components (e.g., **std::string**) that didn't interoperate effectively, showed major stylistic differences (reflecting differing design conventions), and often didn't completely cover what programmers considered a topic. The results were complaints, incomplete adoption, and wide use of commercial alternatives. We were rescued from disaster by Alex Stepanov. The STL was a coherent design that was sufficiently complete for its domain to become widely adopted.

If we can supply a feature as a library, we should do so because it is easier to validate a design through experimentation, the facility will be available to users earlier, and a library is usually easier to specify in isolation. However, a library design should not be an excuse for inelegant interfaces, for irregular interfaces, for stylistic differences from built-in language features, or overelaboration. It is always easy to add another function to a class, so library components have a tendency to bloat. Note

- the trouble we have had from **std::vector** using unsigned subscripts while built-in arrays use signed indices (unsigned arithmetic is modular, leading to correctness, performance, and error-detection problems)
- the dramatic differences in the interfaces to **std::any**, **std::optional**, and **std::variant**

At the time, there were of course reasons for those design choices that looked good to many, but we must aim for coherence:

- among standard-library features
- between the standard-library features and the built-in language features

Or, stated differently:

- No feature should be added without someone explicitly considering its interaction with other features (language and library).

We suggest that WGs (in particular, EWG and LEWG) delegate a "consistency review" to a small ad hoc subgroup for each feature or set of related features to be done concurrently with advanced work on technical details.

- We encourage a discussion of criteria of what should be in the standard library and what should not.

The aim is to be able to discuss proposed new standard-library components in the context of articulated criteria.

4.3. Concrete suggestions

Here are some more concrete suggestions for what can be done over the next few years to bring C++ closer to its long-term ideals:

- ***Pattern matching***: List comprehension, and not just for algebraic types. This could simplify code and strengthen type safety. It would eliminate a lot of ad-hoc selection, eliminate unsafe use of unions, low-level use of variants, and eliminate the visitor pattern (which is an expensive workaround that confuses many).

- ***Exception and error returns***: There are still people who argue against all use of exceptions and people who claim that exceptions should be used consistently instead of error codes. We'd like to see a clear articulation of where each alternative is ideal, but until we do that we should be careful about what error-handling strategies we support. To make progress, we need better data on costs and performance to evaluate the - often simplistic and narrowly focused - solutions suggested. Without better data and analysis, we risk hitting the N+1 problem ("the problem is that we can't manage N alternatives, so we add a new and better solution, thus having to deal with N+1 alternatives") [Stroustrup,2018c]. Contracts (A Contract Design [GDR,2016]) should simplify this by taking care of many nasty cases.

- ***Static reflection***: C++'s lack of run-time reflection is an essential strength in some areas (such as small embedded systems), but lack of simple reflection in specific areas have let many to switch to alternative languages. In particular, the lack of ability to generate serializer/deserializer pairs and object layout maps has been painful in many areas.

- ***Modern networking***: We need the networking as specified in the networking TS ([Wakely,2017]) urgently needed (on the C++20 timescale), but beyond that we need good simple support for interaction with modern browsers and networking (http, html5).

- ***Modern hardware***: We need better support for modern hardware, such as executors/execution context, affinity support in C++ leading to heterogeneous/distributed computing support, SIMD/task blocks, more concurrency data structures, improved atomics/memory model/lock-free data structures support. The challenge is to turn this (incomplete) laundry list into a coherent set of facilities and to introduce them in a manner that leaves each new standard with a coherent subset of our ideal.

- ***Simple graphics and interaction***: Command-line options and console I/O, while still useful in various contexts, haven't been the primary way of interacting with software for decades. Although modern interactive applications deal with broadly varying devices, almost all are controlled using an "event-driven" software model and most involve a graphical screen. We think the C++ standard library needs a component providing

    - a graphical canvas with basic geometric drawing, text rendering, and bitmap output primitives,

    - a simple input API for that graphical canvas abstract enough to deal with both "click" and "touch" events, and

    - a simple, highly-abstract "menus and dialogs" system (again, abstract enough to not force surfacing distinctions between devices as disparate as "phones" and "desktop computers").

Such a component will, of course, be "optional" to allow for platforms that don't have the needed I/O facilities. Practical uses of this component should be simple enough to be easily teachable to first-year students early in a course. Performance wouldn't be a prime concern for such a component as it would be competing with the likes of Python and Excel, rather than the excellent professional C++ graphics tools.

- ***Improved Unicode support***: for which SG16 was set up.
- ***Anything from the Priorities for C++20 that didn't make C++20***: see Priorities for C++20.

We feel that work on a database interface is badly needed, but since there seem to be no current work on that in the committee and no critical mass of expertise and interest, we can't put it on this list.

5. Priorities for C++20

We are very sympathetic to Ville Voutilainen's proposal To boldly suggest an overall plan for C++20 [Voutilainen,2017] to restrict new features to a limited set of significant proposals. Our list differs in detail:

- Concepts. Now partially in the WP.
- Modules (offering proper modularity and dramatic compile-time improvements)
- Ranges (incl., some of the infinite sequence extensions to the TS)
- Networking (Extensions for Networking [Wakely,2017])
- Concepts in the standard library

Beyond that, if time allows while giving priority to the top issues for committee time, we recommend:

- Simple Reflective Metaprogramming TS (relying on "ordinary C++" rather than template tricks).
- Contracts as specified in [GDR,2016], [GDR,2017]. Now in the WP.
- Calendar functions and time zones (Extending <chrono> to Calendars and Time Zones; [Hinnant,2017]). Now in the WP.
- Coroutines [Nishanov,2017].
- A high-level (and simple-to-use) API for common modern networking and browser use (HTML in particular and including Unicode support) TS (an example in current use: WT).
- A way to produce graphical output (probably initially a TS; e.g., see A Proposal to Add 2D Graphics Rendering and Display to C++ [McLaughin,2017]). Also consider the set of requirements submitted by Apple developers [Bastien,2018].

Additions beyond that should be discouraged as time sinks and distractions. Proposals for minor features should be given priority if and only if they support the priority items.

A significant part of the C++ community is panicked by fears of massive growth of features and complexity. This leads to disuse of novel features. The complexity of C++ and the difficulty of learning it are major barriers to increased use. No feature is cost free, there is always the implementation cost, the cost of producing teaching material, the time needed to learn, the opportunities for confusion, and the inevitable distraction from overselling.

We chose to prioritize these particular proposals because they

- relate directly to the long-term and medium-term aims
- are advanced with implementations and have user experience
- have people dedicated to their completion and improvement

When approaching these proposals try to make sure that a proposal

- actually offers the expected benefits
- handles the primary use cases simply and elegantly
- isn't delayed by a futile insistence of theoretical perfection
- isn't complicated to please every possible expert need

Once we know the approximate final goal, we should

- Use an incremental approach to design to benefit from actual experience

Our nightmares include a module system that doesn't offer modularity (e.g., where the order of import is significant or the compile-time cost of using modules approaches that of #includes) and a reflection system that is so integrated with general metaprogramming that nothing can be delivered until after C++23 and significantly increases compile-times. Note that the cost of compilation is among the loudest reasonable complaints about C++ from its users (modules should help).

6. Process Issues

We are "a bunch of volunteers."

- Most are enthusiasts for some aspect of the language or other.
- Few have a global view (geographically, C++ community, C++ usage, C++ Language and standard library).
- Most have a strong interest in only some subset of use, language, or library.
- Most are deeply engaged with a specific form of C++ as part of their daily work.
- Our levels and kinds of relevant computer-science, design, and programming language education vary dramatically.
- Many are clever people attracted to clever solutions.

- Some are devoted to ideas of perfection.
- Many are tool builders.
- Some have full control over their source code whereas others critically rely on code outside their direct control (open-source and/or commercial).
- Some operate in an environment with strong management control, others in environments without a formal management structure (and everything in-between).

This implies that we can't rely on a common vocabulary, a core set of shared values, a common set of basic ideals, or a common understanding of what's a problem. Consequently, we must spend more effort on

- articulating rationales for proposals.
- facilities for the "average programmer," who is seriously underrepresented on the committee.
- facilities aimed more at application builders than at builders of foundational libraries.

Please pay special attention to that last point. We feel that C++'s utility and reputation suffer badly from the committee lacking attention to improvements for relative novices and developers with relatively mundane requirements. Remember:

- Most C++ programmers are not like the members of the committee

We, as a committee, have no mechanism of reward (except accepting someone's proposal) or punishment (except delaying or rejecting someone's proposal). To get something accepted, we need consensus (defined as large majority, but not necessarily unanimity). This has implications on what we can do and how we do it.

- Nothing gets done unless someone cares enough to do it
- A small vocal minority can stop any proposal at any stage of the process.

Currently, C++ is probably as popular as it has ever been and there are definitely more active members of the standards committees than ever before. One effect of this enthusiasm for C++ is to inundate us with a flood of proposals. The sheer volume of proposals leads to fewer proposals getting through the processes to get accepted. Some proposals become warped from the need to gain support in an environment where time to think and present ideas is limited. Many of these proposals come from people who are unacquainted with standardization. We understand that some new members find it hard to accept that

- Progress is less rapid than for their corporate and open-source projects.
- Concerns that are essential to them are not given priority by the committee.
- Committee members don't all understand or accept their experience and design principles.
- Standardizing for decades differs dramatically from shipping the next release with the fewest number of bugs.

- There are millions of programmers who use C++ in ways that differ from what they consider normal and reasonable.
- "No bugs" does not imply that a proposal is good.
- "There are bugs" does not imply that a proposal is bad (there is a saying among mathematicians that the way to recognize an important result is by the number of errors in its initial proof – it is relatively simple to provide a flawless proof for a trivial result).
- "Good enough" isn't always good enough because a second look at the problem might come up with a better, more general, or better integrated solution. We have to consider the long term (decades).
- 90% or more of the work of getting a proposal into the standard is ensuring that it fits smoothly with other facilities (language and standard library).
- Stability/compatibility is an essential feature.

And still some improvements are urgent. However, in the context of the committee "urgent" still implies years of work/delay. Novices are not the only ones who are impatient, but we must try to channel our energies into constructive activities. We encourage members (old and new)

- to get acquainted with C++'s history and design rules
- to accept or contribute to our long-term aims for C++

The aim of most members, new or "vintage" is to improve C++ by having their favorite proposal accepted. However, the sheer number of proposals (constantly on the order of a hundred) and the number of papers (running at more than one hundred for each meeting), implies that not even every ***good*** proposal can be accepted. We encourage everyone to ask themselves

- Is my proposal among the top-20 in terms of long-term benefit to the C++ community as a whole?
- Is my proposal essential for some important application domain?

Push only very gently if the answers are "no," "sort of," or "I don't know" and redirect effort to proposals for which the answer is "Yes!" We encourage the WG chairs to

- Focus on the major high-level and intermediate-level goals
- Articulate the goals as they relate to their WG
- Prioritize proposals based on those goals
- Discourage proposals not in that scope
- Discourage proposals from being re-submitted with only minor changes after rejection (especially if the revised proposal does not include new insights into the problem to be solved).
- Encourage proposers to discuss potential disadvantages and costs of their proposals as well as their benefits. Every new feature comes with a significant cost.

All proposals consume the (limited) committee time and WG21 members should be considering the best *overall* outcome for the future of the language. Hence while small proposals to clean up non-trivial defects are welcome, discussion about these may have lower priority. If such a small proposal proves to be controversial it is probably better to withdraw, or defer, it to avoid preventing progress on more substantive items.

We are a set of interrelated committees currently with about 150 members present at a meeting and more active via the Web. Thus some "design by committee", or rather "design by committees," is unavoidable. We need to consciously and systematically try to minimize those effects by building a shared sense of direction.

- We have no shared aims, no shared taste.

This is a major problem, possibly the most dangerous problem we face as a committee. For C++ to succeed, we must overcome that. For starters, we – as individuals, as SGs, as WGs, and as the committee as a whole – must devote more time to develop and articulate common understanding.

- The alternative is a dysfunctional committee producing an incoherent language.

We need to be more explicit about

- What general problems we are trying to address
- How a particular proposal serves those articulated aims

If would also be a great improvement if members didn't spend all of their time in a single WG. Doing that leads to lack insight and of trust.

- Try to spend at least one day each meeting in a WG that isn't "your own"
- Try to read several papers from each mailing that is not aimed at "your WG"

Note that if "your" proposal progressed, you'll have to shepherd it through different WGs, so it is good to have some understanding of how they operate.

In addition, the committee as a whole should try to avoid that

- a single organization or cohesive group gain control of a WG.

WGs must be open to inputs from many sources.

As specified in the ISO and NB charters

- the operations of the committee must be transparent.

It is only natural that proposals are first developed and discussed among a small group of friends or colleagues. However, once a proposal reaches a larger group, discussions should be made public to all members. Furthermore, proposals should be made available as complete papers in plenty of time for the members to consider them before meetings. It is not acceptable to present only an incomplete summary a week or two before a meeting and expect a serious discussion, let alone a binding vote. Similarly, it is unreasonable to expect members to decide on a large detailed proposal with only a few

days warning; remember that a mailing before a meeting consists of about a hundred paper, including many long ones.

To maximize the likelihood of delivering significant improvements and to increase the predictability of our processes, we discourage

- "Change the World" papers for proposals already in flight

We change the language and standard library by gradually building on previous work or by providing a better alternative to an existing feature. Proposals to completely change direction for a proposal already in process should be treated with suspicion and subjected to at least as much scrutiny as the proposal already in flight has been. Looking at new proposals is more exciting than working out the obscure details of an old proposal, but eventually all successful proposals go through the "polish the last details" stage, so we can't escape that through new and shiny proposals, just postpone the pain. A proposal for a radical change to a proposal already ``in flight'' (i.e., in its second or later discussion) should not be allowed to delay the latter unless it comes with a paper with a detailed discussion of design, use, and implementation.

6.1. Trust

At the start of the formal standardization efforts (ANSI and ISO), P.J. Plauger and Tom Plum emphasized the need for trust in the process. The desired degree of trust is currently missing, and the consequences are dire.

A WG can spend meetings on a proposal, carefully crafting a compromise, balancing concerns, just to have the next WG spend several meetings reviewing the proposal, demanding changes, and rephrasing the wording with patchy understanding of the design rationale as it evolved over time. In particular

- large feature developments should be discussed in joint EWG+CWG, LEWG+LWG, and/or EWG+LEWG sessions as early as the overall design and the understanding of the underlying implementation alternatives allows.

Finally, after years of process, someone then stands up in full committee and raises issues that have been discussed for years stating "lack of comfort" with the proposal, suggesting alternative approaches, and demanding more time to consider or rejection. At this point, everybody unhappy with compromises made along the way chirps in with counter-points made over the years and the proposal is either withdrawn or defeated by a 20% minority, many of whom did not take part in previous discussions. We think that

- "lack of comfort" is not sufficient to block a proposal

Even when (as is common) there are minor remaining issues, typically, there are many months between a proposal being approved (by any part of the committee) and the final standard during which non-fatal flaws can be fixed. We think that a proposal that has passed a committee (say EWG or LEWG) should be accepted without undue delays by further groups. Obviously, new inputs should be considered, but in the absence of new information, "we considered and discussed that" by a WG chair should be conclusive:

- People who were not sufficiently motivated to take part in the discussion or conclusively defeated in WG should feel obliged to at least stay neutral.

Unfortunately, this principle is easily gamed by people who break it to block proposals they don't like. This has been observed in the committee and has led to resentment and erosion of trust.

At a first approximation,

- No proposal is ever perfect.
- Different people have very different ideas of what degree of perfection is needed for a proposal to be part of the standard (or part of a TS).

"The last bug" is a common programmer's joke and our language specification reads like a program in a poorly defined language (English) without advanced control structures, without abstraction mechanisms, and without a compiler. Consequently, perfection will always elude us and we should take that into account when we decide what is "good enough" for a standard (or a TS).

We have seen proposal move forward with an issues list to be considered before a final vote or even after. We strongly encourage that approach to make progress

- If a proposal is fundamentally sound, it should be moved forward, even into the WP text
- Refinement of text can (and should) happen later
- There are problems that are unlikely to be found until after the wording for a proposal is integrated into the WP text (as in software development: "integrate early to allow testing")
- Addition of desired but incompletely-developed features can wait until later, even to a later standard
- As the shipping date for a standard approaches, the efforts to "polish" the text and drain the issue lists should be redoubled at the expense of effort on new proposals (as is currently done)

Through delays, a good proposal can get dated, failing to benefit from years improvements and progress in the community. Note the file system TS and networking TSs each has a decade of use behind them. Similarly, **std::variant**, **std::optional**, and **std::any** have a long history as independent proposals. That wouldn't be too bad if the reason was that significant improvements were added during the process. However, the current process tends to reduce proposals to their most conservative cores. Thus, some "novel" C++ features feel dated by the time they are accepted.

- Aim for prompt delivery followed by incremental improvements

Note that this approach can only succeed if the end-goals are reasonably clear.

- Articulate the end goals for a proposal

When triaging features for consideration, we propose that the WG chairs to put a higher priority on features that conform to these goals, but also keep a watch against features that

- Turn C++ into a radically different language

- Turn parts of C++ into a much significantly different language by providing a segregated sub-language
- Have C++ compete with every other language by adding as many as possible of their features
- Incrementally modify C++ to support a whole new "paradigm" without articulating the end goal
- Hamper C++'s use for the most demanding systems programming tasks
- Increase the complexity of C++ use for the 99% for the benefit of the 1% (us and our best friends.)

The committee members and proposal authors are a group of volunteers who may have different aims, industry views, or company directives. As we propose features, we urge the group to maintain cordial discourse, and aim for what is best for C++ as a language and not merely what is best for your company's current direction. This sometimes means willing collaboration with people with different aims and priorities. For the most part, we have been successful. More specifically, don't oppose a proposal just because:

- it is seen as competing with your favorite proposal for time/resources
- it is not relevant to your current job
- you have not personally reviewed it
- it is not perfect (according to your principles)
- it is not coming from your friends
- it is coming from someone you have been at odds with on another subject

6.2. Proposal processing

WG21 does not lack for proposals. The interest is high and the number of proposals of each mailing has grown to more than 100. That's unmanageable for an individual with a day job. To stay focused on the priorities discussed in this document, we recommend that chairs focus at least 60% of their time on advancing these priorities. How that is done is entirely the chairs prerogative. For example:

- Raise the barrier for repeated presentation of and voting on failed proposals.
- Require a short tutorial for each proposal

For any non-trivial proposal, it is not obvious how to use the new facility well in combination with other features. It is easy to say "teaching this is easy" but such statements should be backed-up with examples and experience report (or be clearly marked as conjecture). In particular, for whom is learning and using a new feature easy?

- Never assume that the use of a proposed feature is easy and obvious to everybody.

The volume of proposals is such that people are not able to track all upcoming votes. This has led to "no" votes in WGs and plenary. We ask the committee to consider process improvements that improve this communication and notification, but also ask that members actively review and discuss with others

their proposal progress, so that the burden is fairly spread. At one time, Alistair Meredith did a great job of summarizing the flow and progress of each paper prior to C++11. Today this is unmanageable for one person. We have noted other forms of process improvement that has helped improved communication, and understanding the issues as we review papers. These include use of githubs, and summary notes recorded for each proposal, to save readers the bother of going through each meeting minutes to trace the history of a proposal and find the related papers. We do not necessarily enforce any particular methods, but feel some method should be considered to allow other groups to track progress:

- WGs and SGs should maintain an up-to-date brief list of the status of proposals being processed.
- WGs and SGs should - as far as possibly - announce what proposals will be up for discussion

Such lists should be easily accessible. It is not good enough to have information aimed at keeping people who are not regular members of a WG or SG posted on a hard-to-find mailing list requiring separate signup.

Remember the need for trust (§6.1):

- "I didn't have time to read the document" is not sufficient reason to oppose (provided the paper was submitted on time)
- "I don't understand" is not by itself sufficient reason to oppose (give reasons and examples why your lack of understanding is not just your own problem)

6.3. The role of TSs

Technical Specifications (TSs) have become popular. In theory, they provide an intermediate stage where major new features can be discussed, specified, tested under fewer constraints than for the standard itself, and later be moved into the standard with changes based on the experiences gained. The move from a TS to an IS is supposedly simplified by the significant work to complete the TS and the experience gained.

In practice, this seems to work reasonably well for libraries, though we see examples where facilities are stripped from a proposal as being too advanced, rather than pushed forward for experimentation. For language features, the experience is less positive. It seems that the barrier to entry into a TS is not significantly lower than for an IS, and the effort devoted to complete a TS detracts from experimentation, freezing the language feature in time, years before acceptance. After the TS, most of the design and specification issues are then revisited a second time. Thus, a TS becomes a method for delaying a proposal. Also, a TS doesn't seem to be sufficient to encourage multiple implementations. This implies that large parts of the C++ community doesn't get to use the facility, large organizations using multiple compilers can't experiment with the TS at scale, and tool builders hold back waiting for a "proper standard" supported by all major implementers. Consequently, detractors can dismiss the experience reports and clamor for novel alternatives.

We recommend

- Use TSs for library components.
- Don't use TSs for a language feature unless the feature is a mostly self-contained unit.

- Never use a TS simply to delay; it doesn't simplify later decision making.
- When proposing a TS, specify the "aim": what the TS is supposed to learn or achieve.
- List "exit criteria" (TS to IS or whatever target) to allow people to determine whether the work is complete and whether it succeeded.
- Consider other vehicles such as SG (Study Groups), IS, and not just TS
- Consider some or all the following incomplete list of frequently asked questions in your deliberations and TS proposal and record their answers along with the aim and exit criteria:
    - Is there an implementation?
    - Is it a Library or Language proposal, or involve both aspects?
    - Is the proposal a foundational proposal, meaning many other C++ aspects/proposal depend on it, and/or it depends on many other C++ aspects/proposals?
    - Is it independent of aspects of the language?
    - Are there competing design proposals?
    - Is the proposal complicated or large that you fear there will be error in design decision?
    - Is it a research idea?
    - Is there substantial invention?
    - Can it be staged?
    - Is there a subpart that deserves to be in IS
    - Is the wording complicated or unconventional?
    - Will the proposal benefit from early integration (can be applied to a WP)?
    - Will you get feedback/testing only after TS publication or IS publication?
    - Is there a motivation to slow down a proposal?
    - What would it take to turn the TS into an IS?
    - Are you juggling a large number of related or dependent proposals (other proposals that depend on this proposal)?
    - Are you aiming for user feedback?
    - Are you aiming for implementation feedback?
    - Is there a scheduling concern to make C++xx for it or its dependents?

We have found some or all these questions are always asked anyway at each SG deliberation, so proposers should consider them early on. Expect the DG to offer non-binding advisory (TS or IS or other target) in some cases and we hope you weigh our opinion as part of your decision process.

If a proposal isn't ready for the standard let it be improved or rejected in the appropriate WG or SG (Study Group). Don't add a formal TS process.

6.4. "Details" vs. Design

The committee spends most of its time on details that the average programmer will never notice. A lot of that is necessary. However, it seems that most members spend most of their time on such details during the design phase; that's wasteful and weakens the language as experienced by users. Instead, early on in the process, try to spend the majority of time on design issues, such as:

- What problems are this feature meant to solve? (D&E recommends not to accept a feature that doesn't solve two apparently unrelated problems)
- What alternative solutions are possible?
- What related features should be considered simultaneously with this one?
- How well does this feature fit with the general style of the language and standard library?
- Is this feature aimed only for a few language or library experts or will it be used in application code?

An individual proposal is essentially never useful in isolation; it will work in combination with other features:

- Consider every feature in the context of features with which it will be used
- Don't approve a set of related features one by one, evaluate them together

In particular, cluster related proposals (language and library) together.

- Be fair when comparing alternative proposals

In particular, try hard to give discussions of advantages and disadvantages equal weight for each alternative. Presenting only advantages for a "favored proposal" and only "disadvantages" for an unfavored alternative is not acceptable.

The text of the standard should be precise and comprehensive. It is not supposed to be a tutorial, but the ideal is that after some "acclimatization" an experienced programmer should be able to interpret the text with some confidence. This is not currently the case. When crafting WP text

- Consider readability by people not members of the WG crafting the text
- Non-normative notes can be useful
- Short examples can be very useful
- Try not to change the meaning of terms common in the C++ community

- Where possible, use terms common in the C++ community

Remember that compiler writers and standard-library implementers are not the only target audience for the standard (or a TS).

7. The C++ Programmers' Bill of Rights

This note was posted to reflectors, presented at the June 2017 Toronto meeting and discussed. We propose that it be formally adopted by WG21:

"The C++ Programmers' Bill of Rights."

We, the ISO C++ Standards Committee, promise to deliver the following to the best of our ability, assuming user code adheres to the current standard:

1. ***Compile-time stability***: Every significant change in behavior in a new version of the standard is detectable by a compiler for the previous version.
2. ***Link-time stability***: ABI breakage is avoided except in rare cases, which will be well documented and supported by a written rationale.
3. ***Compiler performance stability***: Changes will not imply significant added compile-time costs for existing code.
4. ***Run-time Performance stability***: Changes will not imply significant added run-time costs to existing code.
5. ***Progress***: Every revision of the standard will offer improved support for some significant programming activity or community.
6. ***Simplicity***: Every revision of the standard will offer simplification for some significant programming activity.
7. ***Timeliness***: Every revision of the standard will be shipped on time according to a published schedule.

Note "to the best of our abilities". These are ideals or guiding principles, rather than executable statements. For example, if a function is added to a header file, the compilation of code that includes that header will slow down imperceptibly. That's acceptable. Adding enormous amounts of code to a header so that compilation slows noticeably would be another matter.

Note that the "improved support" in "improved support for some significant programming activity or community" can be just about anything, such as a language feature, a standard-library, a hook for tool support, a generalization of an existing set of features. It is not our aim to destabilize the language with a demand of constant change; rather to help the committee focus on what is significant to the community as opposed to insignificant changes and churn.

These are ideals. They are what we would like to see done. If we succeed, most users will be very happy. However, they are not a recipe we could blindly follow to deliver a new standard. As is typical for ideals, they can conflict: there are tensions among the desired goals. This is common: Ideally, we want quality, on-time delivery, and low cost of products, but we know from experience that it is very hard to get all three. We want freedom of speech and absence of verbal intimidation, but balancing those two can be

very hard. It is the same for the ideals of the "The C++ Bill of Rights"; we want all, but the committee will have to make hard choices.

These are ideals. They are meant to be rather grand statements, rather than nit-picked long sentences watered down by adjectives and caveats. It will be up to the committee members to interpret the ideals in the light of real design and scheduling problems.

There are just seven ideals listed and they are quite general. We could add many more, but a laundry list of specifics would dull the appeal and be impossible to remember in the heat of discussions about direction, what can be delivered when, and specific technical concerns.

8. Caveat

The members of the direction group have been members of the standards committee for decades. We were (we think) chosen because we have deep and broad experience with the C++ technology, use, and community. We all have well documented track records. In the role of members of the direction group, we try to serve the good of the C++ community as a whole. It is not possible to completely separate what we consider best for the C++ community from the specific proposals we work on, but we try not to unfairly favor our own proposals. Similarly, we try not to discourage proposals or actions just based on personal biases. Some of our recommendations may not be universally appreciated, but nothing we say is meant to insult anyone.

Some of the recommendations here could be seen as contradictory. We see such as fundamental tensions in a design, requiring tradeoffs, rather than contradictions. Design is hard, design by committees is even harder.

9. Acknowledgements

Many of the arguments and points of view have a long history in the committee. Thanks to all who contributed. In particular, thanks to the authors of the documents we reference here.

10. References

- [Bastien,2018] J.F. Bastien: ????. P1225R0.
- [CG,2015] C++ Core Guidelines.
- [Clow,2018] Marshall Clow, Beman Dawes, Gabriel Dos Reis, Stephan T. Lavavej, Billy O'Neal, Bjarne Stroustrup, and Jonathan Wakely: Standard Library Modules. P0581R1. 2018-2-6.
- [GDR,2016] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: A Contract Design . P0380R1. 2016-07-11.
- [GDR,2017] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup: Support for contract based programming in C++. P0542R2. 2017-11-26.

- [Nishanov,2017] Gor Nishanov: C++ Extensions for Coroutines TS Document . N4680. 2017-07-30.
- [Hinnant,2017] Howard Hinnant: Extending <chrono> to Calendars and Time Zones. P0355R4. 2017-10-1.
- [McLaughin,2017] Michael McLaughlin, Herb Sutter, Jason Zink, Guy Davidson: A Proposal to Add 2D Graphics Rendering and Display to C++. P0267r6. 2017-07-30.
- [Quora1] Can I use C++ for hard real-time embedded systems and kernel development?
- [Quora2] Does C-11 and C-14 make it easy to use -C++ in kernel development?
- [Saks,2016] Dan Saks "extern c: Talking to C Programmers about C++". CppCon 2016 keynote.
- [Stroustrup,1994] B. Stroustrup: The Design and Evolution of C++. Addison-Wesley. 1994.
- [Stroustrup,1993] B. Stroustrup: A History of C++: 1979-1991. Proc. ACM HOPL-2. 1993.
- [Stroustrup,2007] B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. Proc. ACM HOPL-III. 2007.
- [Stroustrup,2014] B. Stroustrup: Programming -- Principles and Practice Using C++ (Second Edition). May 2014. Addison-Wesley. ISBN 978-0321992789. 1st edition 2008.
- [Stroustrup,2015] B. Stroustrup: Thoughts about C++17. N4492. 2015-05-15.
- [Stroustrup,2015b] B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. Revised December 2015.
- [Stroustrup,2018] B. Stroustrup: NDC Keynote: What can C++ do for embedded systems developers?
- [Stroustrup,2018b] B. Stroustrup: Remember the Vasa! P0997R0.
- [Stroustrup,2018c] B. Stroustrup: The Evils of Paradigms. P0176R0.
- [Sutter,2018] WG21 Practices and Procedures. ISO/IEC JTC1/SC22/WG21/SD-4. 2018-01-17.
- [Winkel,2017] JC van Winkel, Jose Daniel Garcia, Ville Voutilainen, Roger Orr, Michael Wong, and Sylvain Bonnal: Operating principles for C++. P0559R0. 2017-01-31.
- [Voutilainen,2017] Ville Voutilainen: To boldly suggest an overall plan for C++20. P0592R0. 2017-02-05.
- [Wakely,2017] Jonathan Wakely: Extensions for Networking. N471. 2017-11-27.
- [Winters,2017] Titus Winters, Bjarne Stroustrup, Daveed Vandevoorde, Beman Dawes, Michael Wong, and Howard Hinnant: C++ Stability, Velocity, and Deployment Plans. P0684R0 2017-06-19.

Doc. no.: P0977r0
Date: 2018-03-6
Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (bs@ms.com)

Remember the Vasa!

Bjarne Stroustrup

Many/most people in WG21 are working independently towards non-shared goals. Individually, many (most?) proposals make sense. Together they are insanity to the point of endangering the future of C++.

Here is a selection of papers from the pre-Jacksonville mailing. I list papers that I think has the potential for significantly change the way we write code, so that each has significant implications on teaching, maintenance, and coding guidelines. Many also have implications for implementations:

1. *Gabriel Dos Reis: Modules TS
2. Herb Sutter: Concepts in-place syntax
3. H. Carter Edwards: Polymorphic Multidimensional Array Reference
4. H. Carter Edwards: Relaxed Incomplete Multidimensional Array Type Declaration
5. H. Carter Edwards: Span - foundation for the future
6. David S. Hollman: An Ontology for Properties of mdspan
7. *Neil MacIntosh: span: bounds-safe views for sequences of objects
8. Vicente J. Botet Escribá: C++ generic overload function (Revision 3)
9. Matúš Chochlík: Static reflection
10. Matúš Chochlík: Static reflection of functions
11. Jonathan Coe: A polymorphic value-type for C++
12. Matthias Kretz: Data-Parallel Vector Types & Operations
13. Vicente Botet: std::expected
14. Tom Honermann : char8_t: A type for UTF-8 characters and strings (Revision 1)
15. *J. D. Garcia: Support for contract based programming in C++
16. Victor Zverovich: Text Formatting
17. Vicente J. Botet Escribá: C++ Monadic interface
18. Axel Naumann: Parametric Functions
19. Bryce Adelstein Lelbach: I Back to the std2::future Part I
20. Herb Sutter : Metaclasses: Generative C++

21. Jeff Snyder: Class Types in Non-Type Template Parameters
22. Vicente J. Botet Escribá: ValuedOrError and ValueOrNone types
23. John McFarlane: Elastic Integers
24. Gašper Ažman: Deducing this
25. Corentin jabot: A plea for a consistent, terse and intuitive declaration syntax
26. *Bruno Cardoso Lopes*: A proposal for modular macros
27. Barry Revzin: Chaining Comparisons
28. Lee Howes: A strawman Future API
29. Tomasz Kamiński : Symmetry for spaceship
30. *Gor Nishanov: merging Coroutines TS [N4723] into the C++20 working draft
31. Vittorio Romeo: Concept-constrained auto
32. Titus Winters: LEWG wishlist for EWG
33. Nathan Sidwell: Modules:Dependent ADL
34. Nathan Sidwell: Modules:Unqualified Using Declarations
35. James Dennett: Towards A (Lazy) Forwarding Mechanism for C++
36. *Herb Sutter: A Modest Proposal: Fixing ADL
37. Richard Smith: Towards consistency between <=> and other comparison operators
38. Richard Smith: Another take on Modules
39. Peter Dimov: Adding support for type-based metaprogramming to the standard library
40. Matúš Chochlík: constexpr reflexpr
41. Mingxin Wang: PFA: A Generic, Extendable and Efficient Solution for Polymorphic Programming
42. Ville Voutilainen: Allow initializing aggregates from a parenthesized list of values
43. Richard Smith: P0936R0 Bind Returned/Initialized Objects to Lifetime of Parameters

To avoid this list looking longer and even scarier, I have listed only first authors.

For truth in advertising, I have marked proposals I support with *. I may support parts of other proposals here, but not all or exactly as written. There are also proposals that I'm working on not present here, such as operator dot, uniform function call, and **stack_array**, see Thoughts about C++17. Since I'm not bringing those forward in Jacksonville, they don't (yet) belong on this list.

Apologies if I forgot a paper with major implications or if I listed one without a potential significant impact. This list contains less than half of the proposals for extensions, changes, and improvements currently being discussed. They are merely the most prominent. Proposals related to concurrency are underrepresented.

Hardly any paper contains extensive discussions of the proposed feature's effect in combination with other new features, existing features, and libraries in "ordinary code" written by "ordinary programmers." Few present details of experience of use or teaching. Hardly any contain a serious discussion of objections raised. Every proposal is subject to the law of unexpected consequences: There will be unexpected consequences.

We are on a path to disaster though enthusiasm and design-by-committee (or rather "design-by-committees"). During the early days of WG21 the story of the Vasa was popular as warning against overelaboration (from 1992):

> Please also understand that there are dozens of reasonable extensions and changes being proposed. If every extension that is reasonably well-defined, clean and general, and would make life easier for a couple of hundred or couple of thousand C++ programmers were accepted, the language would more than double in size. We do not think this would be an advantage to the C++ community.
>
> We often remind ourselves of the good ship Vasa. It was to be the pride of the Swedish navy and was built to be the biggest and most beautiful battleship ever. Unfortunately, to accommodate enough statues and guns it underwent major redesigns and extension during construction. The result was that it only made it half way across Stockholm harbor before a gust of wind blew it over and it sank killing about 50 people. It has been raised and you can now see it in a museum in Stockholm. It is a beauty to behold - far more beautiful at the time than its unextended first design and far more beautiful today than if it had suffered the usual fate of a 17th century battle ship -- but that is no consolation to its designer, builders, and intended users.

Remember the Vasa!

There are people who concluded from the Vasa story that all incremental improvement is a bad strategy. However, if the Vasa had been sent to sea as originally designed, it could not have served its purpose. Being under-gunned, someone would have sent it to the bottom full of holes. Being somewhat ordinary, it would have failed in its representative (image) role. Recent research has shown that a relatively modest increase of the Vasa's length and breadth (claimed technically feasible) would have made it stable, so my reading of the Vasa story is: Work hard on a solid foundation, learn from experience, and don't scrimp on the testing.

The foundation begun in C++11 is not yet complete, and C++17 did little to make our foundation more solid, regular, and complete. Instead, it added significant surface complexity and increased the number of features people need to learn. C++ could crumble under the weight of these – mostly not quite fully-baked – proposals. We should not spend most our time creating increasingly complicated facilities for experts, such as ourselves.

We need a reasonably coherent language that can be used by "ordinary programmers" whose main concern is to ship great applications on time. We now have about 150 cooks; that's not a good way to get a tasty and balanced meal.

We are on the path to something that could destroy C++. We must get off that path!

See (not a feature proposals):

- Ville Voutilainen: To boldly suggest an overall plan for C++20
- B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong: Direction for ISO C++

A Tour of C++

Bjarne Stroustrup

C++ In-Depth Series • Bjarne Stroustrup

A Tour of C++

The C++ In-Depth Series

BJARNE STROUSTRUP, Editor

"I have made this letter longer than usual, because I lack the time to make it short."
— Blaise Pascal

The C++ In-Depth Series is a collection of concise and focused books providing real-world programmers with reliable information about the C++ programming language. Selected by the designer and original implementer of C++, Bjarne Stroustrup, and written by experts in the field, each book in this series presents either a single topic, at a technical level appropriate to that topic, or a fast-paced overview, for a quick understanding of broader language features. Its practical approach, in either case, is designed to lift professionals (and aspiring professionals) to the next level of programming skill or knowledge.

These short books are meant to be read and referenced without the distraction of unrelated material. As C++ matures, it becomes increasingly important to be able to separate essential information from hype and glitz, and to find the deep content and practical guidance needed for continued development. The C++ In-Depth Series provides the background, tools, concepts, techniques, and new approaches that can enable this development, and thereby give readers a valuable, critical edge.

A Tour of C++

Bjarne Stroustrup

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:
International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Stroustrup, Bjarne.
A Tour of C++ / Bjarne Stroustrup.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-958310 (pbk. : alk. paper)—ISBN 0-321-958314 (pbk. : alk. paper)
1. C++ (Computer programming language) I. Title.

QA76.73.C153 S77 2013
005.13’3—dc23 2013002159

Copyright © 2014 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-321-958310
ISBN-10: 0-321-958314
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, September 2013

Contents

Contents v

Preface ix

1 The Basics 1

1.1 Introduction 1
1.2 Programs 1
1.3 Hello, World! 2
1.4 Functions 3
1.5 Types, Variables, and Arithmetic 5
1.6 Scope 8
1.7 Constants 8
1.8 Pointers, Arrays, and References 9
1.9 Tests 12
1.10 Advice 14

2 User-Defined Types 15

2.1 Introduction 15
2.2 Structures 16
2.3 Classes 17
2.4 Unions 19
2.5 Enumerations 20
2.6 Advice 21

3 Modularity **23**

3.1 Introduction ... 23
3.2 Separate Compilation ... 24
3.3 Namespaces ... 26
3.4 Error Handling ... 27
3.5 Advice ... 31

4 Classes **33**

4.1 Introduction ... 33
4.2 Concrete Types ... 34
4.3 Abstract Types ... 39
4.4 Virtual Functions ... 42
4.5 Class Hierarchies ... 42
4.6 Copy and Move ... 48
4.7 Advice ... 56

5 Templates **59**

5.1 Introduction ... 59
5.2 Parameterized Types ... 59
5.3 Function Templates ... 62
5.4 Concepts and Generic Programming ... 62
5.5 Function Objects ... 64
5.6 Variadic Templates ... 66
5.7 Aliases ... 67
5.8 Template Compilation Model ... 68
5.9 Advice ... 68

6 Library Overview **71**

6.1 Introduction ... 71
6.2 Standard-Library Components ... 72
6.3 Standard-Library Headers and Namespace ... 72
6.4 Advice ... 74

7 Strings and Regular Expressions **75**

7.1 Introduction ... 75
7.2 Strings ... 75
7.3 Regular Expressions ... 78
7.4 Advice ... 84

8 I/O Streams **85**

8.1 Introduction 85
8.2 Output 86
8.3 Input 87
8.4 I/O State 89
8.5 I/O of User-Defined Types 90
8.6 Formatting 91
8.7 File Streams 92
8.8 String Streams 92
8.9 Advice 93

9 Containers **95**

9.1 Introduction 95
9.2 **vector** 96
9.3 **list** 100
9.4 **map** 101
9.5 **unordered_map** 102
9.6 Container Overview 103
9.7 Advice 104

10 Algorithms **107**

10.1 Introduction 107
10.2 Use of Iterators 108
10.3 Iterator Types 111
10.4 Stream Iterators 112
10.5 Predicates 113
10.6 Algorithm Overview 114
10.7 Container Algorithms 115
10.8 Advice 115

11 Utilities **117**

11.1 Introduction 117
11.2 Resource Management 117
11.3 Specialized Containers 121
11.4 Time 125
11.5 Function Adaptors 125
11.6 Type Functions 128
11.7 Advice 131

12 Numerics **133**

12.1 Introduction 133
12.2 Mathematical Functions 134
12.3 Numerical Algorithms 135
12.4 Complex Numbers 135
12.5 Random Numbers 136
12.6 Vector Arithmetic 138
12.7 Numeric Limits 138
12.8 Advice 138

13 Concurrency **141**

13.1 Introduction 141
13.2 Tasks and **thread**s 142
13.3 Passing Arguments 143
13.4 Returning Results 144
13.5 Sharing Data 144
13.6 Waiting for Events 146
13.7 Communicating Tasks 147
13.8 Advice 151

14 History and Compatibility **153**

14.1 History 153
14.2 C++11 Extensions 158
14.3 C/C++ Compatibility 161
14.4 Bibliography 166
14.5 Advice 168

Index **171**

Preface

When you wish to instruct,
be brief.
– Cicero

C++ feels like a new language. That is, I can express my ideas more clearly, more simply, and more directly in C++11 than I could in C++98. Furthermore, the resulting programs are better checked by the compiler and run faster.

Like other modern languages, C++ is large and there are a large number of libraries needed for effective use. This thin book aims to give an experienced programmer an idea of what constitutes modern C++. It covers most major language features and the major standard-library components. This book can be read in just a few hours but, obviously, there is much more to writing good C++ than can be learned in a day. However, the aim here is not mastery, but to give an overview, to give key examples, and to help a programmer get started. For mastery, consider my *The C++ Programming Language, Fourth Edition* (TC++PL4) [Stroustrup,2013]. In fact, this book is an extended version of the material that constitutes Chapters 2-5 of TC++PL4, also entitled *A Tour of C++*. I have added extensions and improvements to make this book reasonably self-contained. The structure of this tour follows that of TC++PL4, so it is easy to find supplementary material. Similarly, the exercises for TC++PL4 that are available on my Web site (www.stroustrup.com) can be used to support this tour.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++* [Stroustrup,2009], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here.

As an analogy, think of a short sightseeing tour of a city, such as Copenhagen or New York. In just a few hours, you are given a quick peek at the major attractions, told a few background stories, and usually given some suggestions about what to see next. You do *not* know the city after such a tour. You do *not* understand all you have seen and heard. You do *not* know how to navigate the formal and informal rules that govern life in the city. To really know a city, you have to live in it,

often for years. However, with a bit of luck, you will have gained a bit of an overview, a notion of what is special about the city, and ideas of what might be of interest to you. After the tour, the real exploration can begin.

This tour presents the major C++ language features as they support programming styles, such as object-oriented and generic programming. It does not attempt to provide a detailed, reference-manual, feature-by-feature view of the language. Similarly, it presents the standard libraries in terms of examples, rather than exhaustively. It does not describe libraries beyond those defined by the ISO standard. The reader can search out supporting material as needed. [Stroustrup,2009] and [Stroustrup,2012] are examples of such material, but there is an enormous amount of material (of varying quality) available on the Web. For example, when I mention a standard library function or class, its definition can easily be looked up, and by examining the documentation of its header (also easily accessible on the Web), many related facilities can be found.

This tour presents C++ as an integrated whole, rather than as a layer cake. Consequently, it does not identify language features as present in C, part of C++98, or new in C++11. Such information can be found in Chapter 14 (History and Compatibility).

Acknowledgments
Much of the material presented here is borrowed from TC++PL4 [Stroustrup,2012], so thanks to all who helped completing that book. Also, thanks to my editor at Addison-Wesley, Peter Gordon, who first suggested that the four Tour chapters from TC++PL4 might be expanded into a reasonably self-contained and consistent publication of their own.

College Station, Texas *Bjarne Stroustrup*

1

The Basics

The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

- Introduction
- Programs
- Hello, World!
- Functions
- Types, Variables, and Arithmetic
- Scope and Lifetime
- Constants
- Pointers, Arrays, and References
- Tests
- Advice

1.1 Introduction

This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called *procedural programming*.

1.2 Programs

C++ is a compiled language. For a program to run, its source text has to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program. A C++ program typically consists of many source code files (usually simply called *source files*).

An executable program is created for a specific hardware/system combination; it is not portable, say, from a Mac to a Windows PC. When we talk about portability of C++ programs, we usually mean portability of source code; that is, the source code can be successfully compiled and run on a variety of systems.

The ISO C++ standard defines two kinds of entities:

- *Core language features*, such as built-in types (e.g., **char** and **int**) and loops (e.g., **for**-statements and **while**-statements)
- *Standard-library components*, such as containers (e.g., **vector** and **map**) and I/O operations (e.g., **<<** and **getline()**)

The standard-library components are perfectly ordinary C++ code provided by every C++ implementation. That is, the C++ standard library can be implemented in C++ itself (and is with very minor uses of machine code for things such as thread context switching). This implies that C++ is sufficiently expressive and efficient for the most demanding systems programming tasks.

C++ is a statically typed language. That is, the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use. The type of an object determines the set of operations applicable to it.

1.3 Hello, World!

The minimal C++ program is

```
int main() { }          // the minimal C++ program
```

This defines a function called **main**, which takes no arguments and does nothing.

Curly braces, **{ }**, express grouping in C++. Here, they indicate the start and end of the function body. The double slash, **//**, begins a comment that extends to the end of the line. A comment is for the human reader; the compiler ignores comments.

Every C++ program must have exactly one global function named **main()**. The program starts by executing that function. The **int** value returned by **main()**, if any, is the program's return value to "the system." If no value is returned, the system will receive a value indicating successful completion. A nonzero value from **main()** indicates failure. Not every operating system and execution environment make use of that return value: Linux/Unix-based environments often do, but Windows-based environments rarely do.

Typically, a program produces some output. Here is a program that writes **Hello, World!**:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

The line `#include <iostream>` instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in `iostream`. Without these declarations, the expression

```
std::cout << "Hello, World!\n"
```

would make no sense. The operator `<<` ("put to") writes its second argument onto its first. In this case, the string literal `"Hello, World!\n"` is written onto the standard output stream `std::cout`. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character `\` followed by another character denotes a single "special character." In this case, `\n` is the newline character, so that the characters written are `Hello, World!` followed by a newline.

The `std::` specifies that the name `cout` is to be found in the standard-library namespace (§3.3). I usually leave out the `std::` when discussing standard features; §3.3 shows how to make names from a namespace visible without explicit qualification.

Essentially all executable code is placed in functions and called directly or indirectly from `main()`. For example:

```
#include <iostream>          // include ("import") the declarations for the I/O stream library

using namespace std;         // make names from std visible without std:: (§3.3)

double square(double x)      // square a double precision floating-point number
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
    print_square(1.234);     // print: the square of 1.234 is 1.52276
}
```

A "return type" `void` indicates that a function does not return a value.

1.4 Functions

The main way of getting something done in a C++ program is to call a function to do it. Defining a function is the way you specify how an operation is to be done. A function cannot be called unless it has been previously declared.

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call. For example:

```
Elem* next_elem();      // no argument; return a pointer to Elem (an Elem*)
void exit(int);         // int argument; return nothing
double sqrt(double);    // double argument; return a double
```

In a function declaration, the return type comes before the name of the function and the argument types after the name enclosed in parentheses.

The semantics of argument passing are identical to the semantics of copy initialization. That is, argument types are checked and implicit argument type conversion takes place when necessary (§1.5). For example:

```
double s2 = sqrt(2);          // call sqrt() with the argument double{2}
double s3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

The value of such compile-time checking and type conversion should not be underestimated.

A function declaration may contain argument names. This can be a help to the reader of a program, but unless the declaration is also a function definition, the compiler simply ignores such names. For example:

```
double sqrt(double d);    // return the square root of d
double square(double);    // return the square of the argument
```

The type of a function consists of the return type and the argument types. For class member functions (§2.3, §4.2.1), the name of the class is also part of the function type. For example:

```
double get(const vector<double>& vec, int index);    // type: double(const vector<double>&,int)
char& String::operator[](int index);                 // type: char& String::(int)
```

We want our code to be comprehensible, because that is the first step on the way to maintainability. The first step to comprehensibility is to break computational tasks into comprehensible chunks (represented as functions and classes) and name those. Such functions then provide the basic vocabulary of computation, just as the types (built-in and user-defined) provide the basic vocabulary of data. The C++ standard algorithms (e.g., **find**, **sort**, and **iota**) provide a good start (Chapter 10). Next, we can compose functions representing common or specialized tasks into larger computations.

The number of errors in code correlates strongly with the amount of code and the complexity of the code. Both problems can be addressed by using more and shorter functions. Using a function to do a specific task often saves us from writing a specific piece of code in the middle of other code; making it a function forces us to name the activity and document its dependencies.

If two functions are defined with the same name, but with different argument types, the compiler will choose the most appropriate function to invoke for each call. For example:

```
void print(int);       // takes an integer argument
void print(double);    // takes a floating-point argument
void print(string);    // takes a string argument

void user()
{
    print(42);                  // calls print(int)
    print(9.65);                // calls print(double)
    print("D is for Digital");  // calls print(string)
}
```

If two alternative functions could be called, but neither is better than the other, the call is deemed ambiguous and the compiler gives an error. For example:

```
void print(int,double);
void print(double,int);

void user2()
{
        print(0,0);        // error: ambiguous
}
```

This is known as function overloading and is one of the essential parts of generic programming (§5.4). When a function is overloaded, each function of the same name should implement the same semantics. The **print()** functions are an example of this; each **print()** prints its argument.

1.5 Types, Variables, and Arithmetic

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that **inch** is of type **int**; that is, **inch** is an integer variable.

A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.

C++ offers a variety of fundamental types. For example:

```
bool        // Boolean, possible values are true and false
char        // character, for example, 'a', 'z', and '9'
int         // integer, for example, -273, 42, and 1066
double      // double-precision floating-point number, for example, -273.15, 3.14, and 299793.0
unsigned    // non-negative integer, for example, 0, 1, and 999
```

Each fundamental type corresponds directly to hardware facilities and has a fixed size that determines the range of values that can be stored in it:

A **char** variable is of the natural size to hold a character on a given machine (typically an 8-bit byte), and the sizes of other types are quoted in multiples of the size of a **char**. The size of a type is implementation-defined (i.e., it can vary among different machines) and can be obtained by the

sizeof operator; for example, **sizeof(char)** equals **1** and **sizeof(int)** is often **4**.

The arithmetic operators can be used for appropriate combinations of these types:

```
x+y     // plus
+x      // unary plus
x-y     // minus
-x      // unary minus
x*y     // multiply
x/y     // divide
x%y     // remainder (modulus) for integers
```

So can the comparison operators:

```
x==y    // equal
x!=y    // not equal
x<y     // less than
x>y     // greater than
x<=y    // less than or equal
x>=y    // greater than or equal
```

Furthermore, logical operators are provided:

```
x&y     // bitwise and
x|y     // bitwise or
x^y     // bitwise exclusive or
~x      // bitwise complement
x&&y    // logical and
x||y    // logical or
```

A bitwise logical operator yield a result of their operand type for which the operation has been performed on each bit. The logical operators **&&** and **||** simply return **true** or **false** depending on the values of their operands.

In assignments and in arithmetic operations, C++ performs all meaningful conversions between the basic types so that they can be mixed freely:

```
void some_function()    // function that doesn't return a value
{
    double d = 2.2;     // initialize floating-point number
    int i = 7;          // initialize integer
    d = d+i;            // assign sum to d
    i = d*i;            // assign product to i (truncating the double d*i to an int)
}
```

The conversions use in expressions are called *the usual arithmetic conversions* and aim to ensure that expressions are computed at the highest precision of its operands. For example, an addition of a **double** and an **int** is calculated using double-precision floating-point arithmetic.

Note that **=** is the assignment operator and **==** tests equality.

C++ offers a variety of notations for expressing initialization, such as the **=** used above, and a universal form based on curly-brace-delimited initializer lists:

```
double d1 = 2.3;        // initialize d1 to 2.3
double d2 {2.3};        // initialize d2 to 2.3
```

```
complex<double> z = 1;          // a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2};     // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6};    // a vector of ints
```

The **=** form is traditional and dates back to C, but if in doubt, use the general **{}**-list form. If nothing else, it saves you from conversions that lose information:

```
int i1 = 7.2;       // i1 becomes 7 (surprise?)
int i2 {7.2};       // error: floating-point to integer conversion
int i3 = {7.2};     // error: floating-point to integer conversion (the = is redundant)
```

Unfortunately, conversions that lose information, *narrowing conversions*, such as **double** to **int** and **int** to **char** are allowed and implicitly applied. The problems caused by implicit narrowing conversions is a price paid for C compatibility (§14.3).

A constant (§1.7) cannot be left uninitialized and a variable should only be left uninitialized in extremely rare circumstances. Don't introduce a name until you have a suitable value for it. User-defined types (such as **string**, **vector**, **Matrix**, **Motor_controller**, and **Orc_warrior**) can be defined to be implicitly initialized (§4.2.1).

When defining a variable, you don't actually need to state its type explicitly when it can be deduced from the initializer:

```
auto b = true;      // a bool
auto ch = 'x';      // a char
auto i = 123;       // an int
auto d = 1.2;       // a double
auto z = sqrt(y);   // z has the type of whatever sqrt(y) returns
```

With **auto**, we use the **=** because there is no potentially troublesome type conversion involved.

We use **auto** where we don't have a specific reason to mention the type explicitly. "Specific reasons" include:

- The definition is in a large scope where we want to make the type clearly visible to readers of our code.
- We want to be explicit about a variable's range or precision (e.g., **double** rather than **float**).

Using **auto**, we avoid redundancy and writing long type names. This is especially important in generic programming where the exact type of an object can be hard for the programmer to know and the type names can be quite long (§10.2).

In addition to the conventional arithmetic and logical operators, C++ offers more specific operations for modifying a variable:

```
x+=y    // x = x+y
++x     // increment: x = x+1
x-=y    // x = x-y
--x     // decrement: x = x-1
x*=y    // scaling: x = x*y
x/=y    // scaling: x = x/y
x%=y    // x = x%y
```

These operators are concise, convenient, and very frequently used.

1.6 Scope and Lifetime

A declaration introduces its name into a scope:

- *Local scope*: A name declared in a function (§1.4) or lambda (§5.5) is called a *local name*. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A *block* is delimited by a **{ }** pair. Function argument names are considered local names.
- *Class scope*: A name is called a *member name* (or a *class member name*) if it is defined in a class (§2.2, §2.3, Chapter 4), outside any function (§1.4), lambda (§5.5), or **enum class** (§2.5). Its scope extends from the opening **{** of its enclosing declaration to the end of that declaration.
- *Namespace scope*: A name is called a *namespace member name* if it is defined in a namespace (§3.3) outside any function, lambda (§5.5), class (§2.2, §2.3, Chapter 4), or **enum class** (§2.5). Its scope extends from the point of declaration to the end of its namespace.

A name not declared inside any other construct is called a *global name* and is said to be in the *global namespace*.

In addition, we can have objects without names, such as temporaries and objects created using **new** (§4.2.2). For example:

```
vector<int> vec;      // vec is global (a global vector of integers)

struct Record {
    string name;      // name is a member (a string member)
    // ...
};

void fct(int arg)     // fct is global (a global function)
                      // arg is local (an integer argument)
{
    string motto {"Who dares win"};     // motto is local
    auto p = new Record{"Hume"};        // p points to an unnamed Record (created by new)
    // ...
}
```

An object must be constructed (initialized) before it is used and will be destroyed at the end of its scope. For a namespace object the point of destruction is the end of the program. For a member, the point of destruction is determined by the point of destruction of the object of which it is a member. An object created by **new** "lives" until destroyed by **delete** (§4.2.2).

1.7 Constants

C++ supports two notions of immutability:

- **const**: meaning roughly "I promise not to change this value." This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by **const**.

- **constexpr**: meaning roughly "to be evaluated at compile time." This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted) and for performance.

For example:

```
const int dmv = 17;                          // dmv is a named constant
int var = 17;                                // var is not a constant

constexpr double max1 = 1.4*square(dmv);     // OK if square(17) is a constant expression
constexpr double max2 = 1.4*square(var);     // error: var is not a constant expression
const double max3 = 1.4*square(var);         // OK, may be evaluated at run time

double sum(const vector<double>&);           // sum will not modify its argument (§1.8)
vector<double> v {1.2, 3.4, 4.5};            // v is not a constant
const double s1 = sum(v);                    // OK: evaluated at run time
constexpr double s2 = sum(v);                // error: sum(v) not constant expression
```

For a function to be usable in a *constant expression*, that is, in an expression that will be evaluated by the compiler, it must be defined **constexpr**. For example:

```
constexpr double square(double x) { return x*x; }
```

To be **constexpr**, a function must be rather simple: just a **return**-statement computing a value. A **constexpr** function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a **constexpr** function to be called with non-constant-expression arguments in contexts that do not require constant expressions, so that we don't have to define essentially the same function twice: once for constant expressions and once for variables.

In a few places, constant expressions are required by language rules (e.g., array bounds (§1.8), case labels (§1.9), template value arguments (§5.2), and constants declared using **constexpr**). In other cases, compile-time evaluation is important for performance. Independently of performance issues, the notion of immutability (of an object with an unchangeable state) is an important design concern.

1.8 Pointers, Arrays, and References

An array of elements of type **char** can be declared like this:

```
char v[6];          // array of 6 characters
```

Similarly, a pointer can be declared like this:

```
char* p;            // pointer to character
```

In declarations, **[ ]** means "array of" and * means "pointer to." All arrays have **0** as their lower bound, so **v** has six elements, **v[0]** to **v[5]**. The size of an array must be a constant expression (§1.7). A pointer variable can hold the address of an object of the appropriate type:

```
char* p = &v[3];    // p points to v's fourth element
char x = *p;        // *p is the object that p points to
```

In an expression, prefix unary * means "contents of" and prefix unary & means "address of." We can represent the result of that initialized definition graphically:

Consider copying ten elements from one array to another:

```
void copy_fct()
{
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10];                 // to become a copy of v1

    for (auto i=0; i!=10; ++i)  // copy elements
        v2[i]=v1[i];
    // ...
}
```

This **for**-statement can be read as "set **i** to zero; while **i** is not **10**, copy the **i**th element and increment **i**." When applied to an integer variable, the increment operator, **++**, simply adds **1**. C++ also offers a simpler **for**-statement, called a range-**for**-statement, for loops that traverse a sequence in the simplest way:

```
void print()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v)            // for each x in v
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

The first range-**for**-statement can be read as "for every element of **v**, from the first to the last, place a copy in **x** and print it." Note that we don't have to specify an array bound when we initialize it with a list. The range-**for**-statement can be used for any sequence of elements (§10.1).

If we didn't want to copy the values from **v** into the variable **x**, but rather just have **x** refer to an element, we could write:

```
void increment()
{
    int v[] = {0,1,2,3,4,5,6,7,8,9};
```

```
        for (auto& x : v)
            ++x;
        // ...
}
```

In a declaration, the unary suffix **&** means "reference to." A reference is similar to a pointer, except that you don't need to use a prefix * to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization.

References are particularly useful for specifying function arguments. For example:

```
void sort(vector<double>& v);        // sort v
```

By using a reference, we ensure that for a call **sort(my_vec)**, we do not copy **my_vec** and that it really is **my_vec** that is sorted and not a copy of it.

When we don't want to modify an argument, but still don't want the cost of copying, we use a **const** reference. For example:

```
double sum(const vector<double>&)
```

Functions taking **const** references are very common.

When used in declarations, operators (such as **&**, *, and **[ ]**) are called *declarator operators*:

```
T a[n];     // T[n]: array of n Ts
T* p;       // T*: pointer to T
T& r;       // T&: reference to T
T f(A);     // T(A): function taking an argument of type A returning a result of type T
```

We try to ensure that a pointer always points to an object, so that dereferencing it is valid. When we don't have an object to point to or if we need to represent the notion of "no object available" (e.g., for an end of a list), we give the pointer the value **nullptr** ("the null pointer"). There is only one **nullptr** shared by all pointer types:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;    // pointer to a Link to a Record
int x = nullptr;                // error: nullptr is a pointer not an integer
```

It is often wise to check that a pointer argument that is supposed to point to something, actually points to something:

```
int count_x(char* p, char x)
    // count the number of occurrences of x in p[]
    // p is assumed to point to a zero-terminated array of char (or to nothing)
{
    if (p==nullptr) return 0;
    int count = 0;
    for (; p!=nullptr; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

Note how we can move a pointer to point to the next element of an array using **++** and that we can leave out the initializer in a **for**-statement if we don't need it.

The definition of **count_x()** assumes that the **char*** is a *C-style string*, that is, that the pointer points to a zero-terminated array of **char**.

In older code, **0** or **NULL** is typically used instead of **nullptr**. However, using **nullptr** eliminates potential confusion between integers (such as **0** or **NULL**) and pointers (such as **nullptr**).

The **count_if()** example is unnecessarily complicated. We can simplify it by testing for the **nullptr** in one place only. We are not using the initializer part of the **for**-statement, so we can use the simpler **while**-statement:

```
int count_x(char* p, char x)
    // count the number of occurrences of x in p[]
    // p is assumed to point to a zero-terminated array of char (or to nothing)
{
    int count = 0;
    while (p) {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

The **while**-statement executes until its condition becomes **false**.

A test of a pointer (e.g., **while (p)**) is equivalent to comparing the pointer to the null pointer (e.g., **while (p!=nullptr)**).

1.9 Tests

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question

    char answer = 0;
    cin >> answer;                                   // read answer

    if (answer == 'y')
        return true;
    return false;
}
```

To match the **<<** output operator (“put to”), the **>>** operator (“get from”) is used for input; **cin** is the standard input stream (Chapter 8). The type of the right-hand operand of **>>** determines what input is accepted, and its right-hand operand is the target of the input operation. The **\n** character at the end of the output string represents a newline (§1.3).

Note that the definition of **answer** appears where it is needed (and not before that). A declaration can appear anywhere a statement can.

The example could be improved by taking an **n** (for ‘‘no’’) answer into account:

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question

    char answer = 0;
    cin >> answer;                                   // read answer

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

A **switch**-statement tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the **default** is chosen. If no **default** is provided, no action is taken if the value doesn’t match any case constant.

We don’t have to exit a **case** by returning from the function that contains its **switch**-statement. Often, we just want to continue execution with the statement following the **switch**-statement. We can do that using a **break** statement. As an example, consider an overly clever, yet primitive, parser for a trivial command video game:

```
void action()
{
    while (true) {
        cout << "enter action:\n";          // request action
        string act;
        cin >> act;          // rear characters into a string
        Point delta {0,0};   // Point holds an {x,y} pair

        for (char ch : act) {
            switch (ch) {
            case 'u':  // up
            case 'n':  // north
                ++delta.y;
                break;
            case 'r':  // right
            case 'e':  // east
                ++delta.x;
                break;
            // ... more actions ...
```

```
                default:
                    cout << "I freeze!\n";
                }
                move(current+delta*scale);
                update_display();
            }
        }
    }
```

1.10 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 5-6, 9-10, and 12 of [Stroustrup,2013].
[2] Don't panic! All will become clear in time; §1.1.
[3] You don't have to know every detail of C++ to write good programs.
[4] Focus on programming techniques, not on language features.
[5] For the final word on language definition issues, see the ISO C++ standard; §14.1.3.
[6] "Package" meaningful operations as carefully named functions; §1.4.
[7] A function should perform a single logical operation; §1.4.
[8] Keep functions short; §1.4.
[9] Use overloading when functions perform conceptually the same task on different types; §1.4.
[10] If a function may have to be evaluated at compile time, declare it `constexpr`; §1.7.
[11] Avoid "magic constants;" use symbolic constants; §1.7.
[12] Declare one name (only) per declaration.
[13] Keep common and local names short, and keep uncommon and nonlocal names longer.
[14] Avoid similar-looking names.
[15] Avoid `ALL_CAPS` names.
[16] Prefer the `{}`-initializer syntax for declarations with a named type; §1.5.
[17] Prefer the `=` syntax for the initialization in declarations using `auto`; §1.5.
[18] Avoid uninitialized variables; §1.5.
[19] Keep scopes small; §1.6.
[20] Keep use of pointers simple and straightforward; §1.8.
[21] Use `nullptr` rather than `0` or `NULL`; §1.8.
[22] Don't declare a variable until you have a value to initialize it with; §1.8, §1.9.
[23] Don't say in comments what can be clearly stated in code.
[24] State intent in comments.
[25] Maintain a consistent indentation style.
[26] Avoid complicated expressions.
[27] Avoid narrowing conversions; §1.5.

2

User-Defined Types

Don't Panic!
– Douglas Adams

- Introduction
- Structures
- Classes
- Unions
- Enumerations
- Advice

2.1 Introduction

We call the types that can be built from the fundamental types (§1.5), the **const** modifier (§1.7), and the declarator operators (§1.8) *built-in types*. C++'s set of built-in types and operations is rich, but deliberately low-level. They directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide the programmer with high-level facilities to conveniently write advanced applications. Instead, C++ augments the built-in types and operations with a sophisticated set of *abstraction mechanisms* out of which programmers can build such high-level facilities. The C++ abstraction mechanisms are primarily designed to let programmers design and implement their own types, with suitable representations and operations, and for programmers to simply and elegantly use such types. Types built out of the built-in types using C++'s abstraction mechanisms are called *user-defined types*. They are referred to as classes and enumerations. Most of this book is devoted to the design, implementation, and use of user-defined types. The rest of this chapter presents the simplest and most fundamental facilities for that. Chapters 4-5 are a more complete description of the abstraction mechanisms and the programming styles they support. Chapters 6-13 present an overview of the standard library, and since the standard library mainly consists of user-defined types, they provide examples of what can be built using the language facilities and programming techniques presented in Chapters 1-5.

2.2 Structures

The first step in building a new type is often to organize the elements it needs into a data structure, a **struct**:

```
struct Vector {
    int sz;          // number of elements
    double* elem;    // pointer to elements
};
```

This first version of **Vector** consists of an **int** and a **double***.

A variable of type **Vector** can be defined like this:

```
Vector v;
```

However, by itself that is not of much use because **v**'s **elem** pointer doesn't point to anything. To be useful, we must give **v** some elements to point to. For example, we can construct a **Vector** like this:

```
void vector_init(Vector& v, int s)
{
    v.elem = new double[s];   // allocate an array of s doubles
    v.sz = s;
}
```

That is, **v**'s **elem** member gets a pointer produced by the **new** operator and **v**'s **sz** member gets the number of elements. The **&** in **Vector&** indicates that we pass **v** by non-**const** reference (§1.8); that way, **vector_init()** can modify the vector passed to it.

The **new** operator allocates memory from an area called *the free store* (also known as *dynamic memory* and *heap*). Objects allocated on the free store are independent of the scope from which they are created and "live" until they are destroyed using the **delete** operator (§4.2.2).

A simple use of **Vector** looks like this:

```
double read_and_sum(int s)
    // read s integers from cin and return their sum; s is assumed to be positive
{
    Vector v;
    vector_init(v,s);               // allocate s elements for v
    for (int i=0; i!=s; ++i)
        cin>>v.elem[i];             // read into elements

    double sum = 0;
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i];             // take the sum of the elements
    return sum;
}
```

There is a long way to go before our **Vector** is as elegant and flexible as the standard-library **vector**. In particular, a user of **Vector** has to know every detail of **Vector**'s representation. The rest of this chapter and the next two gradually improve **Vector** as an example of language features and techniques. Chapter 9 presents the standard-library **vector**, which contains many nice improvements.

I use **vector** and other standard-library components as examples

- to illustrate language features and design techniques, and
- to help you learn and use the standard-library components.

Don't reinvent standard-library components, such as **vector** and **string**; use them.

We use **.** (dot) to access **struct** members through a name (and through a reference) and **->** to access **struct** members through a pointer. For example:

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;          // access through name
    int i2 = rv.sz;         // access through reference
    int i4 = pv->sz;        // access through pointer
}
```

2.3 Classes

Having the data specified separately from the operations on it has advantages, such as the ability to use the data in arbitrary ways. However, a tighter connection between the representation and the operations is needed for a user-defined type to have all the properties expected of a "real type." In particular, we often want to keep the representation inaccessible to users, so as to ease use, guarantee consistent use of the data, and allow us to later improve the representation. To do that we have to distinguish between the interface to a type (to be used by all) and its implementation (which has access to the otherwise inaccessible data). The language mechanism for that is called a *class*. A class is defined to have a set of *members*, which can be data, function, or type members. The interface is defined by the **public** members of a class, and **private** members are accessible only through that interface. For example:

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { }    // construct a Vector
    double& operator[](int i) { return elem[i]; }    // element access: subscripting
    int size() { return sz; }
private:
    double* elem;  // pointer to the elements
    int sz;        // the number of elements
};
```

Given that, we can define a variable of our new type **Vector**:

```
Vector v(6);    // a Vector with 6 elements
```

We can illustrate a **Vector** object graphically:

Basically, the **Vector** object is a "handle" containing a pointer to the elements (**elem**) plus the number of elements (**sz**). The number of elements (6 in the example) can vary from **Vector** object to **Vector** object, and a **Vector** object can have a different number of elements at different times (§4.2.3). However, the **Vector** object itself is always the same size. This is the basic technique for handling varying amounts of information in C++: a fixed-size handle referring to a variable amount of data "elsewhere" (e.g., on the free store allocated by **new**; §4.2.2). How to design and use such objects is the main topic of Chapter 4.

Here, the representation of a **Vector** (the members **elem** and **sz**) is accessible only through the interface provided by the **public** members: **Vector()**, **operator[]()**, and **size()**. The **read_and_sum()** example from §2.2 simplifies to:

```
double read_and_sum(int s)
{
    Vector v(s);                    // make a vector of s elements
    for (int i=0; i!=v.size(); ++i)
        cin>>v[i];                  // read into elements

    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=v[i];                  // take the sum of the elements
    return sum;
}
```

A "function" with the same name as its class is called a *constructor*, that is, a function used to construct objects of a class. So, the constructor, **Vector()**, replaces **vector_init()** from §2.2. Unlike an ordinary function, a constructor is guaranteed to be used to initialize objects of its class. Thus, defining a constructor eliminates the problem of uninitialized variables for a class.

Vector(int) defines how objects of type **Vector** are constructed. In particular, it states that it needs an integer to do that. That integer is used as the number of elements. The constructor initializes the **Vector** members using a member initializer list:

```
:elem{new double[s]}, sz{s}
```

That is, we first initialize **elem** with a pointer to **s** elements of type **double** obtained from the free store. Then, we initialize **sz** to **s**.

Access to elements is provided by a subscript function, called **operator[]**. It returns a reference to the appropriate element (a **double&**).

The **size()** function is supplied to give users the number of elements.

Obviously, error handling is completely missing, but we'll return to that in §3.4. Similarly, we did not provide a mechanism to "give back" the array of **double**s acquired by **new**; §4.2.2 shows how to use a destructor to elegantly do that.

There is no fundamental difference between a **struct** and a **class**; a **struct** is simply a **class** with members **public** by default. For example, you can define constructors and other member functions for a **struct**.

2.4 Unions

A **union** is a **struct** in which all members are allocated at the same address so that the **union** occupies only as much space as its largest member. Naturally, a **union** can hold a value for only one member at a time. For example, consider a symbol table entry that holds a name and a value:

```
enum Type { str, num };

struct Entry {
        char* name;
        Type t;
        char* s;    // use s if t==str
        int i;      // use i if t==num
};

void f(Entry* p)
{
        if (p->t == str)
                cout << p->s;
        // ...
}
```

The members **s** and **i** can never be used at the same time, so space is wasted. It can be easily recovered by specifying that both should be members of a **union**, like this:

```
union Value {
        char* s;
        int i;
};
```

The language doesn't keep track of which kind of value is held by a **union**, so the programmer must do that:

```
struct Entry {
        char* name;
        Type t;
        Value v;    // use v.s if t==str; use v.i if t==num
};

void f(Entry* p)
{
        if (p->t == str)
                cout << p->v.s;
        // ...
}
```

Maintaining the correspondence between a *type field* (here, **t**) and the type held in a **union** is error-prone. To avoid errors, one can encapsulate a **union** so that the correspondence between a type field and access to the **union** members is guaranteed. At the application level, abstractions relying on such *tagged unions* are common and useful, but use of "naked" **union**s is best minimized.

2.5 Enumerations

In addition to classes, C++ supports a simple form of user-defined type for which we can enumerate the values:

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Note that enumerators (e.g., **red**) are in the scope of their **enum class**, so that they can be used repeatedly in different **enum class**es without confusion. For example, **Color::red** is **Color**'s **red** which is different from **Traffic_light::red**.

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error-prone than it would have been had the symbolic (and mnemonic) enumerator names not been used.

The **class** after the **enum** specifies that an enumeration is strongly typed and that its enumerators are scoped. Being separate types, **enum class**es help prevent accidental misuses of constants. In particular, we cannot mix **Traffic_light** and **Color** values:

```
Color x = red;                  // error: which red?
Color y = Traffic_light::red;   // error: that red is not a Color
Color z = Color::red;           // OK
```

Similarly, we cannot implicitly mix **Color** and integer values:

```
int i = Color::red;             // error: Color::red is not an int
Color c = 2;                    // error: 2 is not a Color
```

By default, an **enum class** has only assignment, initialization, and comparisons (e.g., **==** and **<**; §1.5) defined. However, an enumeration is a user-defined type so we can define operators for it:

```
Traffic_light& operator++(Traffic_light& t)
    // prefix increment: ++
{
    switch (t) {
    case Traffic_light::green:      return t=Traffic_light::yellow;
    case Traffic_light::yellow:     return t=Traffic_light::red;
    case Traffic_light::red:        return t=Traffic_light::green;
    }
}

Traffic_light next = ++light;       // next becomes Traffic_light::green
```

If you don't want to explicitly qualify enumerator names and want enumerator values to be **int**s (without the need for an explicit conversion), you can remove the **class** from **enum class** to get a "plain" **enum**. The enumerators from a "plain" **enum** are entered into the same scope as the name of their **enum** and implicitly converts to their integer value. For example:

```
enum Color { red, green, blue };
int col = green;
```

Here **col** gets the value **1**. By default, the integer values of enumerators starts with **0** and increases by one for each additional enumerator. The "plain" **enum**s have been in C++ (and C) from the earliest days, so even though they are less well behaved, they are common in current code.

2.6 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 8 of [Stroustrup,2013].
[2] Organize related data into structures (**struct**s or **class**es); §2.2.
[3] Represent the distinction between an interface and an implemetation using a **class**; §2.3.
[4] A **struct** is simply a **class** with its members **public** by default; §2.3.
[5] Define constructors to guarantee and simplify initialization of **class**es; §2.3.
[6] Avoid "naked" **union**s; wrap them in a class together with a type field; §2.4.
[7] Use enumerations to represent sets of named constants; §2.5.
[8] Prefer **class enum**s over "plain" **enum**s to minimize surprises; §2.5.
[9] Define operations on enumerations for safe and simple use; §2.5.

This page intentionally left blank

3

Modularity

Don't interrupt me while I'm interrupting.
– Winston S. Churchill

- Introduction
- Separate Compilation
- Namespaces
- Error Handling
  Exceptions; Invariants; Static Assertions
- Advice

3.1 Introduction

A C++ program consists of many separately developed parts, such as functions (§1.3), user-defined types (Chapter 2), class hierarchies (§4.5), and templates (Chapter 5). The key to managing this is to clearly define the interactions among those parts. The first and most important step is to distinguish between the interface to a part and its implementation. At the language level, C++ represents interfaces by declarations. A *declaration* specifies all that's needed to use a function or a type. For example:

```
double sqrt(double);      // the square root function takes a double and returns a double

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;  // elem points to an array of sz doubles
    int sz;
};
```

The key point here is that the function bodies, the function *definitions*, are "elsewhere." For this example, we might like for the representation of **Vector** to be "elsewhere" also, but we will deal with that later (abstract types; §4.3). The definition of **sqrt()** will look like this:

```
double sqrt(double d)          // definition of sqrt()
{
    // ... algorithm as found in math textbook ...
}
```

For **Vector**, we need to define all three member functions:

```
Vector::Vector(int s)                    // definition of the constructor
    :elem{new double[s]}, sz{s}          // initialize members
{
}

double& Vector::operator[](int i)        // definition of subscripting
{
    return elem[i];
}

int Vector::size()                       // definition of size()
{
    return sz;
}
```

We must define **Vector**'s functions, but not **sqrt()** because it is part of the standard library. However, that makes no real difference: a library is simply some "other code we happen to use" written with the same language facilities as we use.

3.2 Separate Compilation

C++ supports a notion of separate compilation where user code sees only declarations of the types and functions used. The definitions of those types and functions are in separate source files and compiled separately. This can be used to organize a program into a set of semi-independent code fragments. Such separation can be used to minimize compilation times and to strictly enforce separation of logically distinct parts of a program (thus minimizing the chance of errors). A library is often a collection of separately compiled code fragments (e.g., functions).

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. For example:

```
// Vector.h:

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
```

```
private:
    double* elem;       // elem points to an array of sz doubles
    int sz;
};
```

This declaration would be placed in a file **Vector.h**, and users will *include* that file, called a *header file*, to access that interface. For example:

```
// user.cpp:

#include "Vector.h"      // get Vector's interface
#include <cmath>         // get the the standard-library math function interface including sqrt()

using namespace std;     // make std members visible (§3.3)

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=sqrt(v[i]);                // sum of square roots
    return sum;
}
```

To help the compiler ensure consistency, the **.cpp** file providing the implementation of **Vector** will also include the **.h** file providing its interface:

```
// Vector.cpp:

#include "Vector.h" // get the interface

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}         // initialize members
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}
```

The code in **user.cpp** and **Vector.cpp** shares the **Vector** interface information presented in **Vector.h**, but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:

Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation.

3.3 Namespaces

In addition to functions (§1.4), classes (§2.3), and enumerations (§2.5), C++ offers *namespaces* as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names. For example, I might want to experiment with my own complex number type (§4.2.1, §12.4):

```
namespace My_code {
        class complex {
                // ...
        };

        complex sqrt(complex);
        // ...

        int main();
}

int My_code::main()
{
        complex z {1,2};
        auto z2 = sqrt(z);
        std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
        // ...
};

int main()
{
        return My_code::main();
}
```

By putting my code into the namespace **My_code**, I make sure that my names do not conflict with the standard-library names in namespace **std** (§3.3). The precaution is wise, because the standard library does provide support for **complex** arithmetic (§4.2.1, §12.4).

The simplest way to access a name in another namespace is to qualify it with the namespace name (e.g., **std::cout** and **My_code::main**). The "real **main()**" is defined in the global namespace, that is, not local to a defined namespace, class, or function. To gain access to names in the standard-library namespace, we can use a **using**-directive:

```
using namespace std;
```

A **using**-directive makes names from the named namespace accessible as if they were local to the scope in which we placed the directive. So after the **using**-directive for **std**, we can simply write **cout** rather than **std::std**.

Namespaces are primarily used to organize larger program components, such as libraries. They simplify the composition of a program out of separately developed parts.

3.4 Error Handling

Error handling is a large and complex topic with concerns and ramifications that go far beyond language facilities into programming techniques and tools. However, C++ provides a few features to help. The major tool is the type system itself. Instead of painstakingly building up our applications from the built-in types (e.g., **char**, **int**, and **double**) and statements (e.g., **if**, **while**, and **for**), we build more types that are appropriate for our applications (e.g., **string**, **map**, and **regex**) and algorithms (e.g., **sort()**, **find_if()**, and **draw_all()**). Such higher-level constructs simplify our programming, limit our opportunities for mistakes (e.g., you are unlikely to try to apply a tree traversal to a dialog box), and increase the compiler's chances of catching such errors. The majority of C++ constructs are dedicated to the design and implementation of elegant and efficient abstractions (e.g., user-defined types and algorithms using them). One effect of this modularity and abstraction (in particular, the use of libraries) is that the point where a run-time error can be detected is separated from the point where it can be handled. As programs grow, and especially when libraries are used extensively, standards for handling errors become important. It is a good idea to design and articulate a strategy for error handling early on in the development of a program.

3.4.1 Exceptions

Consider again the **Vector** example. What *ought* to be done when we try to access an element that is out of range for the vector from §2.3?

- The writer of **Vector** doesn't know what the user would like to have done in this case (the writer of **Vector** typically doesn't even know in which program the vector will be running).
- The user of **Vector** cannot consistently detect the problem (if the user could, the out-of-range access wouldn't happen in the first place).

The solution is for the **Vector** implementer to detect the attempted out-of-range access and then tell the user about it. The user can then take appropriate action. For example, **Vector::operator[]()** can detect an attempted out-of-range access and throw an **out_of_range** exception:

```
double& Vector::operator[](int i)
{
	if (i<0 || size()<=i)
		throw out_of_range{"Vector::operator[]"};
	return elem[i];
}
```

The **throw** transfers control to a handler for exceptions of type **out_of_range** in some function that directly or indirectly called **Vector::operator[]()**. To do that, the implementation will *unwind* the function call stack as needed to get back to the context of that caller. That is, the exception handling mechanism will exit scopes and function as needed to get back to a caller that has expressed interest in handling that kind of exception, invoking destructors (§4.2.2) along the way as needed. For example:

```
void f(Vector& v)
{
	// ...
	try { // exceptions here are handled by the handler defined below

		v[v.size()] = 7;  // try to access beyond the end of v
	}
	catch (out_of_range) {	// oops: out_of_range error
		// ... handle range error ...
	}
	// ...
}
```

We put code for which we are interested in handling exceptions into a **try**-block. That attempted assignment to **v[v.size()]** will fail. Therefore, the **catch**-clause providing a handler for **out_of_range** will be entered. The **out_of_range** type is defined in the standard library (in **<stdexcept>**) and is in fact used by some standard-library container access functions.

Use of the exception-handling mechanisms can make error handling simpler, more systematic, and more readable. To achieve that don't overuse **try**-statements. The main technique for making error handling simple and systematic (called *Resource Aquisition Is Initialization*) is explained in §4.2.2.

A function that should never throw an exception can be declared **noexcept**. For example:

```
void user(int sz) noexcept
{
	Vector v(sz);
	iota(&v[0],&v[sz],1); // fill v with 1,2,3,4...
	// ...
}
```

If all good intent and planning fails, so that **user()** still throws, the standard-library function **terminate()** is called to immediately terminate the program.

3.4.2 Invariants

The use of exceptions to signal out-of-range access is an example of a function checking its argument and refusing to act because a basic assumption, a *precondition*, didn't hold. Had we formally specified **Vector**'s subscript operator, we would have said something like "the index must be in the [**0**:**size()**) range," and that was in fact what we tested in our **operator[]()**. The [**a**:**b**) notation specifies a half-open range, meaning that **a** is part of the range, but **b** is not. Whenever we define a function, we should consider what its preconditions are and if feasible test them.

However, **operator[]()** operates on objects of type **Vector** and nothing it does makes any sense unless the members of **Vector** have "reasonable" values. In particular, we did say "**elem** points to an array of **sz** doubles" but we only said that in a comment. Such a statement of what is assumed to be true for a class is called a *class invariant*, or simply an *invariant*. It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit. Unfortunately, our **Vector** constructor only partially did its job. It properly initialized the **Vector** members, but it failed to check that the arguments passed to it made sense. Consider:

```
Vector v(–27);
```

This is likely to cause chaos.

Here is a more appropriate definition:

```
Vector::Vector(int s)
{
    if (s<0)
        throw length_error{};
    elem = new double[s];
    sz = s;
}
```

I use the standard-library exception **length_error** to report a non-positive number of elements because some standard-library operations use that exception to report problems of this kind. If operator **new** can't find memory to allocate, it throws a **std::bad_alloc**. We can now write:

```
void test()
{
    try {
        Vector v(–27);
    }
    catch (std::length_error) {
        // handle negative size
    }
    catch (std::bad_alloc) {
        // handle memory exhaustion
    }
}
```

You can define your own classes to be used as exceptions and have them carry arbitrary information from a point where an error is detected to a point where it can be handled (§3.4.1).

Often, a function has no way of completing its assigned task after an exception is thrown. Then, "handling" an exception simply means doing some minimal local cleanup and rethrowing the exception. To throw (*rethrow*) the exception caught in an exception handler, we simply write **throw;**. For example:

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error) {
        cout << "test failed: length error\n";
        throw;    // rethrow
    }
    catch (std::bad_alloc) {
        // Ouch! test() is not designed to handle memory exhaustion
        std::terminate();    // terminate the program
    }
}
```

The notion of invariants is central to the design of classes, and preconditions serve a similar role in the design of functions. Invariants

- helps us to understand precisely what we want
- forces us to be specific; that gives us a better chance of getting our code correct (after debugging and testing).

The notion of invariants underlies C++'s notions of resource management supported by constructors (Chapter 4) and destructors (§4.2.2, §11.2).

3.4.3 Static Assertions

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so. That's what much of the type system and the facilities for specifying the interfaces to user-defined types are for. However, we can also perform simple checks on other properties that are known at compile time and report failures as compiler error messages. For example:

```
static_assert(4<=sizeof(int), "integers are too small");   // check integer size
```

This will write **integers are too small** if **4<=sizeof(int)** does not hold, that is, if an **int** on this system does not have at least 4 bytes. We call such statements of expectations *assertions*.

The **static_assert** mechanism can be used for anything that can be expressed in terms of constant expressions (§1.7). For example:

```
constexpr double C = 299792.458;                    // km/s

void f(double speed)
{
    const double local_max = 160.0/(60*60);         // 160 km/h == 160.0/(60*60) km/s
```

```
        static_assert(speed<C,"can't go that fast");        // error: speed must be a constant
        static_assert(local_max<C,"can't go that fast");    // OK

        // ...
}
```

In general, **static_assert(A,S)** prints **S** as a compiler error message if **A** is not **true**.

The most important uses of **static_assert** come when we make assertions about types used as parameters in generic programming (§5.4, §11.6).

For runtime-checked assertions, use exceptions.

3.5 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 13-15 of [Stroustrup,2013].
[2] Distinguish between declarations (used as interfaces) and definitions (used as implementations); §3.1.
[3] Use header files to represent interfaces and to emphasize logical structure; §3.2.
[4] **#include** a header in the source file that implements its functions; §3.2.
[5] Avoid non-inline function definitions in headers; §3.2.
[6] Use namespaces to express logical structure; §3.3.
[7] Use **using**-directives for transition, for foundational libraries (such as **std**), or within a local scope; §3.3.
[8] Don't put a **using**-directive in a header file; §3.3.
[9] Throw an exception to indicate that you cannot perform an assigned task; §3.4.
[10] Use exceptions for error handling; §3.4.
[11] Develop an error-handling strategy early in a design; §3.4.
[12] Use purpose-designed user-defined types as exceptions (not built-in types); §3.4.1.
[13] Don't try to catch every exception in every function; §3.4.
[14] If your function may not throw, declare it **noexcept**; §3.4.
[15] Let a constructor establish an invariant, and throw if it cannot; §3.4.2.
[16] Design your error-handling strategy around invariants; §3.4.2.
[17] What can be checked at compile time is usually best checked at compile time (using **static_assert**); §3.4.3.

This page intentionally left blank

4

Classes

Those types are not "abstract";
they are as real as **int** *and* **float**.
– Doug McIlroy

- Introduction
- Concrete Types
  An Arithmetic Type; A Container; Initializing Containers
- Abstract Types
- Virtual Functions
- Class Hierarchies
  Explicit Overriding; Benefits from Hierarchies; Hierarchy Navigation; Avoiding Resource Leaks
- Copy and Move
  Copying Containers; Moving Containers; Essential Operations; Resource Management; Suppressing Operations
- Advice

4.1 Introduction

This chapter and the next aim to give you an idea of C++'s support for abstraction and resource management without going into a lot of detail:

- This chapter informally presents ways of defining and using new types (*user-defined types*). In particular, it presents the basic properties, implementation techniques, and language facilities used for *concrete classes*, *abstract classes*, and *class hierarchies*.
- The next chapter introduces templates as a mechanism for parameterizing types and algorithms with (other) types and algorithms. Computations on user-defined and built-in types are represented as functions, sometimes generalized to *template functions* and *function objects*.

These are the language facilities supporting the programming styles known as *object-oriented programming* and *generic programming*. Chapters 6-13 follow up by presenting examples of standard-library facilities and their use.

The central language feature of C++ is the *class*. A class is a user-defined type provided to represent a concept in the code of a program. Whenever our design for a program has a useful concept, idea, entity, etc., we try to represent it as a class in the program so that the idea is there in the code, rather than just in our head, in a design document, or in some comments. A program built out of a well chosen set of classes is far easier to understand and get right than one that builds everything directly in terms of the built-in types. In particular, classes are often what libraries offer.

Essentially all language facilities beyond the fundamental types, operators, and statements exist to help define better classes or to use them more conveniently. By "better," I mean more correct, easier to maintain, more efficient, more elegant, easier to use, easier to read, and easier to reason about. Most programming techniques rely on the design and implementation of specific kinds of classes. The needs and tastes of programmers vary immensely. Consequently, the support for classes is extensive. Here, we will just consider the basic support for three important kinds of classes:

- Concrete classes (§4.2)
- Abstract classes (§4.3)
- Classes in class hierarchies (§4.5)

An astounding number of useful classes turn out to be of these three kinds. Even more classes can be seen as simple variants of these kinds or are implemented using combinations of the techniques used for these.

4.2 Concrete Types

The basic idea of *concrete classes* is that they behave "just like built-in types." For example, a complex number type and an infinite-precision integer are much like built-in **int**, except of course that they have their own semantics and sets of operations. Similarly, a **vector** and a **string** are much like built-in arrays, except that they are better behaved (§7.2, §8.3, §9.2).

The defining characteristic of a concrete type is that its representation is part of its definition. In many important cases, such as a **vector**, that representation is only one or more pointers to data stored elsewhere, but it is present in each object of a concrete class. That allows implementations to be optimally efficient in time and space. In particular, it allows us to

- place objects of concrete types on the stack, in statically allocated memory, and in other objects (§1.6);
- refer to objects directly (and not just through pointers or references);
- initialize objects immediately and completely (e.g., using constructors; §2.3); and
- copy objects (§4.6).

The representation can be private (as it is for **Vector**; §2.3) and accessible only through the member functions, but it is present. Therefore, if the representation changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types. For types that don't change often, and where local variables provide much-needed clarity and efficiency, this is acceptable and often ideal. To increase flexibility, a concrete type can keep

major parts of its representation on the free store (dynamic memory, heap) and access them through the part stored in the class object itself. That's the way **vector** and **string** are implemented; they can be considered resource handles with carefully crafted interfaces.

4.2.1 An Arithmetic Type

The "classical user-defined arithmetic type" is **complex**:

```
class complex {
    double re, im;  // representation: two doubles
public:
    complex(double r, double i) :re{r}, im{i} {}     // construct complex from two scalars
    complex(double r) :re{r}, im{0} {}               // construct complex from one scalar
    complex() :re{0}, im{0} {}                       // default complex: {0,0}

    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }

    complex& operator+=(complex z) { re+=z.re, im+=z.im; return *this; }   // add to re and im
                                                                           // and return the result
    complex& operator-=(complex z) { re-=z.re, im-=z.im; return *this; }

    complex& operator*=(complex);    // defined out-of-class somewhere
    complex& operator/=(complex);    // defined out-of-class somewhere
};
```

This is a slightly simplified version of the standard-library **complex** (§12.4). The class definition itself contains only the operations requiring access to the representation. The representation is simple and conventional. For practical reasons, it has to be compatible with what Fortran provided 50 years ago, and we need a conventional set of operators. In addition to the logical demands, **complex** must be efficient or it will remain unused. This implies that simple operations must be inlined. That is, simple operations (such as constructors, **+=**, and **imag()**) must be implemented without function calls in the generated machine code. Functions defined in a class are inlined by default. It is possible to explicitly require inlining by preceeding a function declaration with the keyword **inline**. An industrial-strength **complex** (like the standard-library one) is carefully implemented to do appropriate inlining.

A constructor that can be invoked without an argument is called a *default constructor*. Thus, **complex()** is **complex**'s default constructor. By defining a default constructor you eliminate the possibility of uninitialized variables of that type.

The **const** specifiers on the functions returning the real and imaginary parts indicate that these functions do not modify the object for which they are called.

Many useful operations do not require direct access to the representation of **complex**, so they can be defined separately from the class definition:

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator–(complex a, complex b) { return a–=b; }
complex operator–(complex a) { return {–a.real(), –a.imag()}; }      // unary minus
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

Here, I use the fact that an argument passed by value is copied, so that I can modify an argument without affecting the caller's copy, and use the result as the return value.

The definitions of `==` and `!=` are straightforward:

```
bool operator==(complex a, complex b)        // equal
{
        return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)        // not equal
{
        return !(a==b);
}

complex sqrt(complex);        // the definition is elsewhere

// ...
```

Class `complex` can be used like this:

```
void f(complex z)
{
        complex a {2.3};            // construct {2.3,0.0} from 2.3
        complex b {1/a};
        complex c {a+z*complex{1,2.3}};
        // ...
        if (c != b)
                c = –(b/a)+2*b;
}
```

The compiler converts operators involving `complex` numbers into appropriate function calls. For example, `c!=b` means `operator!=(c,b)` and `1/a` means `operator/(complex{1},a)`.

User-defined operators ("overloaded operators") should be used cautiously and conventionally. The syntax is fixed by the language, so you can't define a unary `/`. Also, it is not possible to change the meaning of an operator for built-in types, so you can't redefine `+` to subtract `int`s.

4.2.2 A Container

A *container* is an object holding a collection of elements, so we call `Vector` a container because it is the type of objects that are containers. As defined in §2.3, `Vector` isn't an unreasonable container of `double`s: it is simple to understand, establishes a useful invariant (§3.4.2), provides range-checked access (§3.4.1), and provides `size()` to allow us to iterate over its elements. However, it does have a fatal flaw: it allocates elements using `new` but never deallocates them. That's not a good idea because although C++ defines an interface for a garbage collector (§4.6.4), it is not guaranteed that

one is available to make unused memory available for new objects. In some environments you can't use a collector, and sometimes you prefer more precise control of destruction for logical or performance reasons. We need a mechanism to ensure that the memory allocated by the constructor is deallocated; that mechanism is a *destructor*:

```
class Vector {
private:
    double* elem;        // elem points to an array of sz doubles
    int sz;
public:
    Vector(int s) :elem{new double[s]}, sz{s}        // constructor: acquire resources
    {
        for (int i=0; i!=s; ++i)            // initialize elements
            elem[i]=0;
    }

    ~Vector() { delete[] elem; }                        // destructor: release resources

    double& operator[](int i);
    int size() const;
};
```

The name of a destructor is the complement operator, ~, followed by the name of the class; it is the complement of a constructor. **Vector**'s constructor allocates some memory on the free store (also called the *heap* or *dynamic store*) using the **new** operator. The destructor cleans up by freeing that memory using the **delete** operator. This is all done without intervention by users of **Vector.** The users simply create and use **Vector**s much as they would variables of built-in types. For example:

```
void fct(int n)
{
    Vector v(n);

    // ... use v ...

    {
        Vector v2(2*n);
        // ... use v and v2 ...
    } // v2 is destroyed here

    // ... use v ..

} // v is destroyed here
```

Vector obeys the same rules for naming, scope, allocation, lifetime, etc. (§1.6), as does a built-in type, such as **int** and **char**. This **Vector** has been simplified by leaving out error handling; see §3.4.

The constructor/destructor combination is the basis of many elegant techniques. In particular, it is the basis for most C++ general resource management techniques (§11.2). Consider a graphical illustration of a **Vector**:

The constructor allocates the elements and initializes the **Vector** members appropriately. The destructor deallocates the elements. This *handle-to-data model* is very commonly used to manage data that can vary in size during the lifetime of an object. The technique of acquiring resources in a constructor and releasing them in a destructor, known as *Resource Acquisition Is Initialization* or *RAII*, allows us to eliminate "naked **new** operations," that is, to avoid allocations in general code and keep them buried inside the implementation of well-behaved abstractions. Similarly, "naked **delete** operations" should be avoided. Avoiding naked **new** and naked **delete** makes code far less error-prone and far easier to keep free of resource leaks (§11.2).

4.2.3 Initializing Containers

A container exists to hold elements, so obviously we need convenient ways of getting elements into a container. We can handle that by creating a **Vector** with an appropriate number of elements and then assigning to them, but typically other ways are more elegant. Here, I just mention two favorites:

- *Initializer-list constructor*: Initialize with a list of elements.
- **push_back()**: Add a new element at the end (at the back of) the sequence.

These can be declared like this:

```
class Vector {
public:
    Vector(std::initializer_list<double>);      // initialize with a list of doubles
    // ...
    void push_back(double);                     // add element at end, increasing the size by one
    // ...
};
```

The **push_back()** is useful for input of arbitrary numbers of elements. For example:

```
Vector read(istream& is)
{
    Vector v;
    for (double d; is>>d;)          // read floating-point values into d
        v.push_back(d);             // add d to v
    return v;
}
```

The input loop is terminated by an end-of-file or a formatting error. Until that happens, each number read is added to the **Vector** so that at the end, **v**'s size is the number of elements read. I used a **for**-statement rather than the more conventional **while**-statement to keep the scope of **d** limited to the loop. The way to provide **Vector** with a move constructor, so that returning a potentially huge amount of data from **read()** is cheap, is explained in §4.6.2.

The **std::initializer_list** used to define the initializer-list constructor is a standard-library type known to the compiler: when we use a **{}**-list, such as **{1,2,3,4}**, the compiler will create an object of type **initializer_list** to give to the program. So, we can write:

```
Vector v1 = {1,2,3,4,5};          // v1 has 5 elements
Vector v2 = {1.23, 3.45, 6.7, 8}; // v2 has 4 elements
```

Vector's initializer-list constructor might be defined like this:

```
Vector::Vector(std::initializer_list<double> lst)     // initialize with a list
    :elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
    copy(lst.begin(),lst.end(),elem);          // copy from lst into elem (§10.6)
}
```

I use the ugly **static_cast** (§14.2.3) to convert the size of the initializer list to an **int**. This is pedantic because the chance that the number of elements in a hand-written list is larger than the largest integer (32,767 for 16-bit integers and 2,147,483,647 for 32-bit integers) is rather low. However, it is worth remembering that the type system has no common sense. It knows about the possible values of variables, rater than actual values, so it might complain where there is no actual violation. However, sooner or later, such warnings will save the programmer from a bad error.

A **static_cast** is does not check the value it is converting; the programmer is trusted to use it correctly. This is not always a good assumption, so if in doubt, check the value. Explicit type conversions (often called *casts* to remind you that they are used to prop up something broken) are best avoided. Judicious use of the type system and well-designed libraries allow us to eliminate unchecked cast in higher-level software.

4.3 Abstract Types

Types such as **complex** and **Vector** are called *concrete types* because their representation is part of their definition. In that, they resemble built-in types. In contrast, an *abstract type* is a type that completely insulates a user from implementation details. To do that, we decouple the interface from the representation and give up genuine local variables. Since we don't know anything about the representation of an abstract type (not even its size), we must allocate objects on the free store (§4.2.2) and access them through references or pointers (§1.8, §11.2.1).

First, we define the interface of a class **Container** which we will design as a more abstract version of our **Vector**:

```
class Container {
public:
    virtual double& operator[](int) = 0;      // pure virtual function
    virtual int size() const = 0;             // const member function (§4.2.1)
    virtual ~Container() {}                   // destructor (§4.2.2)
};
```

This class is a pure interface to specific containers defined later. The word **virtual** means "may be redefined later in a class derived from this one." Unsurprisingly, a function declared **virtual** is called a *virtual function*. A class derived from **Container** provides an implementation for the

Container interface. The curious **=0** syntax says the function is *pure virtual*; that is, some class derived from **Container** *must* define the function. Thus, it is not possible to define an object that is just a **Container**; a **Container** can only serve as the interface to a class that implements its **operator[]()** and **size()** functions. A class with a pure virtual function is called an *abstract class*.

This **Container** can be used like this:

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

Note how **use()** uses the **Container** interface in complete ignorance of implementation details. It uses **size()** and **[ ]** without any idea of exactly which type provides their implementation. A class that provides the interface to a variety of other classes is often called a *polymorphic type*.

As is common for abstract classes, **Container** does not have a constructor. After all, it does not have any data to initialize. On the other hand, **Container** does have a destructor and that destructor is **virtual**. Again, that is common for abstract classes because they tend to be manipulated through references or pointers, and someone destroying a **Container** through a pointer has no idea what resources are owned by its implementation; see also §4.5.

A container that implements the functions required by the interface defined by the abstract class **Container** could use the concrete class **Vector**:

```
class Vector_container : public Container {    // Vector_container implements Container
        Vector v;
public:
        Vector_container(int s) : v(s) { }      // Vector of s elements
        ~Vector_container() {}

        double& operator[](int i) { return v[i]; }
        int size() const { return v.size(); }
};
```

The **:public** can be read as "is derived from" or "is a subtype of." Class **Vector_container** is said to be *derived* from class **Container**, and class **Container** is said to be a *base* of class **Vector_container**. An alternative terminology calls **Vector_container** and **Container** *subclass* and *superclass*, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as *inheritance*.

The members **operator[]()** and **size()** are said to *override* the corresponding members in the base class **Container**. The destructor (**~Vector_container()**) overrides the base class destructor (**~Container()**). Note that the member destructor (**~Vector()**) is implicitly invoked by its class's destructor (**~Vector_container()**).

For a function like **use(Container&)** to use a **Container** in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

```
void g()
{
    Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    use(vc);
}
```

Since use() doesn't know about Vector_containers but only knows the Container interface, it will work just as well for a different implementation of a Container. For example:

```
class List_container : public Container {      // List_container implements Container
    std::list<double> ld;      // (standard-library) list of doubles (§9.3)
public:
    List_container() { }       // empty List
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}

    double& operator[](int i);
    int size() const { return ld.size(); }

};

double& List_container::operator[](int i)
{
    for (auto& x : ld) {
        if (i==0) return x;
        --i;
    }
    throw out_of_range("List container");
}
```

Here, the representation is a standard-library list<double>. Usually, I would not implement a container with a subscript operation using a list, because performance of list subscripting is atrocious compared to vector subscripting. However, here I just wanted to show an implementation that is radically different from the usual one.

A function can create a List_container and have use() use it:

```
void h()
{
    List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    use(lc);
}
```

The point is that use(Container&) has no idea if its argument is a Vector_container, a List_container, or some other kind of container; it doesn't need to know. It can use any kind of Container. It knows only the interface defined by Container. Consequently, use(Container&) needn't be recompiled if the implementation of List_container changes or a brand-new class derived from Container is used.

The flip side of this flexibility is that objects must be manipulated through pointers or references (§4.6, §11.2.1).

4.4 Virtual Functions

Consider again the use of **Container**:

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

How is the call **c[i]** in **use()** resolved to the right **operator[]()**? When **h()** calls **use()**, **List_container**'s **operator[]()** must be called. When **g()** calls **use()**, **Vector_container**'s **operator[]()** must be called. To achieve this resolution, a **Container** object must contain information to allow it to select the right function to call at run time. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called the *virtual function table* or simply the **vtbl**. Each class with virtual functions has its own **vtbl** identifying its virtual functions. This can be represented graphically like this:

The functions in the **vtbl** allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. The implementation of the caller needs only to know the location of the pointer to the **vtbl** in a **Container** and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the "normal function call" mechanism (within 25%). Its space overhead is one pointer in each object of a class with virtual functions plus one **vtbl** for each such class.

4.5 Class Hierarchies

The **Container** example is a very simple example of a class hierarchy. A *class hierarchy* is a set of classes ordered in a lattice created by derivation (e.g., **: public**). We use class hierarchies to represent concepts that have hierarchical relationships, such as "A fire engine is a kind of a truck which

is a kind of a vehicle" and "A smiley face is a kind of a circle which is a kind of a shape." Huge hierarchies, with hundreds of classes, that are both deep and wide are common. As a semi-realistic classic example, let's consider shapes on a screen:

The arrows represent inheritance relationships. For example, class **Circle** is derived from class **Shape**. To represent that simple diagram in code, we must first specify a class that defines the general properties of all shapes:

```
class Shape {
public:
        virtual Point center() const =0;        // pure virtual
        virtual void move(Point to) =0;

        virtual void draw() const = 0;          // draw on current "Canvas"
        virtual void rotate(int angle) = 0;

        virtual ~Shape() {}                     // destructor
        // ...
};
```

Naturally, this interface is an abstract class: as far as representation is concerned, *nothing* (except the location of the pointer to the **vtbl**) is common for every **Shape**. Given this definition, we can write general functions manipulating vectors of pointers to shapes:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements by angle degrees
{
        for (auto p : v)
                p->rotate(angle);
}
```

To define a particular shape, we must say that it is a **Shape** and specify its particular properties (including its virtual functions):

```
class Circle : public Shape {
public:
        Circle(Point p, int rr);            // constructor

        Point center() const { return x; }
        void move(Point to) { x=to; }
```

```
        void draw() const;
        void rotate(int) {}                // nice simple algorithm
    private:
        Point x;    // center
        int r;      // radius
    };
```

So far, the **Shape** and **Circle** example provides nothing new compared to the **Container** and **Vector_container** example, but we can build further:

```
    class Smiley : public Circle {   // use the circle as the base for a face
    public:
        Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

        ~Smiley()
        {
            delete mouth;
            for (auto p : eyes)
                delete p;
        }

        void move(Point to);

        void draw() const;
        void rotate(int);

        void add_eye(Shape* s) { eyes.push_back(s); }
        void set_mouth(Shape* s);
        virtual void wink(int i);          // wink eye number i

        // ...

    private:
        vector<Shape*> eyes;               // usually two eyes
        Shape* mouth;
    };
```

The **push_back()** member function adds its argument to the **vector** (here, **eyes**), increasing that vector's size by one.

We can now define **Smiley::draw()** using calls to **Smiley**'s base and member **draw()**s:

```
    void Smiley::draw()
    {
        Circle::draw();
        for (auto p : eyes)
            p->draw();
        mouth->draw();
    }
```

Note the way that **Smiley** keeps its eyes in a standard-library **vector** and deletes them in its destructor. **Shape**'s destructor is **virtual** and **Smiley**'s destructor overrides it. A virtual destructor is

essential for an abstract class because an object of a derived class is usually manipulated through the interface provided by its abstract base class. In particular, it may be deleted through a pointer to a base class. Then, the virtual function call mechanism ensures that the proper destructor is called. That destructor then implicitly invokes the destructors of its bases and members.

In this simplified example, it is the programmer's task to place the eyes and mouth appropriately within the circle representing the face.

We can add data members, operations, or both as we define a new class by derivation. This gives great flexibility with corresponding opportunities for confusion and poor design.

4.5.1 Explicit Overriding

A function in a derived class overrides a virtual function in a base class if that function has exactly the same name and type. In large hierachies, it is not always obvious if overriding was intended. A function with a slightly different name or a slightly different type may be intended to override or it may be intended to be a separate function. To avoid confusion in such cases, a programmer can explicitly state that a function is meant to override. For example, I could (equivalently) have defined **Smiley** like this:

```
class Smiley : public Circle {   // use the circle as the base for a face
public:
        Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

        ~Smiley()
        {
                delete mouth;
                for (auto p : eyes)
                        delete p;
        }

        void move(Point to) override;

        void draw() const override;
        void rotate(int) override;

        void add_eye(Shape* s) { eyes.push_back(s); }
        void set_mouth(Shape* s);
        virtual void wink(int i);           // wink eye number i

        // ...

private:
        vector<Shape*> eyes;            // usually two eyes
        Shape* mouth;
};
```

Now, had I mistyped **move** as **mve**, I would have gotten an error because no base of **Smiley** has a virtual function called **mve**. Similarly, had I added **override** to the declaration of **wink()**, I would have gotten an error message.

4.5.2 Benefits from Hierarchies

A class hierarchy offers two kinds of benefits:

- *Interface inheritance*: An object of a derived class can be used wherever an object of a base class is required. That is, the base class acts as an interface for the derived class. The **Container** and **Shape** classes are examples. Such classes are often abstract classes.
- *Implementation inheritance*: A base class provides functions or data that simplifies the implementation of derived classes. **Smiley**'s uses of **Circle**'s constructor and of **Circle::draw()** are examples. Such base classes often have data members and constructors.

Concrete classes – especially classes with small representations – are much like built-in types: we define them as local variables, access them using their names, copy them around, etc. Classes in class hierarchies are different: we tend to allocate them on the free store using **new**, and we access them through pointers or references. For example, consider a function that reads data describing shapes from an input stream and constructs the appropriate **Shape** objects:

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)    // read shape descriptions from input stream is
{
    // ... read shape header from is and find its Kind k ...

    switch (k) {
    case Kind::circle:
        // read circle data {Point,int} into p and r
        return new Circle{p,r};
    case Kind::triangle:
        // read triangle data {Point,Point,Point} into p1, p2, and p3
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1 ,e2, and m
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
    }
}
```

A program may use that shape reader like this:

```
void user()
{
    std::vector<Shape*> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);                    // call draw() for each element
    rotate_all(v,45);               // call rotate(45) for each element
    for (auto p : v)                // remember to delete elements
        delete p;
}
```

Obviously, the example is simplified – especially with respect to error handling – but it vividly illustrates that **user()** has absolutely no idea of which kinds of shapes it manipulates. The **user()** code can be compiled once and later used for new **Shape**s added to the program. Note that there are no pointers to the shapes outside **user()**, so **user()** is responsible for deallocating them. This is done with the **delete** operator and relies critically on **Shape**'s virtual destructor. Because that destructor is virtual, **delete** invokes the destructor for the most derived class. This is crucial because a derived class may have acquired all kinds of resources (such as file handles, locks, and output streams) that need to be released. In this case, a **Smiley** deletes its **eyes** and **mouth** objects.

4.5.3 Hierarchy Navigation

The **read_shape()** function returns **Shape*** so that we can treat all **Shapes** alike. However, what can we do if we want to use a member function that is only provided by a particular derived class, such as **Smiley**'s **wink()**? We can ask "is this **Shape** a kind of **Smiley**?" using the **dynamic_cast** operator:

```
Shape* ps {read_shape(cin)};

if (Smiley* p = dynamic_cast<Smiley*>(ps)) {
        // ... is the Smiley pointer to by p ...
}
else {
        // ... not a Smiley, try something else ...
}
```

If the object pointed to by the argument of **dynamic_cast** (here, **ps**) is not of the expected type (here, **Smiley**) or a class derived from the expected type, **dynamic_cast** returns **nullptr**.

We use **dynamic_cast** to a pointer type when a pointer to an object of a different derived class is a valid argument. We then test whether the result is **nullptr**. This test can often conveniently be placed in the initialization of a variable in a condition.

When a different type is unacceptable, we can simply **dynamic_cast** to a reference type. If the object is not of the expected type, **bad_cast** is thrown:

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)};     // somewhere, catch std::bad_cast
```

Code is cleaner when **dynamic_cast** is used with restraint. If we can avoid using type information, we can write simpler and more efficient code, but occasionally type information is lost and must be recovered. This typically happens when we pass an object to some system that accepts an interface specified by a base class. When that system later passes the object back to use, we might have to recover the original type. Operations similar to **dynamic_cast** are known as "is kind of" and "is instance of" operations.

4.5.4 Avoiding Resource Leaks

Experienced programmers will notice that I left open two obvious opportunities for mistakes:

- A user might fail to **delete** the pointer returned by **read_shape()**.
- The owner of a container of **Shape** pointers might not **delete** the objects pointed to.

In that sense, functions returning a pointer to an object allocated on the free store are dangerous.

One solution to both problems is to return a standard-library **unique_ptr** (§11.2.1) rather than a "naked pointer" and store **unique_ptr**s in the container:

```
unique_ptr<Shape> read_shape(istream& is) // read shape descriptions from input stream is
{
    // read shape header from is and find its Kind k

    switch (k) {
    case Kind::circle:
        // read circle data {Point,int} into p and r
        return unique_ptr<Shape>{new Circle{p,r}};        // §11.2.1
    // ...
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);                    // call draw() for each element
    rotate_all(v,45);               // call rotate(45) for each element
} // all Shapes implicitly destroyed
```

Now the object is owned by the **unique_ptr** which will **delete** the object when it is no longer needed, that is, when its **unique_ptr** goes out of scope.

For the **unique_ptr** version of **user()** to work, we need versions of **draw_all()** and **rotate_all()** that accept **vector<unique_ptr<Shape>>**s. Writing many such **_all()** functions could become tedious, so §5.5 shows an alternative.

4.6 Copy and Move

By default, objects can be copied. This is true for objects of user-defined types as well as for built-in types. The default meaning of copy is memberwise copy: copy each member. For example, using **complex** from §4.2.1:

```
void test(complex z1)
{
    complex z2 {z1};    // copy initialization
    complex z3;
    z3 = z2;            // copy assignment
    // ...
}
```

Now **z1**, **z2**, and **z3** have the same value because both the assignment and the initialization copied both members.

When we design a class, we must always consider if and how an object might be copied. For simple concrete types, memberwise copy is often exactly the right semantics for copy. For some sophisticated concrete types, such as **Vector**, memberwise copy is not the right semantics for copy, and for abstract types it almost never is.

4.6.1 Copying Containers

When a class is a *resource handle* – that is, when the class is responsible for an object accessed through a pointer – the default memberwise copy is typically a disaster. Memberwise copy would violate the resource handle's invariant (§3.4.2). For example, the default copy would leave a copy of a **Vector** referring to the same elements as the original:

```
void bad_copy(Vector v1)
{
    Vector v2 = v1;     // copy v1's representation into v2
    v1[0] = 2;          // v2[0] is now also 2!
    v2[1] = 3;          // v1[1] is now also 3!
}
```

Assuming that **v1** has four elements, the result can be represented graphically like this:

Fortunately, the fact that **Vector** has a destructor is a strong hint that the default (memberwise) copy semantics is wrong and the compiler should at least warn against this example. We need to define better copy semantics.

Copying of an object of a class is defined by two members: a *copy constructor* and a *copy assignment*:

```
class Vector {
private:
    double* elem;  // elem points to an array of sz doubles
    int sz;
public:
    Vector(int s);                          // constructor: establish invariant, acquire resources
    ~Vector() { delete[] elem; }            // destructor: release resources

    Vector(const Vector& a);                // copy constructor
    Vector& operator=(const Vector& a);     // copy assignment

    double& operator[](int i);
    const double& operator[](int i) const;

    int size() const;
};
```

A suitable definition of a copy constructor for **Vector** allocates the space for the required number of elements and then copies the elements into it, so that after a copy each **Vector** has its own copy of the elements:

```
Vector::Vector(const Vector& a)     // copy constructor
        :elem{new double[a.sz]},    // allocate space for elements
         sz{a.sz}
{
        for (int i=0; i!=sz; ++i)   // copy elements
                elem[i] = a.elem[i];
}
```

The result of the **v2=v1** example can now be presented as:

Of course, we need a copy assignment in addition to the copy constructor:

```
Vector& Vector::operator=(const Vector& a)      // copy assignment
{
        double* p = new double[a.sz];
        for (int i=0; i!=a.sz; ++i)
                p[i] = a.elem[i];
        delete[] elem;          // delete old elements
        elem = p;
        sz = a.sz;
        return *this;
}
```

The name **this** is predefined in a member function and points to the object for which the member function is called.

4.6.2 Moving Containers

We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. We avoid the cost of copying when we pass objects to a function by using references, but we can't return a reference to a local object as the result (the local object would be destroyed by the time the caller got a chance to look at it). Consider:

```
Vector operator+(const Vector& a, const Vector& b)
{
        if (a.size()!=b.size())
                throw Vector_size_mismatch{};

        Vector res(a.size());
        for (int i=0; i!=a.size(); ++i)
                res[i]=a[i]+b[i];
        return res;
}
```

Returning from a + involves copying the result out of the local variable res and into some place where the caller can access it. We might use this + like this:

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
    Vector r;
    // ...
    r = x+y+z;
    // ...
}
```

That would be copying a Vector at least twice (one for each use of the + operator). If a Vector is large, say, 10,000 doubles, that could be embarrassing. The most embarrassing part is that res in operator+() is never used again after the copy. We didn't really want a copy; we just wanted to get the result out of a function: we wanted to *move* a Vector rather than to *copy* it. Fortunately, we can state that intent:

```
class Vector {
    // ...

    Vector(const Vector& a);                // copy constructor
    Vector& operator=(const Vector& a);     // copy assignment

    Vector(Vector&& a);                     // move constructor
    Vector& operator=(Vector&& a);          // move assignment
};
```

Given that definition, the compiler will choose the *move constructor* to implement the transfer of the return value out of the function. This means that r=x+y+z will involve no copying of Vectors. Instead, Vectors are just moved.

As is typical, Vector's move constructor is trivial to define:

```
Vector::Vector(Vector&& a)
    :elem{a.elem},          // "grab the elements" from a
     sz{a.sz}
{
    a.elem = nullptr;       // now a has no elements
    a.sz = 0;
}
```

The && means "rvalue reference" and is a reference to which we can bind an rvalue. The word "rvalue" is intended to complement "lvalue," which roughly means "something that can appear on the left-hand side of an assignment." So an rvalue is – to a first approximation – a value that you can't assign to, such as an integer returned by a function call. Thus, an rvalue reference is a reference to something that *nobody else* can assign to, so that we can safely "steal" its value. The res local variable in operator+() for Vectors is an example.

A move constructor does *not* take a const argument: after all, a move constructor is supposed to remove the value from its argument. A *move assignment* is defined similarly.

A move operation is applied when an rvalue reference is used as an initializer or as the right-hand side of an assignment.

After a move, a moved-from object should be in a state that allows a destructor to be run. Typically, we should also allow assignment to a moved-from object.

Where the programmer knows that a value will not be used again, but the compiler can't be expected to be smart enough to figure that out, the programmer can be specific:

```
Vector f()
{
    Vector x(1000);
    Vector y(1000);
    Vector z(1000);
    z = x;                  // we get a copy
    y = std::move(x);       // we get a move
    return z;               // we get a move
};
```

The standard-library function **move()** returns doesn't actually move anything. Instead, it returns a reference to its argument from which we may move – an *rvalue reference*.

Just before the **return** we have:

When **z** is destroyed, it too has been moved from (by the **return**) so that, like **x**, it is empty (it holds no elements).

4.6.3 Essential Operations

Construction of objects plays a key role in many designs. This wide variety of uses is reflected in the range and flexibility of the language features supporting initialization.

Constructors, destructors, and copy and move operations for a type are not logically separate. We must define them as a matched set or suffer logical or performance problems. If a class **X** has a destructor that performs a nontrivial task, such as free-store deallocation or lock release, the class is likely to need the full complement of functions:

```
class X {
public:
    X(Sometype);                // "ordinary constructor": create an object
    X();                        // default constructor
    X(const X&);                // copy constructor
    X(X&&);                     // move constructor
    X& operator=(const X&);     // copy assignment: clean up target and copy
    X& operator=(X&&);          // move assignment: clean up target and move
    ~X();                       // destructor: clean up
    // ...
};
```

There are five situations in which an object is copied or moved:

- As the source of an assignment
- As an object initializer
- As a function argument
- As a function return value
- As an exception

In all cases, the copy or move constructor will be applied (unless it can be optimized away).

In addition to the initialization of named objects and objects on the free store, constructors are used to initialize temporary objects and to implement explicit type conversion.

Except for the "ordinary constructor," these special member functions will be generated by the compiler as needed. If you want to be explicit about generating default implementations, you can:

```
class Y {
Public:
    Y(Sometype);
    Y(const Y&) = default;    // I really do want the default copy constructor
    Y(Y&&) = default;         // and the default copy constructor
    // ...
};
```

If you are explicit about some defaults, other default definitions will not be generated.

When a class has a pointer or a reference member, it is usually a good idea to be explicit about copy of move operations. The reason is that a pointer or reference will point to something that the class needs to delete, in which case the default copy would be wrong, or it points to something that the class must not delete, in which case a reader of the code would like to know that.

A constructor taking a single argument defines a conversion from its argument type. For example, **complex** (§4.2.1) provides a constructor from a **double**:

```
complex z1 = 3.14;  // z1 becomes {3.14,0.0}
complex z2 = z1*2;  // z2 becomes {6.28,0.0}
```

Obviously, this is sometimes ideal, but not always. For example, **Vector** (§4.2.2) provides a constructor from an **int**:

```
Vector v1 = 7;  // OK: v1 has 7 elements
```

This is typically considered unfortunate, and the standard-library **vector** does not allow this **int**-to-**vector** "conversion."

The way to avoid this problem is to say that only explicit "conversion" is allowed; that is, we can define the constructor like this:

```
class Vector {
public:
    explicit Vector(int s);    // no implicit conversion from int to Vector
    // ...
};
```

That gives us:

```
Vector v1(7);   // OK: v1 has 7 elements
Vector v2 = 7;  // error: no implicit conversion from int to Vector
```

When it comes to conversions, more types are like **Vector** than are like **complex**, so use **explicit** for constructors that take a single argument unless there is a good reason not to.

4.6.4 Resource Management

By defining constructors, copy operations, move operations, and a destructor, a programmer can provide complete control of the lifetime of a contained resource (such as the elements of a container). Furthermore, a move constructor allows an object to move simply and cheaply from one scope to another. That way, objects that we cannot or would not want to copy out of a scope can be simply and cheaply moved out instead. Consider a standard-library **thread** representing a concurrent activity (§13.2) and a **Vector** of a million **double**s. We can't copy the former and don't want to copy the latter.

```
std::vector<thread> my_threads;

Vector init(int n)
{
    thread t {heartbeat};                // run heartbeat concurrently (on its own thread)
    my_threads.push_back(move(t));       // move t into my_threads
    // ... more initialization ...

    Vector vec(n);
    for (int i=0; i<vec.size(); ++i)
        vec[i] = 777;
    return vec;                          // move res out of init()
}

auto v = init(10000);        // start heartbeat and initialize v
```

This makes resource handles, such as **Vector** and **thread**, an alternative to using pointers in many cases. In fact, the standard-library "smart pointers," such as **unique_ptr**, are themselves resource handles (§11.2.1).

I used the standard-library **vector** to hold the **thread**s because we don't get to parameterize **Vector** with an element type until §5.2.

In very much the same way as **new** and **delete** disappear from application code, we can make pointers disappear into resource handles. In both cases, the result is simpler and more maintainable code, without added overhead. In particular, we can achieve *strong resource safety*; that is, we can eliminate resource leaks for a general notion of a resource. Examples are **vector**s holding memory, **thread**s holding system threads, and **fstream**s holding file handles.

In many languages, resource management is primarily delegated to a garbage collector. C++ also offers a garbage collection interface so that you can plug in a garbage collector. However, I consider garbage collection the last alternative after cleaner, more general, and better localized alternatives to resource management have been exhausted.

Garbage collection is fundamentally a global memory management scheme. Clever implementations can compensate, but as systems are getting more distributed (think multicores, caches, and clusters), locality is more important than ever.

Also, memory is not the only resource. A resource is anything that has to be acquired and (explicitly or implicitly) released after use. Examples are memory, locks, sockets, file handles, and thread handles. A good resource management system handles all kinds of resources. Leaks must be avoided in any long-running systems, but excessive resource retention can be almost as bad as a leak. For example, if a system holds on to memory, locks, files, etc., for twice as long, the system needs to be provisioned with potentially twice as many resources.

Before resorting to garbage collection, systematically use resource handles: Let each resource have an owner in some scope and by default be released at the end of its owners scope. In C++, this is known as RAII (*Resource Acquisition Is Initialization*) and is integrated with error handling in the form of exceptions. Resources can be moved from scope to scope using move semantics or "smart pointers," and shared ownership can be represented by "shared pointers" (§11.2.1).

In the C++ standard library, RAII is pervasive: for example, memory (**string**, **vector**, **map**, **unordered_map**, etc.), files (**ifstream**, **ofstream**, etc.), threads (**thread**), locks (**lock_guard**, **unique_lock**, etc.), and general objects (through **unique_ptr** and **shared_ptr**). The result is implicit resource management that is invisible in common use and leads to low resource retention durations.

4.6.5 Suppressing Operations

Using the default copy or move for a class in a hierarchy is typically a disaster: given only a pointer to a base, we simply don't know what members the derived class has (§4.3), so we can't know how to copy them. So, the best thing to do is usually to *delete* the default copy and move operations, that is, to eliminate the default definitions of those two operations:

```
class Shape {
public:
    Shape(const Shape&) =delete;                    // no copy operations
    Shape& operator=(const Shape&) =delete;

    Shape(Shape&&) =delete;                         // no move operations
    Shape& operator=(Shape&&) =delete;

    ~Shape();
    // ...
};
```

Now an attempt to copy a **Shape** will be caught by the compiler. If you need to copy an object in a class hierarchy, write a **virtual** clone function.

In this particular case, if you forgot to **delete** a copy or move operation, no harm is done. A move operation is *not* implicitly generated for a class where the user has explicitly declared a destructor, so you get a compiler error if you try to move a **Shape**. Furthermore, the generation of copy operations is deprecated in this case (§14.2.3), so you should expect the compiler to issue a warning if you try to copy a **Shape**.

A base class in a class hierarchy is just one example of an object we wouldn't want to copy. A resource handle generally cannot be copied just by copying its members (§4.6.1).

The **=delete** mechanism is general, that is, it can be used to suppress any operation.

4.7 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 16-22 of [Stroustrup,2013].
[2] Express ideas directly in code; §4.1.
[3] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §4.2.
[4] Use concrete classes to represent simple concepts and performance-critical components; §4.2.
[5] Define a constructor to handle initialization of objects; §4.2.1, §4.6.3.
[6] Make a function a member only if it needs direct access to the representation of a class; §4.2.1.
[7] Define operators primarily to mimic conventional usage; §4.2.1.
[8] Use nonmember functions for symmetric operators; §4.2.1.
[9] Declare a member function that does not modify the state of its object **const**; §4.2.1.
[10] If a constructor acquires a resource, its class needs a destructor to release the resource; §4.2.2.
[11] Avoid "naked" **new** and **delete** operations; §4.2.2.
[12] Use resource handles and RAII to manage resources; §4.2.2.
[13] If a class is a container, give it an initializer-list constructor; §4.2.3.
[14] Use abstract classes as interfaces when complete separation of interface and implementation is needed; §4.3.
[15] Access polymorphic objects through pointers and references; §4.3.
[16] An abstract class typically doesn't need a constructor; §4.3.
[17] Use class hierarchies to represent concepts with inherent hierarchical structure; §4.5.
[18] A class with a virtual function should have a virtual destructor; §4.5.
[19] Use **override** to make overriding explicit in large class hierarchies; §4.5.1.
[20] When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance; §4.5.2.
[21] Use **dynamic_cast** where class hierarchy navigation is unavoidable; §4.5.3.
[22] Use **dynamic_cast** to a reference type when failure to find the required class is considered a failure; §4.5.3.
[23] Use **dynamic_cast** to a pointer type when failure to find the required class is considered a valid alternative; §4.5.3.
[24] Use **unique_ptr** or **shared_ptr** to avoid forgetting to **delete** objects created using **new**; §4.5.4.
[25] Redefine or prohibit copying if the default is not appropriate for a type; §4.6.1, §4.6.5.
[26] Return containers by value (relying on move for efficiency); §4.6.2.
[27] For large operands, use **const** reference argument types; §4.6.2.
[28] If a class has a destructor, it probably needs user-defined or deleted copy and move operations; §4.6.5.
[29] Control construction, copy, move, and destruction of objects; §4.6.3.
[30] Design constructors, assignments, and the destructor as a matched set of operations; §4.6.3.
[31] If a default constructor, assignment, or destructor is appropriate, let the compiler generate it (don't rewrite it yourself); §4.6.3.

[32] By default, declare single-argument constructors **explicit**; §4.6.3.

[33] If a class has a pointer or reference member, it probably needs a destructor and non-default copy operations; §4.6.3.

[34] Provide strong resource safety; that is, never leak anything that you think of as a resource; §4.6.4.

[35] If a class is a resource handle, it needs a constructor, a destructor, and non-default copy operations; §4.6.4.

This page intentionally left blank

5

Templates

Your quote here.
– B. Stroustrup

- Introduction
- Parameterized Types
- Function Templates
- Concepts and Generic Programming
- Function Objects
- Variadic Templates
- Aliases
- Template Compilation Model
- Advice

5.1 Introduction

Someone who wants a vector is unlikely always to want a vector of **double**s. A vector is a general concept, independent of the notion of a floating-point number. Consequently, the element type of a vector ought to be represented independently. A *template* is a class or a function that we parameterize with a set of types or values. We use templates to represent concepts that are best understood as something very general from which we can generate specific types and functions by specifying arguments, such as the element type **double**.

5.2 Parameterized Types

We can generalize our vector-of-doubles type to a vector-of-anything type by making it a **template** and replacing the specific type **double** with a parameter. For example:

```
template<typename T>
class Vector {
private:
    T* elem;  // elem points to an array of sz elements of type T
    int sz;
public:
    explicit Vector(int s);          // constructor: establish invariant, acquire resources
    ~Vector() { delete[] elem; }     // destructor: release resources

    // ... copy and move operations ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const { return sz; }
};
```

The `template<typename T>` prefix makes `T` a parameter of the declaration it prefixes. It is C++'s version of the mathematical "for all T" or more precisely "for all types T." Using `class` to introduce a type parameter is equivalent to using `typename`, and in older code we often see `template<class T>` as the prefix.

The member functions might be defined similarly:

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0)
        throw Negative_size{};
    elem = new T[s];
    sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Given these definitions, we can define `Vector`s like this:

```
Vector<char> vc(200);          // vector of 200 characters
Vector<string> vs(17);         // vector of 17 strings
Vector<list<int>> vli(45);     // vector of 45 lists of integers
```

The `>>` in `Vector<list<int>>` terminates the nested template arguments; it is not a misplaced input operator. It is not (as in C++98) necessary to place a space between the two `>`s.

We can use `Vector`s like this:

```
void write(const Vector<string>& vs)          // Vector of some strings
{
        for (int i = 0; i!=vs.size(); ++i)
                cout << vs[i] << '\n';
}
```

To support the range-**for** loop for our **Vector**, we must define suitable **begin()** and **end()** functions:

```
template<typename T>
T* begin(Vector<T>& x)
{
        return x.size() ? &x[0] : nullptr;      // pointer to first element or nullptr
}

template<typename T>
T* end(Vector<T>& x)
{
        return begin(x)+x.size();          // pointer to one-past-last element
}
```

Given those, we can write:

```
void f2(Vector<string>& vs)    // Vector of some strings
{
        for (auto& s : vs)
                cout << s << '\n';
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), unordered maps (that is, hash tables), etc., as templates (Chapter 9).

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to hand-crafted code. In fact, the code generated for **Vector<double>** is identical to the code generated for the version of **Vector** from Chapter 4. Furthermore, the code generated for the standard-library **vector<double>** is likely to be better (because more effort has gone into its implementation).

In addition to type arguments, a template can take value arguments. For example:

```
template<typename T, int N>
struct Buffer {
        using value_type = T;
        constexpr int size() { return N; }
        T[N];
        // ...
};
```

The alias (**value_type**) and the **constexpr** function are provided to allow users (read-only) access to the template arguments.

Value arguments are useful in many contexts. For example, **Buffer** allows us to create arbitrarily sized buffers with no overheads from the use of free store (dynamic memory):

```
Buffer<char,1024> glob;  // global buffer of characters (statically allocated)

void fct()
{
        Buffer<int,10> buf;  // local buffer of integers (on the stack)
        // ...
}
```

A template value argument must be a constant expression.

5.3 Function Templates

Templates have many more uses than simply parameterizing a container with an element type. In particular, they are extensively used for parameterization of both types and algorithms in the standard library (§9.6, §10.6). For example, we can write a function that calculates the sum of the element values of any container like this:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
        for (auto x : c)
                v+=x;
        return v;
}
```

The **Value** template argument and the function argument **v** are there to allow the caller to specify the type and initial value of the accumulator (the variable in which to accumulate the sum):

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
        int x = sum(vi,0);                          // the sum of a vector of ints (add ints)
        double d = sum(vi,0.0);                     // the sum of a vector of ints (add doubles)
        double dd = sum(ld,0.0);                    // the sum of a list of doubles
        auto z = sum(vc,complex<double>{});         // the sum of a vector of complex<double>
                                                    // the initial value is {0.0,0.0}
}
```

The point of adding **int**s in a **double** would be to gracefully handle a number larger than the largest **int**. Note how the types of the template arguments for **sum<T,V>** are deduced from the function arguments. Fortunately, we do not need to explicitly specify those types.

This **sum()** is a simplified version of the standard-library **accumulate()** (§12.3).

5.4 Concepts and Generic Programming

What are templates for? In other words, what programming techniques are effective when you use templates? Templates offer:

- The ability to pass types (as well as values and templates) as arguments without loss of information. This implies excellent opportunities for inlining, of which current

implementations take great advantage.

- Delayed type checking (done at instantiation time). This implies opportunities to weave together information from different contexts.
- The ability to pass constant values as arguments. This implies the ability to do compile-time computation.

In other words, templates provide a powerful mechanism for compile-time computation and type manipulation that can lead to very compact and efficient code. Remember that types (classes) can contain both code and values.

The first and most common use of templates is to support *generic programming*, that is, programming focused on the design, implementation, and use of general algorithms. Here, "general" means that an algorithm can be designed to accept a wide variety of types as long as they meet the algorithm's requirements on its arguments. The template is C++'s main support for generic programming. Templates provide (compile-time) parametric polymorphism.

Consider the **sum()** from §5.3. It can be invoked for any data structure that supports **begin()** and **end()** so that the range-**for** will work. Such structures include the standard-library **vector**, **list**, and **map.** Furthermore, the element type of the data structure is limited only by its use: it must be a type that we can add to the **Value** argument. Examples are **int**s, **double**s, and **Matrix**es (for any reasonable definition of **Matrix**). We could say that the **sum()** algorithm is generic in two dimensions: the type of the data structure used to store elements ("the container") and the type of elements.

So, **sum()** requires that its first template argument is some kind of container and its second template argument is some kind of number. We call such requirements *concepts*. Unfortunately, concepts cannot be expressed directly in C++11. All we can say is that the template argument for **sum()** must be types. There are techniques for checking concepts and proposals for direct language support for concepts [Stroustrup,2013] [Sutton,2012], but both are beyond the scope of this thin book.

Good, useful concepts are fundamental and are discovered more than they are designed. Examples are integer and floating-point number (as defined even in Classic C), more general mathematical concepts such as field and vector space, and container. They represent the fundamental concepts of a field of application. Identifying and formalizing to the degree necessary for effective generic programming can be a challenge.

For basic use, consider the concept *Regular*. A type is regular when it behaves much like an **int** or a **vector**. An object of a regular type

- can be default constructed.
- can be copied (with the usual semantics of copy yielding two objects that are independent and compare equal) using a constructor or an assignment.
- can be compared using **==** and **!=**.
- doesn't suffer technical problems from overly clever programming tricks.

A **string** is another example of a regular type. Like **int**, **string** is also *Ordered*. That is, two strings can be compared using **<**, **<=**, **>**, and **>=** with the appropriate semantics. Concepts is not just a syntactic notion, it is fundamentally about semantics. For example, don't define **+** to divide; that would not match the requirements for any reasonable number.

5.5 Function Objects

One particularly useful kind of template is the *function object* (sometimes called a *functor*), which is used to define objects that can be called like functions. For example:

```
template<typename T>
class Less_than {
    const T val;    // value to compare against
public:
    Less_than(const T& v) :val(v) { }
    bool operator()(const T& x) const { return x<val; } // call operator
};
```

The function called `operator()` implements the "function call," "call," or "application" operator `()`.

We can define named variables of type `Less_than` for some argument type:

```
Less_than<int> lti {42};              // lti(i) will compare i to 42 using < (i<42)
Less_than<string> lts {"Backus"};  // lts(s) will compare s to "Backus" using < (s<"Backus")
```

We can call such an object, just as we call a function:

```
void fct(int n, const string & s)
{
    bool b1 = lti(n);     // true if n<42
    bool b2 = lts(s);     // true if s<"Backus"
    // ...
}
```

Such function objects are widely used as arguments to algorithms. For example, we can count the occurrences of values for which a predicate returns `true`:

```
template<typename C, typename P>
int count(const C& c, P pred)
{
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

A *predicate* is something that we can invoke to return `true` or `false`. For example:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
        << ": " << count(vec,Less_than<int>{x})
        << '\n';
    cout << "number of values less than " << s
        << ": " << count(lst,Less_than<string>{s})
        << '\n';
}
```

Here, `Less_than<int>{x}` constructs an object for which the call operator compares to the `int` called `x`;

Less_than<string>{s} constructs an object that compares to the string called s. The beauty of these function objects is that they carry the value to be compared against with them. We don't have to write a separate function for each value (and each type), and we don't have to introduce nasty global variables to hold values. Also, for a simple function object like Less_than inlining is simple, so that a call of Less_than is far more efficient than an indirect function call. The ability to carry data plus their efficiency make function objects particularly useful as arguments to algorithms.

Function objects used to specify the meaning of key operations of a general algorithm (such as Less_than for count()) are often referred to as *policy objects*.

We have to define Less_than separately from its use. That could be seen as inconvenient. Consequently, there is a notation for implicitly generating function objects:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "number of values less than " << x
        << ": " << count(vec,[&](int a){ return a<x; })
        << '\n';
    cout << "number of values less than " << s
        << ": " << count(lst,[&](const string& a){ return a<s; })
        << '\n';
}
```

The notation [&](int a){ return a<x; } is called a *lambda expression*. It generates a function object exactly like Less_than<int>{x}. The [&] is a *capture list* specifying that local names used (such as x) will be accessed through references. Had we wanted to "capture" only x, we could have said so: [&x]. Had we wanted to give the generated object a copy of x, we could have said so: [=x]. Capture nothing is [ ], capture all local names used by reference is [&], and capture all local names used by value is [=].

Using lambdas can be convenient and terse, but also obscure. For nontrivial actions (say, more than a simple expression), I prefer to name the operation so as to more clearly state its purpose and to make it available for use in several places in a program.

In §4.5.4, we noted the annoyance of having to write many functions to perform operations on elements of vectors of pointers and unique_ptrs, such as draw_all() and rotate_all(). Function objects (in particular, lambdas) can help by allowing us to separate the traversal of the container from the specification of what is to be done with each element.

First, we need a function that applies an operation to each object pointed to by the elements of a container of pointers:

```
template<typename C, typename Oper>
void for_all(C& c, Oper op)          // assume that C is a container of pointers
{
    for (auto& x : c)
        op(*x);          // pass op() a reference to each element pointed to
}
```

Now, we can write a version of user() from §4.5 without writing a set of _all functions:

```
void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        for_all(v,[](Shape& s){ s.draw(); });          // draw_all()
        for_all(v,[](Shape& s){ s.rotate(45); });      // rotate_all(45)
}
```

I pass a reference to **Shape** to a lambda so that the lambda doesn't have to care exactly how the objects are stored in the container. In particular, those **for_all()** calls would still work if I changed **v** to a **vector<Shape*>**.

5.6 Variadic Templates

A template can be defined to accept an arbitrary number of arguments of arbitrary types. Such a template is called a *variadic template*. For example:

```
void f() { }        // do nothing

template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
        g(head);   // do something to head
        f(tail...);   // try again with tail
}
```

The key to implementing a variadic template is to note that when you pass a list of arguments to it, you can separate the first argument from the rest. Here, we do something to the first argument (the **head**) and then recursively call **f()** with the rest of the arguments (the **tail**). The ellipsis, **...**, is used to indicate "the rest" of a list. Eventually, of course, **tail** will become empty and we need a separate function to deal with that.

We can call this **f()** like this:

```
int main()
{
        cout << "first: ";
        f(1,2.2,"hello");

        cout << "\nsecond: ";
        f(0.2,'c',"yuck!",0,1,2);
        cout << "\n";
}
```

This would call **f(1,2.2,"hello")**, which will call **f(2.2,"hello")**, which will call **f("hello")**, which will call **f()**. What might the call **g(head)** do? Obviously, in a real program it will do whatever we wanted done to each argument. For example, we could make it write its argument (here, **head**) to output:

```
template<typename T>
void g(T x)
{
        cout << x << " ";
}
```

Given that, the output will be:

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

It seems that **f()** is a simple variant of **printf()** printing arbitrary lists or values – implemented in three lines of code plus their surrounding declarations.

The strength of variadic templates (sometimes just called *variadics*) is that they can accept any arguments you care to give them. The weakness is that the type checking of the interface is a possibly elaborate template program.

Because of their flexibility, variadic templates are widely used in the standard library.

5.7 Aliases

Surprisingly often, it is useful to introduce a synonym for a type or a template. For example, the standard header **<cstddef>** contains a definition of the alias **size_t**, maybe:

```
using size_t = unsigned int;
```

The actual type named **size_t** is implementation-dependent, so in another implementation **size_t** may be an **unsigned long**. Having the alias **size_t** allows the programmer to write portable code.

It is very common for a parameterized type to provide an alias for types related to their template arguments. For example:

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

In fact, every standard-library container provides **value_type** as the name of its value type (Chapter 9). This allows us to write code that will work for every container that follows this convention. For example:

```
template<typename C>
using Element_type = typename C::value_type;     // the type of C's elements

template<typename Container>
void algo(Container& c)
{
        Vector<Element_type<Container>> vec;     // keep results here
        // ...
}
```

The aliasing mechanism can be used to define a new template by binding some or all template arguments. For example:

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;      // m is a Map<string,int>
```

5.8 Template Compilation Model

The type checking provided for templates checks the use of arguments in the template definition rather than against an explicit interface (in a template declaration). This provides a compile-time variant of what is often called *duck typing* (“If it walks like a duck and it quacks like a duck, it’s a duck”). Or – using more technical terminology – we operate on values, and the presence and meaning of an operation depend solely on its operand values. This differs from the alternative view that objects have types, which determine the presence and meaning of operations. Values “live” in objects. This is the way objects (e.g., variables) work in C++, and only values that meet an object’s requirements can be put into it. What is done at compile time using templates does not involve objects, only values.

The practical effect of this is that to use a template, its definition (not just its declaration) must be in scope. For example, the standard header `<vector>` holds the definition of `vector`. An unfortunate side effect is that a type error can be found uncomfortably late in the compilation process and can yield spectacularly bad error messages because the compiler found the problem by combining information from several places in the program.

5.9 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 20-29 of [Stroustrup,2013].
[2] Use templates to express algorithms that apply to many argument types; §5.1.
[3] Use templates to express containers; §5.2.
[4] Use templates to raise the level of abstraction of code; §5.2.
[5] When defining a template, first design and debug a non-template version; later generalize by adding parameters.
[6] Templates are type-safe, but checking happens too late; §5.4.
[7] A template can pass argument types without loss of information.
[8] Use function templates to deduce class template argument types; §5.3.
[9] Templates provide a general mechanism for compile-time programming; §5.4.

[10] When designing a template, carefully consider the concepts (requirements) assumed for its template arguments; §5.4.
[11] Use concepts as a design tool; §5.4.
[12] Use function objects as arguments to algoritms; §5.5.
[13] Use a lambda if you need a simple function object in one place only; §5.5.
[14] A virtual function member cannot be a template member function.
[15] Use template aliases to simplify notation and hide implementation details; §5.7.
[16] Use variadic templates when you need a function that takes a variable number of arguments of a variety of types; §5.6.
[17] Don't use variadic templates for homogeneous argument lists (prefer initializer lists for that); §5.6.
[18] To use a template, make sure its definition (not just its declaration) is in scope; §5.8.
[19] Templates offer compile-time "duck typing"; §5.8.
[20] There is no separate compilation of templates: `#include` template definitions in every translation unit that uses them.

This page intentionally left blank

6

Library Overview

Why waste time learning
when ignorance is instantaneous?
– Hobbes

- Introduction
- Standard-Library Components
- Standard-Library Headers and Namespace
- Advice

6.1 Introduction

No significant program is written in just a bare programming language. First, a set of libraries is developed. These then form the basis for further work. Most programs are tedious to write in the bare language, whereas just about any task can be rendered simple by the use of good libraries.

Continuing from Chapters 1-5, Chapters 6-13 give a quick tour of key standard-library facilities.

I very briefly present useful standard-library types, such as **string**, **ostream**, **vector**, **map**, **unique_ptr**, **thread**, **regex**, and **complex**, as well as the most common ways of using them. As in Chapters 1-5, you are strongly encouraged not to be distracted or discouraged by an incomplete understanding of details. The purpose of this chapter is to convey a basic understanding of the most useful library facilities.

The specification of the standard library is almost two thirds of the ISO C++ standard. Explore it, and prefer it to home-made alternatives. Much thought has gone into its design, more still into its implementations, and much effort will go into its maintenance and extension.

The standard-library facilities described in this book are part of every complete C++ implementation. In addition to the standard-library components, most implementations offer "graphical user interface" systems (GUIs), Web interfaces, database interfaces, etc. Similarly, most application-development environments provide "foundation libraries" for corporate or industrial "standard" development and/or execution environments. Here, I do not describe such systems and libraries.

The intent is to provide a self-contained description of C++ as defined by the standard and to keep the examples portable. Naturally, a programmer is encouraged to explore the more extensive facilities available on most systems.

6.2 Standard-Library Components

The facilities provided by the standard library can be classified like this:

- Run-time language support (e.g., for allocation and run-time type information).
- The C standard library (with very minor modifications to minimize violations of the type system).
- Strings (with support for international character sets and localization); see §7.2.
- Support for regular expression matching; see §7.3.
- I/O streams is an extensible framework for input and output to which users can add their own types, streams, buffering strategies, locales, and character sets.
- A framework of containers (such as **vector** and **map**) and algorithms (such as **find()**, **sort()**, and **merge()**); see Chapter 9 and Chapter 10. This framework, conventionally called the STL [Stepanov,1994], is extensible so users can add their own containers and algorithms.
- Support for numerical computation (such as standard mathematical functions, complex numbers, vectors with arithmetic operations, and random number generators); see §4.2.1 and Chapter 12.
- Support for concurrent programming, including **thread**s and locks; see Chapter 13. The concurrency support is foundational so that users can add support for new models of concurrency as libraries.
- Utilities to support template metaprogramming (e.g., type traits; §11.6), STL-style generic programming (e.g., **pair**; §11.3.3), and general programming (e.g., **clock**; §11.4).
- "Smart pointers" for resource management (e.g., **unique_ptr** and **shared_ptr**; §11.2.1) and an interface to garbage collectors (§4.6.4).
- Special-purpose containers, such as **array** (§11.3.1), **bitset** (§11.3.2), and **tuple** (§11.3.3).

The main criteria for including a class in the library were that:

- it could be helpful to almost every C++ programmer (both novices and experts),
- it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and
- that simple uses should be easy to learn (relative to the inherent complexity of their task).

Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

6.3 Standard-Library Headers and Namespace

Every standard-library facility is provided through some standard header. For example:

```
#include<string>
#include<list>
```

This makes the standard **string** and **list** available.

The standard library is defined in a namespace (§3.3) called **std**. To use standard library facilities, the **std::** prefix can be used:

```
std::string s {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is Peace", "Freedom is Slavery", "Ignorance is Strength"};
```

For simplicity, I will rarely use the **std::** prefix explicitly in examples. Neither will I always **#include** the necessary headers explicitly. To compile and run the program fragments here, you must **#include** the appropriate headers and make the names they declare accessible. For example:

```
#include<string>            // make the standard string facilities accessible
using namespace std;        // make std names available without std:: prefix

string s {"C++ is a general-purpose programming language"};    // OK: string is std::string
```

It is generally in poor taste to dump every name from a namespace into the global namespace. However, in this book, I use the standard library exclusively and it is good to know what it offers.

Here is a selection of standard-library headers, all supplying declarations in namespace **std**:

Selected Standard Library Headers			
<algorithm>	**copy(), find(), sort()**	Chapter 10	§iso.25
<array>	**array**	§11.3.1	§iso.23.3.2
<chrono>	**duration, time_point**	§11.4	§iso.20.11.2
<cmath>	**sqrt(), pow()**	§12.2	§iso.26.8
<complex>	**complex, sqrt(), pow()**	§12.4	§iso.26.8
<forward_list>	**forward_list**	§9.6	§iso.23.3.4
<fstream>	**fstream, ifstream, ofstream**	§8.7	§iso.27.9.1
<future>	**future, promise**	§13.7	§iso.30.6
<ios>	**hex,dec,scientific,fixed,defaultfloat**	§8.6	§iso.27.5
<iostream>	**istream, ostream, cin, cout**	Chapter 8	§iso.27.4
<map>	**map, multimap**	§9.5	§iso.23.4.4
<memory>	**unique_ptr, shared_ptr, allocator**	§11.2.1	§iso.20.6
<random>	**default_random_engine, normal_distribution**	§12.5	§iso.26.5
<regex>	**regex, smatch**	§7.3	§iso.28.8
<string>	**string, basic_string**	§7.2	§iso.21.3
<set>	**set, multiset**	§9.6	§iso.23.4.6
<sstream>	**istrstream, ostrstream**	§8.8	§iso.27.8
<stdexcept>	**length_error, out_of_range, runtime_error**	§3.4.1	§iso.19.2
<thread>	**thread**	§13.2	§iso.30.3
<unordered_map>	**unordered_map, unordered_multimap**	§9.5	§iso.23.5.4
<utility>	**move(), swap(), pair**	Chapter 11	§iso.20.1
<vector>	**vector**	§9.2	§iso.23.3.6

This listing is far from complete.

Headers from the C standard library, such as **<stdlib.h>** are provided. For each such header there is also a version with its name prefixed by **c** and the **.h** removed. This version, such as **<cstdlib>** places its declarations in the **std** namespace.

6.4 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 30 of [Stroustrup,2013].
[2] Don't reinvent the wheel; use libraries; §6.1.
[3] When you have a choice, prefer the standard library over other libraries; §6.1.
[4] Do not think that the standard library is ideal for everything; §6.1.
[5] Remember to `#include` the headers for the facilities you use; §6.3.
[6] Remember that standard-library facilities are defined in namespace `std`; §6.3.

7

Strings and Regular Expressions

Prefer the standard to the offbeat.
– Strunk & White

- Introduction
- Strings
  - **string** Implementation
- Regular Expressions
  - Searching; Regular Expression Notation; Iterators
- Advice

7.1 Introduction

Text manipulation is a major part of most programs. The C++ standard library offers a **sting** type to save most users from C-style manipulation of arrays of characters through pointers. In addition, regular expression matching is offered to help find patterns in text. The regular expressions are provided in a form similar to what is common in most modern languages. Both **string**s and **regex** objects can use a variety of character types (e.g., Unicode).

7.2 Strings

The standard library provides a **string** type to complement the string literals (§1.3). The **string** type provides a variety of useful string operations, such as concatenation. For example:

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}

auto addr = compose("dmr","bell-labs.com");
```

Here, **addr** is initialized to the character sequence **dmr@bell–labs.com**. ‘‘Addition’’ of strings means concatenation. You can concatenate a **string**, a string literal, a C-style string, or a character to a **string**. The standard **string** has a move constructor so returning even long **string**s by value is efficient (§4.6.2).

In many applications, the most common form of concatenation is adding something to the end of a **string**. This is directly supported by the **+=** operation. For example:

```
void m2(string& s1, string& s2)
{
        s1 = s1 + '\n';   // append newline
        s2 += '\n';       // append newline
}
```

The two ways of adding to the end of a **string** are semantically equivalent, but I prefer the latter because it is more explicit about what it does, more concise, and possibly more efficient.

A **string** is mutable. In addition to **=** and **+=**, subscripting (using **[ ]**), and substring operations are supported. Among other useful features, it provides the ability to manipulate substrings. For example:

```
string name = "Niels Stroustrup";

void m3()
{
        string s = name.substr(6,10);          // s = "Stroustrup"
        name.replace(0,5,"nicholas");          // name becomes "nicholas Stroustrup"
        name[0] = toupper(name[0]);            // name becomes "Nicholas Stroustrup"
}
```

The **substr()** operation returns a **string** that is a copy of the substring indicated by its arguments. The first argument is an index into the **string** (a position), and the second is the length of the desired substring. Since indexing starts from **0**, **s** gets the value **Stroustrup**.

The **replace()** operation replaces a substring with a value. In this case, the substring starting at **0** with length **5** is **Niels**; it is replaced by **nicholas**. Finally, I replace the initial character with its uppercase equivalent. Thus, the final value of **name** is **Nicholas Stroustrup**. Note that the replacement string need not be the same size as the substring that it is replacing.

Naturally, **string**s can be compared against each other and against string literals. For example:

```
string incantation;

void respond(const string& answer)
{
        if (answer == incantation) {
                // perform magic
        }
        else if (answer == "yes") {
                // ...
        }
        // ...
}
```

Among the many useful **string** operations are assignment (using **=**), subscripting (using **[ ]** or **at()** as for **vector**; §9.2.2), iteration (using iterators as for **vector**; §10.2), input (§8.3), streaming (§8.8).

If you need a C-style string (a zero-terminated array of **char**), **string** offers read-only access to its contained characters. For example:

```
void print(const string& s)
{
    printf("For people who like printf: %s\n",s.c_str());
    cout << "For people who like streams: " << s << '\n';
}
```

7.2.1 string Implementation

Implementing a string class is a popular and useful exercise. However, for general-purpose use, our carefully crafted first attempts rarely match the standard **string** in convenience or performance. These days, **string** is usually implemented using the *short-string optimization*. That is, short string values are kept in the **string** object itself and only longer strings are placed on free store. Consider:

```
string s1 {"Annemarie"};          // short string
string s2 {"Annemarie Stroustrup"};   // long string
```

The memory layout will be something like:

When a **string**'s value changes from a short to a long string (and vice verse) its representation adjusts appropriately.

The actual performance of **string**s can depend critically on the run-time environment. In particular, in multi-threaded implementations, memory allocation can be relatively costly. Also, when lots of strings of differing lengths are used, memory fragmentation can result. These are the main reasons that the short-string optimization has become ubiquitous.

To handle multipe character sets, **string** is really an alias for a general template **basic_string** with the character type **char**:

```
template<typename Char>
class basic_string {
    // ... string of Char ...
};

using string = basic_string<char>
```

A user can define strings of arbitrary character types. For example, assuming we have a Japanese character type **Jchar**, we can write:

```
using Jstring = basic_string<Jchar>;
```

Now we can do all the usual string operations on **Jstring**, a string of Japanese characters. Similarly, we can handle Unicode string.

7.3 Regular Expressions

Regular expressions are a powerful tool for text processing. They provide a way to simply and tersely describe patterns in text (e.g., a U.S. postal code such as **TX 77845**, or an ISO-style date, such as **2009–06–07**) and to efficiently find such patterns in text. In **<regex>**, the standard library provides support for regular expressions in the form of the **std::regex** class and its supporting functions. To give a taste of the style of the **regex** library, let us define and print a pattern:

```
regex pat (R"(\w{2}\s*\d{5}(–\d{4})?)");    // US postal code pattern: XXddddd-dddd and variants
```

People who have used regular expressions in just about any language will find **\w{2}\s*\d{5}(–\d{4})?** familiar. It specifies a pattern starting with two letters **\w{2}** optionally followed by some space **\s*** followed by five digits **\d{5}** and optionally followed by a dash and four digits **–\d{4}**. If you are not familiar with regular expressions, this may be a good time to learn about them ([Stroustrup,2009], [Maddock,2009], [Friedl,1997]).

To express the pattern, I use a *raw string literal* starting with **R"(** and terminated by **)"**. This allows backslashes and quotes to be used directly in the string. Raw strings are particularly suitable for regular expressions because they tend to contain a lot of backslashes. Had I used a conventional string, the pattern definition would have been:

```
regex pat {"\\w{2}\\s*\\d{5}(–\\d{4})?"};    // U.S. postal code pattern
```

In **<regex>**, the standard library provides support for regular expressions:

- **regex_match()**: Match a regular expression against a string (of known size) (§7.3.2).
- **regex_search()**: Search for a string that matches a regular expression in an (arbitrarily long) stream of data (§7.3.1).
- **regex_replace()**: Search for strings that match a regular expression in an (arbitrarily long) stream of data and replace them.
- **regex_iterator**: Iterate over matches and submatches (§7.3.3).
- **regex_token_iterator**: Iterate over non-matches.

7.3.1 Searching

The simplest way of using a pattern is to search for it in a stream:

```
int lineno = 0;
for (string line; getline(cin,line); ) {            // read into line buffer
    ++lineno;
    smatch matches;                                 // matched strings go here
    if (regex_search(line,matches,pat))             // search for pat in line
        cout << lineno << ": " << matches[0] << '\n';
}
```

The **regex_search(line,matches,pat)** searches the **line** for anything that matches the regular expression stored in **pat** and if it finds any matches, it stores them in **matches**. If no match was found,

regex_search(line,matches,pat) returns **false**. The **matches** variable is of type **smatch**. The "s" stands for "sub" or "string," and an **smatch** is a **vector** of sub-matches of type **string**. The first element, here **matches[0]**, is the complete match. The result of a **regex_search()** is a collection of matches, typically represented as an **smatch**:

```
void use()
{
    ifstream in("file.txt");      // input file
    if (!in)                      // check that the file was opened
        cerr << "no file\n";

    regex pat {R"(\w{2}\s*\d{5}(–\d{4})?)"};   // U.S. postal code pattern

    int lineno = 0;
    for (string line; getline(in,line); ) {
        ++lineno;
        smatch matches;    // matched strings go here
        if (regex_search(line, matches, pat)) {
            cout << lineno << ": " << matches[0] << '\n';          // the complete match
            if (1<matches.size() && matches[1].matched)
                cout  << "\t: " << matches[1] << '\n';             // submatch
        }
    }
}
```

This function reads a file looking for U.S. postal codes, such as **TX77845** and **DC 20500–0001**. An **smatch** type is a container of regex results. Here, **matches[0]** is the whole pattern and **matches[1]** is the optional four-digit subpattern.

The regular expression syntax and semantics are designed so that regular expressions can be compiled into state machines for efficient execution [Cox,2007]. The **regex** type performs this compilation at run time.

7.3.2 Regular Expression Notation

The **regex** library can recognize several variants of the notation for regular expressions. Here, I use the default notation used, a variant of the ECMA standard used for ECMAScript (more commonly known as JavaScript).

The syntax of regular expressions is based on characters with special meaning:

Regular Expression Special Characters			
.	Any single character (a "wildcard")	\	Next character has a special meaning
[	Begin character class	*	Zero or more (suffix operation)
]	End character class	+	One or more (suffix operation)
{	Begin count	?	Optional (zero or one) (suffix operation)
}	End count	|	Alternative (or)
(	Begin grouping	^	Start of line; negation
)	End grouping	$	End of line

For example, we can specify a line starting with zero or more **A**s followed by one or more **B**s followed by an optional **C** like this:

```
^A*B+C?$
```

Examples that match:

```
AAAAAAAAAAAABBBBBBBBBC
BC
B
```

Examples that do not match:

```
AAAAA          // no B
  AAAABC       // initial space
AABBCC         // too many Cs
```

A part of a pattern is considered a subpattern (which can be extracted separately from an **smatch**) if it is enclosed in parentheses. For example:

```
\d+–\d+        // no subpatterns
\d+(–\d+)      // one subpattern
(\d+)(–\d+)    // two subpatterns
```

A pattern can be optional or repeated (the default is exactly once) by adding a suffix:

Repetition	
{ n }	Exactly **n** times
{ n, }	**n** or more times
{n,m}	At least **n** and at most **m** times
*	Zero or more, that is, **{0,}**
+	One or more, that is, **{1,}**
?	Optional (zero or one), that is **{0,1}**

For example:

```
A{3}B{2,4}C*
```

Examples that match:

```
AAABBC
AAABBB
```

Example that do not match:

```
AABBC          // too few As
AAABC          // too few Bs
AAABBBBBCCC    // too many Bs
```

A suffix **?** after any of the repetition notations (**?**, *, **?**, and **{ }**) makes the pattern matcher "lazy" or "non-greedy." That is, when looking for a pattern, it will look for the shortest match rather than the longest. By default, the pattern matcher always looks for the longest match; this is known as the *Max Munch rule*. Consider:

ababab

The pattern **(ab)*** matches all of **ababab**. However, **(ab)*?** matches only the first **ab**.

The most common character classifications have names:

Character Classes	
alnum	Any alphanumeric character
alpha	Any alphabetic character
blank	Any whitespace character that is not a line separator
cntrl	Any control character
d	Any decimal digit
digit	Any decimal digit
graph	Any graphical character
lower	Any lowercase character
print	Any printable character
punct	Any punctuation character
s	Any whitespace character
space	Any whitespace character
upper	Any uppercase character
w	Any word character (alphanumeric characters plus the underscore)
xdigit	Any hexadecimal digit character

In a regular expression, a character class name must be bracketed by **[: :]**. For example, **[:digit:]** matches a decimal digit. Furthermore, they must be used within a **[ ]** pair defining a character class.

Several character classes are supported by shorthand notation:

Character Class Abbreviations		
\d	A decimal digit	**[[:digit:]]**
\s	A space (space, tab, etc.)	**[[:space:]]**
\w	A letter (**a**-**z**) or digit (**0**-**9**) or underscore (_)	**[_[:alnum:]]**
\D	Not **\d**	**[^[:digit:]]**
\S	Not **\s**	**[^[:space:]]**
\W	Not **\w**	**[^_[:alnum:]]**

In addition, languages supporting regular expressions often provide:

Nonstandard (but Common) Character Class Abbreviations		
\l	A lowercase character	**[[:lower:]]**
\u	An uppercase character	**[[:upper:]]**
\L	Not **\l**	**[^[:lower:]]**
\U	Not **\u**	**[^[:upper:]]**

For full portability, use the character class names rather than these abbreviations.

As an example, consider writing a pattern that describes C++ identifiers: an underscore or a letter followed by a possibly empty sequence of letters, digits, or underscores. To illustrate the

subtleties involved, I include a few false attempts:

```
[:alpha:][:alnum:]*          // wrong: characters from the set ":alph" followed by ...
[[:alpha:]][[:alnum:]]*      // wrong: doesn't accept underscore ('_' is not alpha)
([[:alpha:]]|_)[[:alnum:]]*  // wrong: underscore is not part of alnum either

([[:alpha:]]|_)([[:alnum:]]|_)*      // OK, but clumsy
[[:alpha:]_][[:alnum:]_]*            // OK: include the underscore in the character classes
[_[:alpha:]][_[:alnum:]]*            // also OK
[_[:alpha:]]\w*                      // \w is equivalent to [_[:alnum:]]
```

Finally, here is a function that uses the simplest version of **regex_match()** (§7.3.1) to test whether a string is an identifier:

```
bool is_identifier(const string& s)
{
    regex pat {"[_[:alpha:]]\\w*"};  // underscore or letter
                                     // followed by zero or more underscores, letters, or digits
    return regex_match(s,pat);
}
```

Note the doubling of the backslash to include a backslash in an ordinary string literal. Use raw string literals to alleviate problems with special characters. For example:

```
bool is_identifier(const string& s)
{
    regex pat {R"([_[:alpha:]]\w*)"};
    return regex_match(s,pat);
}
```

Here are some examples of patterns:

```
Ax*          // A, Ax, Axxxx
Ax+          // Ax, Axxx      Not A
\d-?\d       // 1-2, 12       Not 1--2
\w{2}-\d{4,5} // Ab-1234, XX-54321, 22-5432      Digits are in \w
(\d*:)?(\d+) // 12:3, 1:23, 123, :123     Not 123:
(bs|BS)      // bs, BS        Not bS
[aeiouy]     // a, o, u       An English vowel, not x
[^aeiouy]    // x, k          Not an English vowel, not e
[a^eiouy]    // a, ^, o, u    An English vowel or ^
```

A **group** (a subpattern) potentially to be represented by a **sub_match** is delimited by parentheses. If you need parentheses that should not define a subpattern, use **(?** rather than plain **(**. For example:

```
(\s|:|,)*(\d*)    // spaces, colons, and/or commas followed by a number
```

Assuming that we were not interested in the characters before the number (presumably separators), we could write:

```
(?\s|:|,)*(\d*)   // spaces, colons, and/or commas followed by a number
```

This would save the regular expression engine from having to store the first characters: the **(?** variant has only one subpattern.

Regular Expression Grouping Examples	
`\d*\s\w+`	No groups (subpatterns)
`(\d*)\s(\w+)`	Two groups
`(\d*)(\s(\w+))+`	Two groups (groups do not nest)
`(\s*\w*)+`	One group, but one or more subpatterns; only the last subpattern is saved as a `sub_match`
`<(.*?)>(.*?)</\1>`	Three groups; the `\1` means "same as group 1"

That last pattern is useful for parsing XML. It finds tag/end-of-tag markers. Note that I used a non-greedy match (a *lazy match*), `.*?`, for the subpattern between the tag and the end tag. Had I used plain `.*`, this input would have caused a problem:

```
Always look for the <b>bright</b> side of <b>life</b>.
```

A *greedy match* for the first subpattern would match the first `<` with the last `>`. A greedy match on the second subpattern would match the first `<b>` with the last `</b>`. Both would be correct behavior, but unlikely what the programmer wanted.

For a more exhaustive presentation of regular expressions, see [Friedl,1997].

7.3.3 Iterators

We can define a `regex_iterator` for iterating over a stream finding matches for a pattern. For example, we can output all whitespace-separated words in a `string`:

```
void test()
{
    string input = "aa as; asd ++e^asdf asdfg";
    regex pat {R"(\s+(\w+))"};
    for (sregex_iterator p(input.begin(),input.end(),pat); p!=sregex_iterator{}; ++p)
        cout << (*p)[1] << '\n';
}
```

This outputs:

```
as
asd
asdfg
```

Note that we are missing the first word, `aa`, because it has no preceding whitespace. If we simplify the pattern to `R"((\ew+))"`, we get

```
aa
as
asd
e
asdf
asdfg
```

A `regex_iterator` is a bidirectional iterator, so we cannot directly iterate over an `istream`. Also, we cannot write through a `regex_iterator`, and the default `regex_iterator` (`regex_iterator{}`) is the only possible end-of-sequence.

7.4 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 36-37 of [Stroustrup,2013].
[2] Prefer **string** operations to C-style string functions; §7.1.
[3] Use **string** to declare variables and members rather than as a base class; §7.2.
[4] Return **string**s by value (rely on move semantics); §7.2, §7.2.1.
[5] Directly or indirectly, use **substr()** to read substrings and **replace()** to write substrings; §7.2.
[6] A **string** can grow and shrink, as needed; §7.2.
[7] Use **at()** rather than iterators or **[ ]** when you want range checking; §7.2.
[8] Use iterators and **[ ]** rather than **at()** when you want to optimize speed; §7.2.
[9] **string** input doesn't overflow; §7.2, §8.3.
[10] Use **c_str()** to produce a C-style string representation of a **string** (only) when you have to; §7.2.
[11] Use a **string_stream** or a generic value extraction function (such as **to<X>**) for numeric conversion of strings; §8.8.
[12] A **basic_string** can be used to make strings of characters on any type; §7.2.1.
[13] Use **regex** for most conventional uses of regular expressions; §7.3.
[14] Prefer raw string literals for expressing all but the simplest patterns; §7.3.
[15] Use **regex_match()** to match a complete input; §7.3, §7.3.2.
[16] Use **regex_search()** to search for a pattern in an input stream; §7.3.1.
[17] The regular expression notation can be adjusted to match various standards; §7.3.2.
[18] The default regular expression notation is that of ECMAScript; §7.3.2.
[19] Be restrained; regular expressions can easily become a write-only language; §7.3.2.
[20] Note that **\i** allows you to express a subpattern in terms of a previous subpattern; §7.3.2.
[21] Use **?** to make patterns "lazy"; §7.3.2.
[22] Use **regex_iterator**s for iterating over a stream looking for a pattern; §7.3.3

8

I/O Streams

What you see is all you get.
– Brian W. Kernighan

- Introduction
- Output
- Input
- I/O State
- I/O of User-Defined Types
- Formatting
- File Streams
- String Streams
- Advice

8.1 Introduction

The I/O stream library provides formatted and unformatted buffered I/O of text and numeric values.

An **ostream** converts typed objects to a stream of characters (bytes):

An **istream** converts a stream of characters (bytes) to typed objects:

The operations on **istream**s and **ostream**s are described in §8.3 and §8.2. The operations are type-safe, type-sensitive, and extensible to handle user-defined types.

Other forms of user interaction, such as graphical I/O, are handled through libraries that are not part of the ISO standard and therefore not described here.

These streams can be used for binary I/O, be used for a variety of character types, be locale specific, and use advanced buffering strategies, but these topics are beyond the scope of this book.

8.2 Output

In **<ostream>**, the I/O stream library defines output for every built-in type. Further, it is easy to define output of a user-defined type (§8.5). The operator **<<** ("put to") is used as an output operator on objects of type **ostream**; **cout** is the standard output stream and **cerr** is the standard stream for reporting errors. By default, values written to **cout** are converted to a sequence of characters. For example, to output the decimal number **10**, we can write:

```
void f()
{
    cout << 10;
}
```

This places the character **1** followed by the character **0** on the standard output stream.

Equivalently, we could write:

```
void g()
{
    int i {10};
    cout << i;
}
```

Output of different types can be combined in the obvious way:

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

For **h(10)**, the output will be:

```
the value of i is 10
```

People soon tire of repeating the name of the output stream when outputting several related items. Fortunately, the result of an output expression can itself be used for further output. For example:

```
void h2(int i)
{
	cout << "the value of i is " << i << '\n';
}
```

This **h2()** produces the same output as **h()**.

A character constant is a character enclosed in single quotes. Note that a character is output as a character rather than as a numerical value. For example:

```
void k()
{
	int b = 'b';		// note: char implicitly converted to int
	char c = 'c';
	cout << 'a' << b << c;
}
```

The integer value of the character **'b'** is **98** (in the ASCII encoding used on the C++ implementation that I used), so this will output **a98c**.

8.3 Input

In **<istream>**, the standard library offers **istream**s for input. Like **ostream**s, **istream**s deal with character string representations of built-in types and can easily be extended to cope with user-defined types.

The operator **>>** ("get from") is used as an input operator; **cin** is the standard input stream. The type of the right-hand operand of **>>** determines what input is accepted and what is the target of the input operation. For example:

```
void f()
{
	int i;
	cin >> i;		// read an integer into i

	double d;
	cin >> d;		// read a double-precision floating-point number into d
}
```

This reads a number, such as **1234**, from the standard input into the integer variable **i** and a floating-point number, such as **12.34e5**, into the double-precision floating-point variable **d**.

Like output operations, input operations can be chained, so I could equivalently have written:

```
void f()
{
        int i;
        double d;
        cin >> i >> d;          // read into i and d
}
```

In both cases, the read of the integer is terminated by any character that is not a digit. By default, **>>** skips initial whitespace, so a suitable complete input sequence would be

```
1234
12.34e5
```

Often, we want to read a sequence of characters. A convenient way of doing that is to read into a **string**. For example:

```
void hello()
{
        cout << "Please enter your name\n";
        string str;
        cin >> str;
        cout << "Hello, " << str << "!\n";
}
```

If you type in **Eric** the response is:

```
Hello, Eric!
```

By default, a whitespace character, such as a space or a newline, terminates the read, so if you enter **Eric Bloodaxe** pretending to be the ill-fated king of York, the response is still:

```
Hello, Eric!
```

You can read a whole line (including the terminating newline character) using the **getline()** function. For example:

```
void hello_line()
{
        cout << "Please enter your name\n";
        string str;
        getline(cin,str);
        cout << "Hello, " << str << "!\n";
}
```

With this program, the input **Eric Bloodaxe** yields the desired output:

```
Hello, Eric Bloodaxe!
```

The newline that terminated the line is discarded, so **cin** is ready for the next input line.

The standard strings have the nice property of expanding to hold what you put in them; you don't have to precalculate a maximum size. So, if you enter a couple of megabytes of semicolons, the program will echo pages of semicolons back at you.

8.4 I/O State

An **iostream** has a state that we can examine to determine whether an operation succeeded. The most common use is to read a sequence of values:

```
vector<int> read_ints(istream& is)
{
    vector<int> res;
    int i;
    while (is>>i)
        res.push_back(i);
    return res;
}
```

This reads from **is** until something that is not an integer is encountered. That something will typically be the end of input. What is happening here is that the operation **is>>i** returns a reference to **is**, and testing an **iostream** yields **true** if the stream is ready for another operation.

In general, the I/O state holds all the information needed to read or write, such as formatting information (§8.6), error state (e.g., has end-of-input been reached?), and what kind of buffering is used. In particular, a user can set the state to reflect that an error has occurred (§8.5) and clear the state if an error wasn't serious. For example, we could imagine reading a sequence of integers than might contain some form of nesting:

```
while (cin) {
    for (int i; cin>>i; ) {
        // ... use the integer ...
    }

    if (cin.eof()) {
        // .. all is well we reached the end-of-file ...
    }
    else if (cin.fail()) {          // a potentially recoverable error
        cin.clear();                // reset the state to good()
        char ch;
        if (cin>>ch) {              // look for nesting represented by { ... }
            switch (ch) {
            case '{':
                // ... start nested structure ...
                break;
            case '}':
                // ... end nested structure ...
                break;
            default:
                cin.setstate(ios_base::failbit);        // add fail() to cin's state
            }
        }
    }
    // ...
}
```

8.5 I/O of User-Defined Types

In addition to the I/O of built-in types and standard **string**s, the **iostream** library allows programmers to define I/O for their own types. For example, consider a simple type **Entry** that we might use to represent entries in a telephone book:

```
struct Entry {
    string name;
    int number;
};
```

We can define a simple output operator to write an **Entry** using a *{"name",number}* format similar to the one we use for initialization in code:

```
ostream& operator<<(ostream& os, const Entry& e)
{
    return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

A user-defined output operator takes its output stream (by reference) as its first argument and returns it as its result.

The corresponding input operator is more complicated because it has to check for correct formatting and deal with errors:

```
istream& operator>>(istream& is, Entry& e)
    // read { "name" , number } pair. Note: formatted with { " " , and }
{
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') { // start with a { "
        string name;                    // the default value of a string is the empty string: ""
        while (is.get(c) && c!='"')     // anything before a " is part of the name
            name+=c;

        if (is>>c && c==',') {
            int number = 0;
            if (is>>number>>c && c=='}') { // read the number and a }
                e = {name,number};      // assign to the entry
                return is;
            }
        }
    }
    is.state_base::failbit);            // register the failure in the stream
    return is;
}
```

An input operation returns a reference to its **istream** which can be used to test if the operation succeeded. For example, when used as a condition, **is>>c** means "Did we succeed at reading from **is** into **c**?"

The **is>>c** skips whitespace by default, but **is.get(c)** does not, so that this **Entry**-input operator ignores (skips) whitespace outside the name string, but not within it. For example:

```
{ "John Marwood Cleese" , 123456        }
{"Michael Edward Palin",987654}
```

We can read such a pair of values from input into an **Entry** like this:

```
for (Entry ee; cin>>ee; )   // read from cin into ee
    cout << ee << '\n';     // write ee to cout
```

The output is:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

See §7.3 for a more systematic technique for recognizing patterns in streams of characters (regular expression matching).

8.6 Formatting

The **iostream** library provides a large set of operations for controlling the format of input and output. The simplest formatting controls are called *manipulators* and are found in **<ios>**, **<istream>**, **<ostream>**, and **<iomanip>** (for manipulators that take arguments): For example, we can output integers as decimal (the default), octal, or hexadecimal numbers:

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n';        // print 1234,4d2,2322
```

We can explicitly set the output format for floating-point numbers:

```
constexpr double d = 123.456;

cout << d << "; "                    // use the default format for d
     << scientific <<  d << "; "     // use 1.123e2 style format for d
     << hexfloat <<  d << "; "       // use hexadecimal notation for d
     << fixed << d << "; "           // use 123.456 style format for f
     << defaultfloat << d << '\n';   // use the default format for d
```

This produces:

```
123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456
```

Precision is an integer that determines the number of digits used to display a floating-point number:

- The *general* format (**defaultfloat**) lets the implementation choose a format that presents a value in the style that best preserves the value in the space available. The precision specifies the maximum number of digits.
- The *scientific* format (**scientific**) presents a value with one digit before a decimal point and an exponent. The precision specifies the maximum number of digits after the decimal point.
- The *fixed* format (**fixed**) presents a value as an integer part followed by a decimal point and a fractional part. The precision specifies the maximum number of digits after the decimal point.

Floating-point values are rounded rather than just truncated, and **precision()** doesn't affect integer output. For example:

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

This produces:

```
1234.5679 1234.5679 123456
1235 1235 123456
```

These manipulators as "sticky"; that is, it persists for subsequent floating-point operations.

8.7 File Streams

In **<fstream>**, the standard library provides streams to and from a file:

- **ifstream**s for reading from a file
- **ofstream**s for writing to a file
- **fstream**s for reading from and writing to a file

For example:

```
ofstream ofs("target");                    // "o" for "output"
if (!ofs)
        error("couldn't open 'target' for writing");
```

Testing that a file stream has been properly opened is usually done by checking its state.

```
fstream ifs;                               // "i" for "input"
if (!ifs)
        error("couldn't open 'source' for reading");
```

Assuming that the tests succeeded, **ofs** can be used as an ordinary **ostream** (just like **cout**) and **ifs** can be used as an ordinary **istream** (just like **cin**).

File positioning and more detailed control of the way a file is opened is possible, but beyond the scope of this book.

8.8 String Streams

In **<sstream>**, the standard library provides streams to and from a **string**:

- **istringstream**s for reading from a **string**
- **ostringstream**s for writing to a **string**
- **stringstream**s for reading from and writing to a **string**.

For example:

```
void test()
{
        ostringstream oss;
```

```
        oss << "{temperature," << scientific << 123.4567890 << "}";
        cout << oss.str() << '\n';
    }
```

The result from an **istringstream** can be read using **str()**. One common use of an **ostringstream** is to format before giving the resulting string to a GUI. Similarly, a string received from a GUI can be read using formatted input operations (§8.3) by putting it into an **istringstream**.

A **stringstream** can be used for both reading and writing. For example, we can define an operation that can convert any type with a string representation to another that also has a string representation:

```
template<typename Target =string, typename Source =string>
Target to(Source arg)           // convert Source to Target
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                   // write arg into stream
        || !(interpreter >> result)             // read result from stream
        || !(interpreter >> std::ws).eof())     // stuff left in stream?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

A function template argument needs to be explicitly mentioned only if it cannot be deduced or if there is no default, so we can write:

```
auto x1 = to<string,double>(1.2);   // very explicit (and verbose)
auto x2 = to<string>(1.2);          // Source is deduced to double
auto x3 = to<>(1.2);                // Target is defaulted to string; Source is deduced to double
auto x4 = to(1.2);                  // the <> is redundant;
                                    // Target is defaulted to string; Source is deduced to double
```

If all function template arguments are defaulted, the **<>** can be left out.

I consider this a good example of the generality and ease of use that can be achieved by a combination of language features and standard-library facilities.

8.9 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 38 of [Stroustrup,2013].

[2] **iostream**s are type-safe, type-sensitive, and extensible; §8.1.

[3] Define **<<** and **>>** for user-defined types with values that have meaningful textual representations; §8.1, §8.2, §8.3.

[4] Use **cout** for normal output and **cerr** for errors; §8.1.

[5] There are **iostream**s for ordinary characters and wide characters, and you can define an **iostream** for any kind of character; §8.1.

[6] Binary I/O is supported; §8.1.
[7] There are standard **iostream**s for standard I/O streams, files, and **string**s; §8.2, §8.3, §8.7, §8.8.
[8] Chain **<<** operations for a terser notation; §8.2.
[9] Chain **>>** operations for a terser notation; §8.3.
[10] Input into **string**s does not overflow; §8.3.
[11] By default **>>** skips initial whitespace; §8.3.
[12] Use the stream state **fail** to handle potentially recoverable I/O errors; §8.4.
[13] You can define **<<** and **>>** operators for your own types; §8.5.
[14] You don't need to modify **istream** or **ostream** to add new **<<** and **>>** operators; §8.5.
[15] Use manipulators to control formatting; §8.6.
[16] **precision()** specifications apply to all following floating-point output operations; §8.6.
[17] Floating-point format specifications (e.g., **scientific**) apply to all following floating-point output operations; §8.6.
[18] **#include <ios>** when using standard manipulators; §8.6.
[19] **#include <iomanip>** when using standard manipulators taking arguments; §8.6.
[20] Don't try to copy a file stream.
[21] Remember to check that a file stream is attached to a file before using it; §8.7.
[22] Use **stringstream**s for in-memory formatting; §8.8.
[23] You can define conversions between any two types that both have string representation; §8.8.

9

Containers

It was new.
It was singular.
It was simple.
It must succeed!
– H. Nelson

- Introduction
- **vector**
  Elements; Range Checking
- **list**
- **map**
- **unordered_map**
- Container Overview
- Advice

9.1 Introduction

Most computing involves creating collections of values and then manipulating such collections. Reading characters into a **string** and printing out the **string** is a simple example. A class with the main purpose of holding objects is commonly called a *container*. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program.

To illustrate the standard-library containers, consider a simple program for keeping names and telephone numbers. This is the kind of program for which different approaches appear "simple and obvious" to people of different backgrounds. The **Entry** class from §8.5 can be used to hold a simple phone book entry. Here, we deliberately ignore many real-world complexities, such as the fact that many phone numbers do not have a simple representation as a 32-bit **int**.

9.2 vector

The most useful standard-library container is **vector**. A **vector** is a sequence of elements of a given type. The elements are stored contiguously in memory. A typical implementation of **vector** (§4.2.2, §4.6) will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (§10.1) (or the equivalent information represented as a pointer plus offsets):

In addition, it holds an allocator (here, **alloc**), from which the **vector** can acquire memory for its elements. The default allocator uses **new** and **delete** to acquire and release memory.

We can initialize a **vector** with a set of values of its element type:

```
vector<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

Elements can be accessed through subscripting:

```
void print_book(const vector<Entry>& book)
{
        for (int i = 0; i!=book.size(); ++i)
                cout << book[i] << '\n';
}
```

As usual, indexing starts at **0** so that **book[0]** holds the entry for **David Hume**. The **vector** member function **size()** gives the number of elements.

The elements of a **vector** constitute a range, so we can use a range-**for** loop (§1.8):

```
void print_book(const vector<Entry>& book)
{
        for (const auto& x : book)      // for "auto" see §1.5
                cout << x << '\n';
}
```

When we define a **vector**, we give it an initial size (initial number of elements):

```
vector<int> v1 = {1, 2, 3, 4};          // size is 4
vector<string> v2;                      // size is 0
vector<Shape*> v3(23);                  // size is 23; initial element value: nullptr
vector<double> v4(32,9.9);              // size is 32; initial element value: 9.9
```

An explicit size is enclosed in ordinary parentheses, for example, **(23)**, and by default the elements

are initialized to the element type's default value (e.g., nullptr for pointers and 0 for numbers). If you don't want the default value, you can specify one as a second argument (e.g., 9.9 for the 32 elements of v4).

The initial size can be changed. One of the most useful operations on a vector is push_back(), which adds a new element at the end of a vector, increasing its size by one. For example:

```
void input()
{
    for (Entry e; cin>>e; )
        phone_book.push_back(e);
}
```

This reads Entrys from the standard input into phone_book until either the end-of-input (e.g., the end of a file) is reached or the input operation encounters a format error.

The standard-library vector is implemented so that growing a vector by repeated push_back()s is efficient. To show how, consider an elaboration of the simple Vector from (Chapter 4 and Chapter 5) using the representation indicated in the diagram above:

```
template<typename T>
class Vector {
    T* elem;        // pointer to first element
    T* space;       // pointer to first unused (and uninitialized) slot
    T* last;        // pointer to last slot
public:
    // ...
    int size();                         // number of elements (space-elem)
    int capacity();                     // number of slots available for elements (last-elem)
    // ...
    void reserve(int newsz);            // increase capacity() to newsz
    // ...
    void push_back(const T& t);         // copy t into Vector
    void push_back(T&& t);              // move t into Vector
};
```

The standard-libray vector has members capacity(), reserve(), and push_back(). The reserve() is used by users of vector and other vector members to make room for more elements. It may have to allocate new memory and when it does it moves the elements to the new allocation.

Given capacity() and reserve(), implementing push_back() is trivial:

```
template<typename T>
void Vector<T>::push_back(const T& t)
{
    if (capacity()<size()+1)            // make sure we have space for t
        reserve(size()==0?8:2*size());  // double the capacity
    new(space){t};                      // initialize *space to t
    ++space;
}
```

Now allocation and relocation of elements happens only infrequently. I used to use reserve() to try to improve performance, but that turned out to be a waste of effort: The heuristic used by vector is

better than my guesses, so now I only use **reserve()** to avoid rellocation of elements when I want to use pointers to elements.

A **vector** can be copied in assignments and initializations. For example:

```
vector<Entry> book2 = phone_book;
```

Copying and moving of **vector**s are implemented by constructors and assignment operators as described in §4.6. Assigning a **vector** involves copying its elements. Thus, after the initialization of **book2**, **book2** and **phone_book** hold separate copies of every **Entry** in the phone book. When a **vector** holds many elements, such innocent-looking assignments and initializations can be expensive. Where copying is undesirable, references or pointers (§1.8) or move operations (§4.6.2) should be used.

The standard-library **vector** is very flexible and efficient. Use it as your default container; that is, use it unless you have a solid reason to use some other container. If your reason is "efficiency," measure. Our intuition is most fallible in matters of the performance of container uses.

9.2.1 Elements

Like all standard-library containers, **vector** is a container of elements of some type **T**, that is, a **vector<T>**. Just about any type qualifies as an element type: built-in numeric types (such as **char**, **int**, and **double**), user-defined types (such as **string**, **Entry**, **list<int>**, and **Matrix<double,2>**), and pointers (such as **const char***, **Shape***, and **double***). When you insert a new element, its value is copied into the container. For example, when you put an integer with the value **7** into a container, the resulting element really has the value **7**. The element is not a reference or a pointer to some object containing **7**. This makes for nice, compact containers with fast access. For people who care about memory sizes and run-time performance this is critical.

If you have a class hierachy (§4.5) that relies on **virtual** functions to get polymorphic behavior, do not store objects directly in a container. Instead store a pointer (or a smart pointer; §11.2.1). For example:

```
vector<Shape> vs;                    // No, don't - there is no room for a Circle or a Smiley
vector<Shape*> vps;                  // better, but see §4.5.4
vector<unique_ptr<Shape>> vups;      // OK
```

9.2.2 Range Checking

The standard-library **vector** does not guarantee range checking. For example:

```
void silly(vector<Entry>& book)
{
    int i = book[book.size()].number;        // book.size() is out of range
    // ...
}
```

That initialization is likely to place some random value in **i** rather than giving an error. This is undesirable, and out-of-range errors are a common problem. Consequently, I often use a simple range-checking adaptation of **vector**:

```
template<typename T>
class Vec : public std::vector<T> {
public:
    using vector<T>::vector;            // use the constructors from vector (under the name Vec)

    T& operator[](int i)                // range check
        { return vector<T>::at(i); }

    const T& operator[](int i) const    // range check const objects; §4.2.1
        { return vector<T>::at(i); }
};
```

Vec inherits everything from **vector** except for the subscript operations that it redefines to do range checking. The **at()** operation is a **vector** subscript operation that throws an exception of type **out_of_range** if its argument is out of the **vector**'s range (§3.4.1).

For **Vec**, an out-of-range access will throw an exception that the user can catch. For example:

```
void checked(Vec<Entry>& book)
{
    try {
        book[book.size()] = {"Joe",999999};     // will throw an exception
        // ...
    }
    catch (out_of_range) {
        cout << "range error\n";
    }
}
```

The exception will be thrown, and then caught (§3.4.1). If the user doesn't catch an exception, the program will terminate in a well-defined manner rather than proceeding or failing in an undefined manner. One way to minimize surprises from uncaught exceptions is to use a **main()** with a **try**-block as its body. For example:

```
int main()
try {
    // your code
}
catch (out_of_range) {
    cerr << "range error\n";
}
catch (...) {
    cerr << "unknown exception thrown\n";
}
```

This provides default exception handlers so that if we fail to catch some exception, an error message is printed on the standard error-diagnostic output stream **cerr** (§8.2).

Some implementations save you the bother of defining **Vec** (or equivalent) by providing a range-checked version of **vector** (e.g., as a compiler option).

9.3 list

The standard library offers a doubly-linked list called **list**:

We use a **list** for sequences where we want to insert and delete elements without moving other elements. Insertion and deletion of phone book entries could be common, so a **list** could be appropriate for representing a simple phone book. For example:

```
list<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

When we use a linked list, we tend not to access elements using subscripting the way we commonly do for vectors. Instead, we might search the list looking for an element with a given value. To do this, we take advantage of the fact that a **list** is a sequence as described in Chapter 10:

```
int get_number(const string& s)
{
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
    return 0;  // use 0 to represent "number not found"
}
```

The search for **s** starts at the beginning of the list and proceeds until **s** is found or the end of **phone_book** is reached.

Sometimes, we need to identify an element in a **list**. For example, we may want to delete it or insert a new entry before it. To do that we use an *iterator*: a **list** iterator identifies an element of a **list** and can be used to iterate through a **list** (hence its name). Every standard-library container provides the functions **begin()** and **end()**, which return an iterator to the first and to one-past-the-last element, respectively (Chapter 10). Using iterators explicitly, we can – less elegantly – write the **get_number()** function like this:

```
int get_number(const string& s)
{
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
        if (p->name==s)
            return p->number;
    return 0;  // use 0 to represent "number not found"
}
```

In fact, this is roughly the way the terser and less error-prone range-**for** loop is implemented by the

compiler. Given an iterator **p**, ∗**p** is the element to which it refers, **++p** advances **p** to refer to the next element, and when **p** refers to a class with a member **m**, then **p–>m** is equivalent to **(∗p).m**.

Adding elements to a **list** and removing elements from a **list** is easy:

```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
        phone_book.insert(p,ee);        // add ee before the element referred to by p
        phone_book.erase(q);            // remove the element referred to by q
}
```

For a **list**, **insert(p,elem)** inserts an element with a copy of the value **elem** before the element pointed to by **p**. Similarly, **erase(p)** removes the element pointed to by **p** and destroys it. In both cases, **p** may be an iterator pointing one-beyond-the-end of the **List**.

These **list** examples could be written identically using **vector** and (surprisingly, unless you understand machine architecture) perform better with a small **vector** than with a small **list**. When all we want is a sequence of elements, we have a choice between using a **vector** and a **list**. Unless you have a reason not to, use a **vector**. A **vector** performs better for traversal (e.g., **find()** and **count()**) and for sorting and searching (e.g., **sort()** and **binary_search()**).

9.4 map

Writing code to look up a name in a list of *(name,number)* pairs is quite tedious. In addition, a linear search is inefficient for all but the shortest lists. The standard library offers a search tree (a red-black tree) called **map**:

In other contexts, a **map** is known as an associative array or a dictionary. It is implemented as a balanced binary tree.

The standard-library **map** is a container of pairs of values optimized for lookup. We can use the same initializer as for **vector** and **list** (§9.2, §9.3):

```
map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

When indexed by a value of its first type (called the *key*), a **map** returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

In other words, subscripting a **map** is essentially the lookup we called **get_number()**. If a **key** isn't found, it is entered into the **map** with a default value for its **value**. The default value for an integer type is **0**; the value I just happened to choose represents an invalid telephone number.

If we wanted to avoid entering invalid numbers into our phone book, we could use **find()** and **insert()** instead of **[ ]**.

9.5 unordered_map

The cost of a **map** lookup is **O(log(n))** where **n** is the number of elements in the **map**. That's pretty good. For example, for a **map** with 1,000,000 elements, we perform only about 20 comparisons and indirections to find an element. However, in many cases, we can do better by using a hashed lookup rather than comparison using an ordering function, such as **<**. The standard-library hashed containers are referred to as "unordered" because they don't require an ordering function:

For example, we can use an **unordered_map** from **<unordered_map>** for our phone book:

```
unordered_map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

As for a **map**, we can subscript an **unordered_map**:

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

The standard-library provides a default hash function for **string**s as well as for other built-in and standard-library types. If necessary, you can provide your own. Possibly, the most common need for a "custom" hash function comes when we want an unordered container of one of our own types. A hash function is often provided as a function object (§5.5). For example:

```
struct Record {
        string name;
        int product_code;
        // ...
};

struct Rhash {          // a hash function for Record
        size_t operator()(const Record& r) const
        {
                return hash<string>()(r.name) ^ hash<int>()(r.product_code);
        }
};

unordered_set<Record,Rhash> my_set; // set of Recoreds using Rhash for lookup
```

Creaing a new hash function by combining existing hash functions using exclusive or (^) is simple and often very effective.

9.6 Container Overview

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

Standard Container Summary	
vector<T>	A variable-size vector (§9.2)
list<T>	A doubly-linked list (§9.3)
forward_list<T>	A singly-linked list
deque<T>	A double-ended queue
set<T>	A set (a map with just a key and no value)
multiset<T>	A set in which a value can occur many times
map<K,V>	An associative array (§9.4)
multimap<K,V>	A map in which a key can occur many times
unordered_map<K,V>	A map using a hashed lookup (§9.5)
unordered_multimap<K,V>	A multimap using a hashed lookup
unordered_set<T>	A set using a hashed lookup
unordered_multiset<T>	A multiset using a hashed lookup

The unordered containers are optimized for lookup with a key (often a string); in other words, they are implemented using hash tables.

The containers are defined in namespace std and presented in headers <vector>, <list>, <map>, etc. (§6.3). In addition, the standard library provides container adaptors queue<T>, stack<T>, and priority_queue<T>. Look them up if you need them. The standard library also provides more specialized container-like types, such as a fixed-size array array<T,N> (§11.3.1) and bitset<N> (§11.3.2).

The standard containers and their basic operations are designed to be similar from a notational point of view. Furthermore, the meanings of the operations are equivalent for the various

containers. Basic operations apply to every kind of container for which they make sense and can be efficiently implemented. For example:

- **begin()** and **end()** give iterators to the first and one-beyond-the-last elements, respectively.
- **push_back()** can be used (efficiently) to add elements to the end of a **vector**, **list**, and other containers.
- **size()** returns the number of elements.

This notational and semantic uniformity enables programmers to provide new container types that can be used in a very similar manner to the standard ones. The range-checked vector, **Vector** (§3.4.2, Chapter 4), is an example of that. The uniformity of container interfaces allows us to specify algorithms independently of individual container types. However, each has strengths and weaknesses. For example, subscripting and traversing a **vector** is cheap and easy. On the other hand, **vector** elements are moved when we insert or remove elements; **list** has exactly the opposite properties. Please note that a **vector** is usually more efficient than a **list** for short sequences of small elements (even for **insert()** and **erase()**). I recommend the standard-library **vector** as the default type for sequences of elements: you need a reason to choose another.

Consider the singly-linked list, **forward_list**, a container optimized for the empty sequence (which occupies just one word) because the number of elements are zero or very low; such sequences are surprisingly useful.

9.7 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 31 of [Stroustrup,2013].
[2] An STL container defines a sequence; §9.2.
[3] STL containers are resource handles; §9.2, §9.3, §9.4, §9.5.
[4] Use **vector** as your default container; §9.2, §9.6.
[5] For simple traversals of a container, use a range-**for** loop or a begin/end pair of iterators; §9.2, §9.3.
[6] Use **reserve()** to avoid invalidating pointers and iterators to elements; §9.2.
[7] Don't assume performance benefits from **reserve()** without measurement; §9.2.
[8] Use **push_back()** or **resize()** on a container rather than **realloc()** on an array; §9.2.
[9] Don't use iterators into a resized **vector**; §9.2.
[10] Do not assume that **[ ]** range checks; §9.2.
[11] Use **at()** when you need guaranteed range checks; §9.2.
[12] Elements are copied into a container; §9.2.1.
[13] To preserve polymorphic behavior of elements, store pointers; §9.2.1.
[14] Insertion operators, such as **insert()** and **push_back()** are often surprisingly efficient on a **vector**; §9.3.
[15] Use **forward_list** for sequences that are usually empty; §9.6.
[16] When it comes to performance, don't trust your intuition: measure; §9.2.
[17] A **map** is usually implemented as a red-black tree; §9.4.
[18] An **unordered_map** is a hash table; §9.5.

[19] Pass a container by reference and return a container by value; §9.2.
[20] For a container, use the `()`-initializer syntax for sizes and the `{}`-initializer syntax for lists of elements; §4.2.3, §9.2.
[21] Prefer compact and contiguous data structures; §9.3.
[22] A `list` is relatively expensive to traverse; §9.3.
[23] Use unordered containers if you need fast lookup for large amounts of data; §9.5.
[24] Use ordered associative containers (e.g., `map` and `set`) if you need to iterate over their elements in order; §9.4.
[25] Use unordered containers for element types with no natural order (e.g., no reasonable `<`); §9.4.
[26] Experiment to check that you have an acceptable hash function; §9.5.
[27] Hash function obtained by combining standard hash functions for elements using exclusive or are often good; §9.5.
[28] Know your standard-library containers and prefer them to hand-crafted data structures; §9.6.

This page intentionally left blank

10

Algorithms

Do not multiply entities beyond necessity.
– William Occam

- Introduction
- Use of Iterators
- Iterator Types
- Stream Iterators
- Predicates
- Algorithm Overview
- Container Algorithms
- Advice

10.1 Introduction

A data structure, such as a list or a vector, is not very useful on its own. To use one, we need operations for basic access such as adding and removing elements (as is provided for **list** and **vector**). Furthermore, we rarely just store objects in a container. We sort them, print them, extract subsets, remove elements, search for objects, etc. Consequently, the standard library provides the most common algorithms for containers in addition to providing the most common container types. For example, the we can simply and efficiently sort a **vector** of **Entry**s and place a copy of each unique **vector** element on a **list**:

```
void f(vector<Entry>& vec, list<Entry>& lst)
{
    sort(vec.begin(),vec.end());                    // use < for order
    unique_copy(vec.begin(),vec.end(),lst.begin()); // don't copy adjacent equal elements
}
```

For this to work, less than (**<**) must be defined for **Entry**s. For example:

```
bool operator<(const Entry& x, const Entry& y)     // less than
{
    return x.name<y.name;          // order Entrys by their names
}
```

A standard algorithm is expressed in terms of (half-open) sequences of elements. A *sequence* is represented by a pair of iterators specifying the first element and the one-beyond-the-last element:

In the example, **sort()** sorts the sequence defined by the pair of iterators **vec.begin()** and **vec.end()** – which just happens to be all the elements of a **vector**. For writing (output), you need only to specify the first element to be written. If more than one element is written, the elements following that initial element will be overwritten. Thus, to avoid errors, **lst** must have at least as many elements as there are unique values in **vec**.

If we wanted to place the unique elements in a new container, we could have written:

```
list<Entry> f(vector<Entry>& vec)
{
    list<Entry> res;
    sort(vec.begin(),vec.end());
    unique_copy(vec.begin(),vec.end(),back_inserter(res));     // append to res
    return res;
}
```

The call **back_inserter(res)** constructs an iterator for **res** that adds elements at the end of a container, extending the container to make room for them. This saves us from first having to allocate a fixed amount of space and then filling it. Thus, the standard containers plus **back_inserter()**s eliminate the need to use error-prone, explicit C-style memory management using **realloc()**. The standard-library **list** has a move constructor (§4.6.2) that makes returning **res** by value efficient (even for **list**s of thousands of elements).

If you find the pair-of-iterators style of code, such as **sort(vec.begin(),vec.end())**, tedious, you can define container versions of the algorithms and write **sort(vec)** (§10.7).

10.2 Use of Iterators

When you first encounter a container, a few iterators referring to useful elements can be obtained; **begin()** and **end()** are the best examples of this. In addition, many algorithms return iterators. For example, the standard algorithm **find** looks for a value in a sequence and returns an iterator to the element found:

```
bool has_c(const string& s, char c)        // does s contain the character c?
{
        auto p = find(s.begin(),s.end(),c);
        if (p!=s.end())
                return true;
        else
                return false;
}
```

Like many standard-library search algorithms, **find** returns **end()** to indicate "not found." An equivalent, shorter, definition of **has_c()** is:

```
bool has_c(const string& s, char c)        // does s contain the character c?
{
        return find(s.begin(),s.end(),c)!=s.end();
}
```

A more interesting exercise would be to find the location of all occurrences of a character in a string. We can return the set of occurrences as a **vector** of **string** iterators. Returning a **vector** is efficient because **vector** provides move semantics (§4.6.1). Assuming that we would like to modify the locations found, we pass a non-**const** string:

```
vector<string::iterator> find_all(string& s, char c)        // find all occurrences of c in s
{
        vector<string::iterator> res;
        for (auto p = s.begin(); p!=s.end(); ++p)
                if (*p==c)
                        res.push_back(p);
        return res;
}
```

We iterate through the string using a conventional loop, moving the iterator **p** forward one element at a time using **++** and looking at the elements using the dereference operator *. We could test **find_all()** like this:

```
void test()
{
        string m {"Mary had a little lamb"};
        for (auto p : find_all(m,'a'))
                if (*p!='a')
                        cerr << "a bug!\n";
}
```

That call of **find_all()** could be graphically represented like this:

Iterators and standard algorithms work equivalently on every standard container for which their use makes sense. Consequently, we could generalize **find_all()**:

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)          // find all occurrences of v in c
{
        vector<typename C::iterator> res;
        for (auto p = c.begin(); p!=c.end(); ++p)
                if (*p==v)
                        res.push_back(p);
        return res;
}
```

The **typename** is needed to inform the compiler that **C**'s **iterator** is supposed to be a type and not a value of some type, say, the integer **7**. We can hide this implementation detail by introducing a type alias (§5.7) for **Iterator**:

```
template<typename T>
using Iterator = typename T::iterator;          // T's iterator

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)         // find all occurrences of v in c
{
        vector<Iterator<C>> res;
        for (auto p = c.begin(); p!=c.end(); ++p)
                if (*p==v)
                        res.push_back(p);
        return res;
}
```

We can now write:

```
void test()
{
        string m {"Mary had a little lamb"};

        for (auto p : find_all(m,'a'))                  // p is a string::iterator
                if (*p!='a')
                        cerr << "string bug!\n";

        list<double> ld {1.1, 2.2, 3.3, 1.1};
        for (auto p : find_all(ld,1.1))
                if (*p!=1.1)
                        cerr << "list bug!\n";

        vector<string> vs { "red", "blue", "green", "green", "orange", "green" };
        for (auto p : find_all(vs,"red"))
                if (*p!="red")
                        cerr << "vector bug!\n";
```

```
        for (auto p : find_all(vs,"green"))
            *p = "vert";
    }
```

Iterators are used to separate algorithms and containers. An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored. Conversely, a container knows nothing about the algorithms operating on its elements; all it does is to supply iterators upon request (e.g., **begin()** and **end()**). This model of separation between data storage and algorithm delivers very general and flexible software.

10.3 Iterator Types

What are iterators really? Any particular iterator is an object of some type. There are, however, many different iterator types, because an iterator needs to hold the information necessary for doing its job for a particular container type. These iterator types can be as different as the containers and the specialized needs they serve. For example, a **vector**'s iterator could be an ordinary pointer, because a pointer is quite a reasonable way of referring to an element of a **vector**:

Alternatively, a **vector** iterator could be implemented as a pointer to the **vector** plus an index:

Using such an iterator would allow range checking.

A **list** iterator must be something more complicated than a simple pointer to an element because an element of a **list** in general does not know where the next element of that **list** is. Thus, a **list** iterator might be a pointer to a link:

What is common for all iterators is their semantics and the naming of their operations. For example, applying **++** to any iterator yields an iterator that refers to the next element. Similarly, * yields

the element to which the iterator refers. In fact, any object that obeys a few simple rules like these is an iterator – *Iterator* is a concept (§5.4). Furthermore, users rarely need to know the type of a specific iterator; each container "knows" its iterator types and makes them available under the conventional names **iterator** and **const_iterator**. For example, **list<Entry>::iterator** is the general iterator type for **list<Entry>**. We rarely have to worry about the details of how that type is defined.

10.4 Stream Iterators

Iterators are a general and useful concept for dealing with sequences of elements in containers. However, containers are not the only place where we find sequences of elements. For example, an input stream produces a sequence of values, and we write a sequence of values to an output stream. Consequently, the notion of iterators can be usefully applied to input and output.

To make an **ostream_iterator**, we need to specify which stream will be used and the type of objects written to it. For example:

```
ostream_iterator<string> oo {cout};      // write strings to cout
```

The effect of assigning to ***oo** is to write the assigned value to **cout.** For example:

```
int main()
{
    *oo = "Hello, ";      // meaning cout<<"Hello, "
    ++oo;
    *oo = "world!\n";     // meaning cout<<"world!\n"
}
```

This is yet another way of writing the canonical message to standard output. The **++oo** is done to mimic writing into an array through a pointer.

Similarly, an **istream_iterator** is something that allows us to treat an input stream as a read-only container. Again, we must specify the stream to be used and the type of values expected:

```
istream_iterator<string> ii {cin};
```

Input iterators are used in pairs representing a sequence, so we must provide an **istream_iterator** to indicate the end of input. This is the default **istream_iterator**:

```
istream_iterator<string> eos {};
```

Typically, **istream_iterator**s and **ostream_iterator**s are not used directly. Instead, they are provided as arguments to algorithms. For example, we can write a simple program to read a file, sort the words read, eliminate duplicates, and write the result to another file:

```
int main()
{
    string from, to;
    cin >> from >> to;                  // get source and target file names

    ifstream is {from};                 // input stream for file "from"
    istream_iterator<string> ii {is};   // input iterator for stream
    istream_iterator<string> eos {};    // input sentinel
```

```
    ofstream os {to};                          // output stream for file "to"
    ostream_iterator<string> oo {os,"\n"};     // output iterator for stream

    vector<string> b {ii,eos};                 // b is a vector initialized from input
    sort(b.begin(),b.end());                   // sort the buffer

    unique_copy(b.begin(),b.end(),oo);         // copy buffer to output, discard replicated values

    return !is.eof() || !os;                   // return error state (§1.3, §8.4)
}
```

An **ifstream** is an **istream** that can be attached to a file, and an **ofstream** is an **ostream** that can be attached to a file (§8.7). The **ostream_iterator**'s second argument is used to delimit output values.

Actually, this program is longer than it needs to be. We read the strings into a **vector**, then we **sort()** them, and then we write them out, eliminating duplicates. A more elegant solution is not to store duplicates at all. This can be done by keeping the **string**s in a **set**, which does not keep duplicates and keeps its elements in order (§9.4). That way, we could replace the two lines using a **vector** with one using a **set** and replace **unique_copy()** with the simpler **copy()**:

```
    set<string> b {ii,eos};            // collect strings from input
    copy(b.begin(),b.end(),oo);        // copy buffer to output
```

We used the names **ii**, **eos**, and **oo** only once, so we could further reduce the size of the program:

```
int main()
{
    string from, to;
    cin >> from >> to;              // get source and target file names

    ifstream is {from};             // input stream for file "from"
    ofstream os {to};               // output stream for file "to"

    set<string> b {istream_iterator<string>{is},istream_iterator<string>{}}; // read input
    copy(b.begin(),b.end(),ostream_iterator<string>{os,"\n"});              // copy to output

    return !is.eof() || !os;        // return error state (§1.3, §8.4)
}
```

It is a matter of taste and experience whether or not this last simplification improves readability.

10.5 Predicates

In the examples above, the algorithms have simply "built in" the action to be done for each element of a sequence. However, we often want to make that action a parameter to the algorithm. For example, the **find** algorithm (§10.2, §10.6) provides a convenient way of looking for a specific value. A more general variant looks for an element that fulfills a specified requirement, a *predicate*. For example, we might want to search a **map** for the first value larger than **42**. A **map** allows us to access its elements as a sequence of *(key,value)* pairs, so we can search a **map<string,int>**'s sequence for a **pair<const string,int>** where the **int** is greater than **42**:

```
void f(map<string,int>& m)
{
        auto p = find_if(m.begin(),m.end(),Greater_than{42});
        // ...
}
```

Here, Greater_than is a function object (§5.5) holding the value (42) to be compared against:

```
struct Greater_than {
        int val;
        Greater_than(int v) : val{v} { }
        bool operator()(const pair<string,int>& r) { return r.second>val; }
};
```

Alternatively, we could use a lambda expression (§5.5):

```
auto p = find_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });
```

A predicate should not modify the elements to which it is applied.

10.6 Algorithm Overview

A general definition of an algorithm is "a finite set of rules which gives a sequence of operations for solving a specific set of problems [and] has five important features: Finiteness ... Definiteness ... Input ... Output ... Effectiveness" [Knuth,1968,§1.1]. In the context of the C++ standard library, an algorithm is a function template operating on sequences of elements.

The standard library provides dozens of algorithms. The algorithms are defined in namespace std and presented in the <algorithm> header. These standard-library algorithms all take sequences as inputs. A half-open sequence from b to e is referred to as [b:e). Here are a few examples:

Selected Standard Algorithms	
p=find(b,e,x)	p is the first p in [b:e) so that *p==x
p=find_if(b,e,f)	p is the first p in [b:e) so that f(*p)==true
n=count(b,e,x)	n is the number of elements *q in [b:e) so that *q==x
n=count_if(b,e,f)	n is the number of elements *q in [b:e) so that f(*q,x)
replace(b,e,v,v2)	Replace elements *q in [b:e) so that *q==v by v2
replace_if(b,e,f,v2)	Replace elements *q in [b:e) so that f(*q) by v2
p=copy(b,e,out)	Copy [b:e) to [out:p)
p=copy_if(b,e,out,f)	Copy elements *q from [b:e) so that f(*q) to [out:p)
p=move(b,e,out)	Move [b:e) to [out:p)
p=unique_copy(b,e,out)	Copy [b:e) to [out:p); don't copy adjacent duplicates
sort(b,e)	Sort elements of [b:e) using < as the sorting criterion
sort(b,e,f)	Sort elements of [b:e) using f as the sorting criterion
(p1,p2)=equal_range(b,e,v)	[p1:p2) is the subsequence of the sorted sequence [b:e) with the value v; basically a binary search for v
p=merge(b,e,b2,e2,out)	Merge two sorted sequences [b:e) and [b2:e2) into [out:p)

These algorithms, and many more (e.g., §12.3), can be applied to elements of containers, **string**s, and built-in arrays.

Some algorithms, such as **replace()** and **sort()**, modify element values, but no algorithm add or subtract elements of a container. The reason is that a sequence does not identify the container that holds the elements of the sequence. If you want to add elements, you need something, such as an **back_inserter** that knows about the container (§10.1), or directly refer to the container itself, such as **push_back()** or **erase()** (§9.2).

The standard-library algorithms tend to be more carefully designed, specified, and implemented than the average hand-crafted loop, so know them and use them in preference to code written in the bare language.

10.7 Container Algorithms

A sequence is defined by a pair of iterators [**begin**:**end**). This is general and flexible, but most often, we apply an algorithm to a sequence that is the contents of a container. For example:

```
sort(v.begin(),v.end());
```

Why don't we just say **sort(v)**? We can easily provide that shorthand:

```
namespace Estd {
    using namespace std;

    template<typename C>
    void sort(C& c)
    {
        sort(c.begin(),c.end());
    }

    template<typename C, typename Pred>
    void sort(C& c, Pred p)
    {
        sort(c.begin(),c.end(),p);
    }

    // ...
}
```

I put the container versions of **sort()** (and other algorithms) into their own namespace **Estd** ("extended **std**") to avoid interfering with other programmers' uses of namespace **std**.

10.8 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 32 of [Stroustrup,2013].

[2] An STL algorithm operates on one or more sequences; §10.1.

[3] An input sequence is half-open and defined by a pair of iterators; §10.1.
[4] When searching, an algorithm usually returns the end of the input sequence to indicate ‘‘not found’’; §10.2.
[5] Algorithms do not directly add or subtract elements from their argument sequences; §10.2, §10.6.
[6] When writing a loop, consider whether it could be expressed as a general algorithm; §10.2.
[7] Use predicates and other function objects to give standard algorithms a wider range of meanings; §10.5, §10.6.
[8] A predicate must not modify its argument; §10.5.
[9] Know your standard-library algorithms and prefer them to hand-crafted loops; §10.6.
[10] When the pair-of-iterators style becomes tedious, introduce a container/range algorithm; §10.7.

11

Utilities

The time you enjoy wasting is not wasted time.
– Bertrand Russell

- Introduction
- Resource Management
  unique_ptr and **shared_ptr**
- Specialized Containers
  array; **bitset**; **pair** and **tuple**
- Time
- Function Adaptors
  bind(); **mem_fn()**; **function**
- Type Functions
  iterator_traits; Type Predicates
- Advice

11.1 Introduction

Not all standard-library components come as part of obviously labeled facilities, such as "containers" or "I/O." This section gives a few examples of small, widely useful components. The point here is that a function or a type need not be complicated or closely tied to a mass of other functions and types to be useful. Such library components mostly act as building blocks for more powerful library facilities, including other components of the standard library.

11.2 Resource Management

One of the key tasks of any nontrivial program is to manage resources. A resource is something that must be acquired and later (explicitly or implicitly) released. Examples are memory, locks,

sockets, thread handles, and file handles. For a long-running program, failing to release a resource in a timely manner (“a leak”) can cause serious performance degradation and possibly even a miserable crash. Even for short programs, a leak can become an embarrassment, say by a resource shortage increasing the run time by orders of magnitude.

The standard library components are designed not to leak resources. To do this, they rely on the basic language support for resource management using constructor/destructor pairs to ensure that a resource doesn’t outlive an object responsible for it. The use of a constructor/destructor pair in **Vector** to manage the lifetime of its elements is an example (§4.2.2) and all standard-library containers are implemented in similar ways. Importantly, this approach interacts correctly with error handling using exceptions. For example, the technique is used for the standard-library lock classes:

```
mutex m; // used to protect access to shared data
// ...
void f()
{
    unique_lock<mutex> lck {m}; // acquire the mutex m
    // ... manipulate shared data ...
}
```

A **thread** will not proceed until **lck**’s constructor has acquired its **mutex**, **m** (§13.5). The corresponding destructor releases the resource. So, in this example, **unique_lock**’s destructor releases the **mutex** when the thread of control leaves **f()** (through a **return**, by “falling off the end of the function,” or through an exception throw).

This is an application of the “Resource Acquisition Is Initialization” technique (RAII; §4.2.2). RAII is fundamental to the idiomatic handling of resources in C++. Containers (such as **vector** and **map**), **string**, and **iostream** manage their resources (such as file handles and buffers) similarly.

11.2.1 unique_ptr and shared_ptr

The examples so far take care of objects defined in a scope, releasing the resources they acquire at the exit from the scope, but what about objects allocated on the free store? In **<memory>**, the standard library provides two “smart pointers” to help manage objects on the free store:

[1] **unique_ptr** to represent unique ownership
[2] **shared_ptr** to represent shared ownership

The most basic use of these “smart pointers” is to prevent memory leaks caused by careless programming. For example:

```
void f(int i, int j)      // X* vs. unique_ptr<X>
{
    X* p = new X;                  // allocate a new X
    unique_ptr<X> sp {new X};      // allocate a new X and give its pointer to unique_ptr
    // ...
    if (i<99) throw Z{};           // may throw an exception
    if (j<77) return;              // may return "early"
    // ...
    p->do_something();             // may throw an exception
    sp->do_something();            // may throw an exception
```

```
        // ...
        delete p;                       // destroy *p
}
```

Here, we "forgot" to delete p if i<99 or if j<77. On the other hand, unique_ptr ensures that its object is properly destroyed whichever way we exit f() (by throwing an exception, by executing return, or by "falling off the end"). Ironically, we could have solved the problem simply by *not* using a pointer and *not* using new:

```
void f(int i, int j)    // use a local variable
{
        X x;
        // ...
}
```

Unfortunately, overuse of new (and of pointers and references) seems to be an increasing problem.

However, when you really need the semantics of pointers, unique_ptr is a very lightweight mechanism with no space or time overhead compared to correct use of a built-in pointer. Its further uses include passing free-store allocated objects in and out of functions:

```
unique_ptr<X> make_X(int i)
        // make an X and immediately give it to a unique_ptr
{
        // ... check i, etc. ...
        return unique_ptr<X>{new X{i}};
}
```

A unique_ptr is a handle to an individual object (or an array) in much the same way that a vector is a handle to a sequence of objects. Both control the lifetime of other objects (using RAII) and both rely on move semantics to make return simple and efficient.

The shared_ptr is similar to unique_ptr except that shared_ptrs are copied rather than moved. The shared_ptrs for an object share ownership of an object and that object is destroyed when the last of its shared_ptrs is destroyed. For example:

```
void f(shared_ptr<fstream>);
void g(shared_ptr<fstream>);

void user(const string& name, ios_base::openmode mode)
{
        shared_ptr<fstream> fp {new fstream(name,mode)};
        if (!*fp)                       // make sure the file was properly opened
                throw No_file{};

        f(fp);
        g(fp);
        // ...
}
```

Now, the file opened by fp's constructor will be closed by the last function to (explicitly or implicitly) destroy a copy of fp. Note that f() or g() may spawn a task holding a copy of fp or in some other way store a copy that outlives user(). Thus, shared_ptr provides a form of garbage collection

that respects the destructor-based resource management of the memory-managed objects. This is neither cost free nor exorbitantly expensive, but it does make the lifetime of the shared object hard to predict. Use shared_ptr only if you actually need shared ownership.

Creating an object on the free store and then passing a pointer to it to a smart pointer is logically a bit odd and can be verbose. To compensate, the standard library (in <memory>) provides a function make_shared(). For example:

```
struct S {
    int i;
    string s;
    double d;
    // ...
};

shared_ptr<S> p1 {new S {1,"Ankh Morpork",4.65}};

auto p2 = make_shared<S>(2,"Oz",7.62);
```

Now, p2 is a shared_ptr<S> pointing to an object of type S allocated on the free store, containing {1,string{"Ankh Morpork"},4.65}.

Currently, there is no standard-library make_unique() similar to make_shared() and make_pair() (§11.3.3). However, it is easily defined:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>{new T{std::forward<Args>(args)...}};
}
```

No, I don't claim that this definition is trivial to understand, but it is efficient and quite general. The elipses, ..., indicate the use of a variadic template (§5.6). We can now write:

```
auto p2 = make_unique<S>(3,"Atlantis",11.3);
```

Given unique_ptr and shared_ptr, we can implement a complete "no naked new" policy (§4.2.2) for many programs. However, these "smart pointers" are still conceptually pointers and therefore only my second choice for resource management – after containers and other types that manage their resources at a higher conceptual level. In particular, shared_ptrs do not in themselves provide any rules for which of their owners can read and/or write the shared object. Data races (§13.7) and other forms of confusion are not addressed simply by eliminating the resource management issues.

Where do we use "smart pointers" (such as unique_ptr) rather than resource handles with operations designed specifically for the resource (such as vector or thread)? Unsurprisingly, the answer is "when we need pointer semantics."

- When we share an object, we need pointers (or references) to refer to the shared object, so a shared_ptr becomes the obvious choice (unless there is an obvious single owner).
- When we refer to a polymorphic object, we need a pointer (or a reference) because we don't know the exact type of the object referred to (or even its size), so a unique_ptr becomes the obvious choice.

- A shared polymorphic object typically requires shared_ptrs.

We do *not* need to use a pointer to return a collection of objects from a function; a container that is a resource handle will do that simply and efficiently (§4.6.2).

11.3 Specialized Containers

The standard library provides several containers that don't fit perfectly into the STL framework (Chapter 9, Chapter 10). Examples are built-in arrays, array, and string. I sometimes refer to those as "almost containers," but that is not quite fair: they hold elements, so they are containers, but each has restrictions or added facilities that make them awkward in the context of the STL. Describing them separately also simplifies the description of the STL.

"Almost Containers"	
T[N]	Built-in array: a fixed-size continuously allocated sequence of N elements of type T; implicitly converts to a T*
array<T,N>	A fixed-size continuously allocated sequence of N elements of type T; like the built-in array, but with most problems solved
bitset<N>	A fixed-size sequence of N bits
vector<bool>	A sequence of bits compactly stored in a specialization of vector
pair<T,U>	Two elements of types T and U
tuple<T...>	A sequence of an arbitrary number of elements of arbitrary types
basic_string<C>	A sequence of characters of type C; provides string operations
valarray<T>	An array of numeric values of type T; provides numeric operations

Why does the standard library provide so many containers? They serve common but different (often overlapping) needs. If the standard library didn't provide them, many people would have to design and implement their own. For example:

- pair and tuple are heterogeneous; all other containers are homogeneous (all elements are of the same type).
- array, vector, and tuple elements are contiguously allocated; forward_list and map are linked structures.
- bitset and vector<bool> hold bits and access them through proxy objects; all other standard-library containers can hold a variety of types and access elements directly.
- basic_string requires its elements to be some form of character and to provide string manipulation, such as concatenation and locale-sensitive operations
- valarray requires its elements to be numbers and to provide numerical operations.

All of these containers can be seen as providing specialized services needed by large communities of programmers. No single container could serve all of these needs because some needs are contradictory, for example, "ability to grow" vs. "guaranteed to be allocated in a fixed location," and "elements do not move when elements are added" vs. "contiguously allocated." Furthermore, a very general container would imply overhead deemed unacceptable for individual containers.

11.3.1 array

An **array**, defined in **<array>**, is a fixed-size sequence of elements of a given type where the number of elements is specified at compile time. Thus, an **array** can be allocated with its elements on the stack, in an object, or in static storage. The elements are allocated in the scope where the **array** is defined. An **array** is best understood as a built-in array with its size firmly attached, without implicit, potentially surprising conversions to pointer types, and with a few convenience functions provided. There is no overhead (time or space) involved in using an **array** compared to using a built-in array. An **array** does *not* follow the "handle to elements" model of STL containers. Instead, an **array** directly contains its elements.

An **array** can be initialized by an initializer list:

```
array<int,3> a1 = {1,2,3};
```

The number of elements in the initializer must be equal to or less than the number of elements specified for the **array**.

The element count is not optional:

```
array<int> ax = {1,2,3};          // error size not specified
```

The element count must be a constant expression:

```
void f(int n)
{
    array<string,n> aa = {"John's", "Queens' "};          // error: size not a constant expression
    //
}
```

If you need the element count to be a variable, use **vector**.

When necessary, an **array** can be explicitly passed to a C-style function that expects a pointer. For example:

```
void f(int* p, int sz);        // C-style interface

void g()
{
    array<int,10> a;

    f(a,a.size());                // error: no conversion
    f(&a[0],a.size());            // C-style use
    f(a.data(),a.size());         // C-style use

    auto p = find(a.begin(),a.end(),777);        // C++/STL-style use
    // ...
}
```

Why would we use an **array** when **vector** is so much more flexible? Because an **array** is less flexible, it is simpler. Occasionally, there is a significant performance advantage to be had by directly accessing elements allocated on the stack rather than allocating elements on the free store, accessing them indirectly through the **vector** (a handle), and then deallocating them. On the other hand, the stack is a limited resource (especially on some embedded systems), and stack overflow is nasty.

Why would we use an **array** when we could use a built-in array? An **array** knows its size, so it is easy to use with standard-library algorithms, and it can be copied (using **=** or initialization). However, my main reason to prefer **array** is that it saves me from surprising nasty conversions to pointers. Consider:

```
void h()
{
    Circle a1[10];
    array<Circle,10> a2;
    // ...
    Shape* p1 = a1;     // OK: disaster waiting to happen
    Shape* p2 = a2;     // error: no conversion of array<Circle,10> to Shape*
    p1[3].draw();       // disaster
}
```

The "disaster" comment assumes that **sizeof(Shape)<sizeof(Circle)**, so that subscripting a **Circle[]** through a **Shape*** gives a wrong offset. All standard containers provide this advantage over built-in arrays.

11.3.2 bitset

Aspects of a system, such as the state of an input stream, are often represented as a set of flags indicating binary conditions such as good/bad, true/false, and on/off. C++ supports the notion of small sets of flags efficiently through bitwise operations on integers (§1.5). Class **bitset<N>** generalizes this notion and offers greater convenience by providing operations on a sequence of **N** bits [**0**:**N**), where **N** is known at compile time. For sets of bits that don't fit into a **long long int**, using a **bitset** is much more convenient than using integers directly. For smaller sets, **bitset** is usually optimized. If you want to name the bits, rather than numbering them, you can use a **set** (§9.4) or an enumeration (§2.5).

A **bitset** can be initialized with an integer or a string:

```
bitset<9> bs1 {"110001111"};
bitset<9> bs2 {399};
```

The usual bitwise operations (§1.5) can be applied, as can left- and right-shift operations (**<<** and **>>**):

```
bitset<9> bs3 = ~bs1;          // complement: bs3=="001110000"
bitset<9> bs4 = bs1&bs3;       // all zeros
bitset<9> bs5 = bs1<<2;        // shift left: bs5 = "111000000"
```

The shift operators (here, **<<**) "shifts in" zeros.

The operations **to_ullong()** and **to_string()** provide the inverse operations to the constructors. For example, we could write out the binary representation of an **int**:

```
void binary(int i)
{
    bitset<8*sizeof(int)> b = i;        // assume 8-bit byte (see also §12.7)
    cout << b.to_string() << '\n';      // write out the bits of i
}
```

This prints the bits represented as 1s and 0s from left to right, with the most significant bit leftmost, so that argument 123 would give the output

```
00000000000000000000000001111011
```

For this example, it is simpler to directly use the bitset output operator:

```
void binary2(int i)
{
    bitset<8*sizeof(int)> b = i;    // assume 8-bit byte (see also §12.7)
    cout << b << '\n';              // write out the bits of i
}
```

11.3.3 pair and tuple

Often, we need some data that is just data; that is, a collection of values, rather than an object of a class with a well-defined semantics and an invariant for its value (§3.4.2). In such cases, we could define a simple struct with an appropriate set of appropriately named members. Alternatively, we could let the standard library write the definition for us. For example, the standard-library algorithm equal_range returns a pair of iterators specifying a subsequence meeting a predicate:

```
template<typename Forward_iterator, typename T, typename Compare>
    pair<Forward_iterator,Forward_iterator>
    equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

Given a sorted sequence [first:last), equal_range() will return the pair representing the subsequence that matches the predicate cmp. We can use that to search in a sorted sequence of Records:

```
auto rec_eq = [](const Record& r1, const Record& r2) { return r1.name<r2.name;};    // compare names

void f(const vector<Record>& v)         // assume that v is sorted on its "name" field
{
    auto er = equal_range(v.begin(),v.end(),Record{"Reg"},rec_eq);

    for (auto p = er.first; p!=er.second; ++p)      // print all equal records
        cout << *p;                                 // assume that << is defined for Record
}
```

The first member of a pair is called first and the second member is called second. This naming is not particularly creative and may look a bit odd at first, but such consistent naming is a boon when we want to write generic code.

The standard-library pair (from <utility>) is quite frequently used in the standard library and elsewhere. A pair provides operators, such as =, ==, and <, if its elements do. The make_pair() function makes it easy to create a pair without explicitly mentioning its type. For example:

```
void f(vector<string>& v)
{
    auto pp = make_pair(v.begin(),2);   // pp is a pair<vector<string>::iterator,int>
    // ...
}
```

If you need more than two elements (or less), you can use **tuple** (from **<utility>**). A **tuple** is a heterogeneous sequence of elements; for example:

```
tuple<string,int,double> t2{"Sild",123, 3.14};        // the type is explicitly specified

auto t = make_tuple(string{"Herring"},10, 1.23);      // the type is deduced to tuple<string,int,double>

string s = get<0>(t);       // get first element of tuple: "Herring"
int x = get<1>(t);          // 10
double d = get<2>(t);       // 1.23
```

The elements of a **tuple** are numbered (starting with zero), rather than named the way elements of **pair**s are (**first** and **second**). To get compile-time selection of elements, I must unfortunately use the ugly **get<1>(t)**, rather than **get(t,1)** or **t[1]**.

Like **pair**s, **tuple**s can be assigned and compared if their elements can be.

A **pair** is common in interfaces because often we want to return more than one value, such as a result and an indicator of the quality of that result. It is less common to need three or more parts to a result, so **tuple**s are more often found in the implementations of generic algorithms.

11.4 Time

The standard library provides facilities for dealing with time. For example, here is the basic way of timing something:

```
using namespace std::chrono;      // see §3.3

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1−t0).count() << "msec\n";
```

The clock returns a **time_point** (a point in time). Subtracting two **time_point**s gives a **duration** (a period of time). Various clocks give their results in various units of time (the clock I used measures **nanoseconds**), so it is usually a good idea to convert a **duration** into a known unit. That's what **duration_cast** does.

The standard-library facilities for dealing with time are found in the subnamespace **std::chrono** in **<chrono>**.

Don't make statements about "efficiency" of code without first doing time measurements. Guesses about performance are most unreliable.

11.5 Function Adaptors

A function adaptor takes a function as argument and returns a function object that can be used to invoke the original function. The standard library provides **bind()** and **mem_fn()** adaptors to do argument binding, also called *Currying* or *partial evaluation*. Binders were heavily used in the past, but most uses seem to be more easily expressed using lambdas (§5.5).

11.5.1 bind()

Given a function and a set of arguments, **bind()** produces a function object that can be called with "the remaining" arguments, if any, of the function. For example:

```
double cube(double);

auto cube2 = bind(cube,2);
```

A call **cube2()** will invoke **cube** with the argument **2**, that is, **cube(2)**. We don't have to bind every argument of a function. For example:

```
using namespace placeholders;

void f(int,const string&);
auto g = bind(f,2,_1);          // bind f()'s first argument to 2
f(2,"hello");
g("hello");                     // also calls f(2,"hello");
```

The curious **_1** argument to the binder is a placeholder telling **bind()** where arguments to the resulting function object should go. In this case, **g()**'s (first) argument is used as **f()**'s second argument.

The placeholders are found in the (sub)namespace **std::placeholders** that is part of **<functional>**.

To bind arguments for an overloaded function, we have to explicitly state which version of the function we want to bind:

```
int pow(int,int);
double pow(double,double);   // pow() is overloaded

auto pow2 = bind(pow,_1,2);                          // error: which pow()?
auto pow2 = bind((double(*)(double,double))pow,_1,2);   // OK (but ugly)
```

I assigned the result of **bind()** to a variable declared using **auto**. This saves me the bother of specifying the return type of a call of **bind()**. That can be useful because the return type of **bind()** varies with the type of function to be called and the argument values stored. In particular, the returned function object is larger when it has to hold values of bound parameters. When we want to be specific about the types of the arguments required and the type of result returned, we can use a **function** (§11.5.3).

11.5.2 mem_fn()

The function adaptor **mem_fn(mf)** produces a function object that can be called as a nonmember function. For example:

```
void user(Shape* p)
{
    p->draw();
    auto draw = mem_fn(&Shape::draw);
    draw(p);
}
```

The major use of **mem_fn()** is when an algorithm requires an operation to be called as a nonmember function. For example:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),mem_fn(&Shape::draw));
}
```

Thus, mem_fn() can be seen as a mapping from the object-oriented calling style to the functional one.

Often, lambdas provide a simple and general alternative to binders. For example:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),[](Shape* p) { p->draw(); });
}
```

11.5.3 function

A bind() can be used directly, and it can be used to initialize an auto variable. In that, bind() resembles a lambda.

If we want to assign the result of bind() to a variable with a specific type, we can use the standard-library type function. A function is specified with a specific return type and a specific argument type. For example:

```
int f1(double);
function<int(double)> fct {f1}; // initialize to f1
int f2(int);

void user()
{
    fct = [](double d) { return round(d); };    // assign lambda to fct
    fct = f1;                                   // assign function to fct
    fct = f2;                                   // error: incorrect argument type
}
```

The standard-library function is a type that can hold any object you can invoke using the call operator (). That is, an object of type function is a function object (§5.5). For example:

```
int round(double x) { return static_cast<int>(floor(x+0.5)); }   // conventional 4/5 rounding

function<int(double)> f;  // f can hold anything that can be called with a double and return an int

enum class Round_style { truncate, round };

struct Round {        // function object carrying a state
    Round_style s;
    Round(Round_style ss) :s(ss) { }
    int operator()(double x) const { return static_cast<int>((s==Round_style::round) ? (x+0.5) : x); };
};
```

I use static_cast (§14.2.3) to make it explicit that I want to return an int.

```
void t1()
{
    f = round;
    cout << f(7.6) << '\n';                    // call through f to the function round

    f = Round(Round_style::truncate);
    cout << f(7.6) << '\n';                    // call the function object

    Round_style style = Round_style::round;
    f = [style] (double x){ return static_cast<int>((style==Round_style::round) ? x+0.5 : x); };

    cout << f(7.6) << '\n';                    // call the lambda

    vector<double> v {7.6};
    f = Round(Round_style::round);
    std::transform(v.begin(),v.end(),v.begin(),f);   // pass to algorithm

    cout << v[0] << '\n';                      // transformed by the lambda
}
```

We get **8**, **7**, **8**, and **8**.

Obviously, **function**s are useful for callbacks, for passing operations as arguments, etc.

11.6 Type Functions

A *type function* is a function that is evaluated at compile-time given a type as its argument or returning a type. The standard library provides a variety of type functions to help library implementers and programmers in general to write code that take advantage of aspects of the language, the standard library, and code in general.

For numerical types, **numeric_limits** from **<limits>** presents a variety of useful information (§12.7). For example:

```
constexpr float min = numeric_limits<float>::min();    // smallest positive float
```

Similarly, object sizes can be found by the built-in **sizeof** operator (§1.5). For example:

```
constexpr int szi = sizeof(int);    // the number of bytes in an int
```

Such type functions are part of C++'s mechanisms for compile-time computation that allow tighter type checking and better performance than would otherwise have been possible. Use of such features is often called *metaprogramming* or (when templates are involved) *template metaprogramming*. Here, I just present two facilities provided by the standard library: **iterator_traits** (§11.6.1) and type predicates (§11.6.2).

11.6.1 iterator_traits

The standard-library **sort()** takes a pair of iterators supposed to define a sequence (Chapter 10). Furthermore, those iterators must offer random access to that sequence, that is, they must be

random-access iterators. Some containers, such as **forward_list**, do not offer that. In particular, a **forward_list** is a singly-linked list so subscripting would be expensive and there is no reasonable way to refer back to a previous element. However, like most containers, **forward_list** offers *forward iterators* that can be used to traverse the sequence by algorithms and **for**-statements (§5.2).

The standard library provides a mechanism, **iterator_traits** that allows us to check which kind of iterator is supported. Given that, we can improve the range **sort()** from §10.7 to accept either a **vector** or a **forward_list**. For example:

```
void test(vector<string>& v, forward_list<int>& lst)
{
    sort(v);    // sort the vector
    sort(lst);  // sort the singly-linked list
}
```

The techniques needed to make that work are generally useful.

First, I write two helper functions that take an extra argument indicating whether they are to be used for random-access iterators or forward iterators. The version taking random-access iterator arguments is trivial:

```
template<typename Ran>                                          // for random-access iterators
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)  // we can subscript into [beg:end)
{
    sort(beg,end);      // just sort it
}
```

The version for forward iterators simply copies the list into a **vector**, sorts, and copies back:

```
template<typename For>                                  // for forward iterators
void sort_helper(For beg, For end, forward_iterator_tag)  // we can traverse [beg:end)
{
    vector<Value_type<For>> v {beg,end};  // initialize a vector from [beg:end)
    sort(v.begin(),v.end());
    copy(v.begin(),v.end(),beg);          // copy the elements back
}
```

Value_type<For>> is the type of **For**'s elements, called it's *value type*. Every standard-library iterator has a member **value_type**. I get the **Value_type<For>>** notation by defining a type alias (§5.7):

```
template<typename C>
    using Value_type = typename C::value_type; // C's value type
```

Thus, **v** is a **vector<X>** where **X** is the element type of the input sequence.

The real "type magic" is in the selection of helper functions:

```
template<typename C>
void sort(C& c)
{
    using Iter = Iterator_type<C>;
    sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

Here, I use two type functions: **Iterator_type<C>** returns the iterator type of **C** (that is, **C::iterator**) and

then `Iterator_category<Iter>{}` constructs a "tag" value indicating the kind of iterator provided:

- `std::random_access_iterator_tag` if `C`'s iterator supports random access.
- `std::forward_iterator_tag` if `C`'s iterator supports forward iteration.

Given that, we can select between the two sorting algorithms at compile time. This technique, called *tag dispatch* is one of several used in the standard library and elsewhere to improve flexibility and performance.

The standard-library support for techniques for using iterators, such as tag dispatch, comes in the form of a simple class template `iterator_traits` from `<iterator>`. This allows simple definitions of the type functions used in `sort()`:

```
template<typename C>
    using Iterator_type = typename C::iterator;    // C's iterator type

template<typename Iter>
    using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;  // Iter's category
```

If you don't want to know what kind of "compile-time type magic" is used to provide the standard-library features, you are free to ignore facilities such as `iterator_traits`. But then you can't use the techniques they support to improve your own code.

11.6.2 Type Predicates

A standard-library type predicate is a simple type function that answers a fundamental question about types. For example:

```
bool b1 = Is_arithmetic<int>();       // yes, int is an arithmetic type
bool b2 = Is_arithmetic<string>();    // no, std::string is not an arithmetic type
```

These predicates are found in `<type_traits>`. Other examples are `is_class`, `is_pod`, `is_literal_type`, `has_virtual_destructor`, and `is_base_of`. They are most useful when we write templates. For example:

```
template<typename Scalar>
class complex {
    Scalar re, im;
public:
    static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
    // ...
};
```

To improve readability compared to using the standard library directly, I defined a type function:

```
template<typename T>
constexpr bool Is_arithmetic()
{
    return std::is_arithmetic<T>::value ;
}
```

Older programs use `::value` directly instead of `()`, but I consider that quite ugly and it exposes implementation details.

11.7 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 33-35 of [Stroustrup,2013].
[2] A library doesn't have to be large or complicated to be useful; §11.1.
[3] A resource is anything that has to be acquired and (explicitly or implicitly) released; §11.2.
[4] Use resource handles to manage resources (RAII); §11.2.
[5] Use **unique_ptr** to refer to objects of polymorphic type; §11.2.1.
[6] Use **shared_ptr** to refer to shared objects; §11.2.1.
[7] Prefer resource handles with specific semantics to smart pointers; §11.2.1.
[8] Prefer **unique_ptr** to **shared_ptr**; §4.6.4, §11.2.1.
[9] Prefer smart pointers to garbage collection; §4.6.4, §11.2.1.
[10] Use **array** where you need a sequence with a **constexpr** size; §11.3.1.
[11] Prefer **array** over built-in arrays; §11.3.1.
[12] Use **bitset** if you need **N** bits and **N** is not necessarily the number of bits in a built-in integer type; §11.3.2.
[13] When using **pair**, consider **make_pair()** for type deduction; §11.3.3.
[14] When using **tuple**, consider **make_tuple()** for type deduction; §11.3.3.
[15] Time your programs before making claims about efficiency; §11.4.
[16] Use **duration_cast** to report time measurements with proper units; §11.4.
[17] Often, a lambda is an alternative to using **bind()** or **mem_fn()**; §11.5.
[18] Use **bind()** to create variants of functions and function objects; §11.5.1.
[19] Use **mem_fn()** to create function objects that can invoke a member function when called using the traditional function call notation; §11.5.2.
[20] Use **function** when you need to store something that can be called; §11.5.3.
[21] You can write code to explicitly depend on properties of types; §11.6.

This page intentionally left blank

12

Numerics

The purpose of computing is insight, not numbers.
– R. W. Hamming

... but for the student,
numbers are often the best road to insight.
– A. Ralston

- Introduction
- Mathematical Functions
- Numerical Algorithms
- Complex Numbers
- Random Numbers
- Vector Arithmetic
- Numeric Limits
- Advice

12.1 Introduction

C++ was not designed primarily with numeric computation in mind. However, numeric computation typically occurs in the context of other work – such as database access, networking, instrument control, graphics, simulation, and financial analysis – so C++ becomes an attractive vehicle for computations that are part of a larger system. Furthermore, numeric methods have come a long way from being simple loops over vectors of floating-point numbers. Where more complex data structures are needed as part of a computation, C++'s strengths become relevant. The net effect is that C++ is widely used for scientific, engineering, financial, and other computation involving sophisticated numerics. Consequently, facilities and techniques supporting such computation have emerged. This chapter describes the parts of the standard library that support numerics.

12.2 Mathematical Functions

In **<cmath>**, we find the *standard mathematical functions*, such as **sqrt()**, **log()**, and **sin()** for arguments of type **float**, **double**, and **long double**:

Standard Mathematical Functions	
abs(x)	Absolute value
ceil(x)	Smallest integer >= **x**
floor(x)	Largest integer <= **x**
sqrt(x)	Square root; **x** must be non-negative
cos(x)	Cosine
sin(x)	Sine
tan(x)	Tangent
acos(x)	Arccosine; the result is non-negative
asin(x)	Arcsine; the result nearest to 0 is returned
atan(x)	Arctangent
sinh(x)	Hyperbolic sine
cosh(x)	Hyperbolic cosine
tanh(x)	Hyperbolic tangent
exp(x)	Base e exponential
log(x)	Natural logarithm, base e; **x** must be positive
log10(x)	Base 10 logarithm

The versions for **complex** (§12.4) are found in **<complex>**. For each function, the return type is the same as the argument type.

Errors are reported by setting **errno** from **<cerrno>** to **EDOM** for a domain error and to **ERANGE** for a range error. For example:

```
void f()
{
    errno = 0; // clear old error state
    sqrt(−1);
    if (errno==EDOM)
        cerr << "sqrt() not defined for negative argument";

    errno = 0; // clear old error state
    pow(numeric_limits<double>::max(),2);
    if (errno == ERANGE)
        cerr << "result of pow() too large to represent as a double";
}
```

A few more mathematical functions are found in **<cstdlib>** and there is a separate ISO standard for *special mathematical functions* [C++Math,2010].

12.3 Numerical Algorithms

In <numeric>, we find a small set of generalized numerical algorithms, such as accumulate().

Numerical Algorithms (§iso.26.7)	
x=accumulate(b,e,i)	x is the sum of i and the elements of [b:e)
x=accumulate(b,e,i,f)	accumulate using f instead of +
x=inner_product(b,e,b2,i)	x is the inner product of [b:e) and [b2:b2+(e−b)), that is, the sum of i and (∗p1)∗(∗p2) for each p1 in [b:e) and the corresponding p2 in [b2:b2+(e−b))
x=inner_product(b,e,b2,i,f,f2)	inner_product using f and f2 instead of + and ∗
p=partial_sum(b,e,out)	Element i of [out:p) is the sum of elements [b:b+i]
p=partial_sum(b,e,out,f)	partial_sum using f instead of +
p=adjacent_difference(b,e,out)	Element i of [out:p) is (∗b+i)−∗(b+i−1) for i>0; if e−b>0, then ∗out is ∗b
p=adjacent_difference(b,e,out,f)	adjacent_difference using f instead of −
iota(b,e,v)	For each element in [b:e) assign ++v; thus the sequence becomes v+1, v+2, ...

These algorithms generalize common operations such as computing a sum by letting them apply to all kinds of sequences and by making the operation applied to elements of those sequences a parameter. For each algorithm, the general version is supplemented by a version applying the most common operator for that algorithm. For example:

```
void f()
{
    list<double> lst {1, 2, 3, 4, 5, 9999.99999};
    auto s = accumulate(lst.begin(),lst.end(),0.0); // calculate the sum
    cout << s << '\n';                              // print 10014.9999
}
```

These algorithms work for every standard-library sequence and can have operations supplied as arguments (§12.3).

12.4 Complex Numbers

The standard library supports a family of complex number types along the lines of the complex class described in §4.2.1. To support complex numbers where the scalars are single-precision floating-point numbers (float s), double-precision floating-point numbers (double s), etc., the standard library complex is a template:

```
template<typename Scalar>
class complex {
public:
    complex(const Scalar& re ={}, const Scalar& im ={});
    // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
void f(complex<float> fl, complex<double> db)
{
        complex<long double> ld {fl+sqrt(db)};
        db += fl*3;
        fl = pow(1/fl,2);
        // ...
}
```

The **sqrt()** and **pow()** (exponentiation) functions are among the usual mathematical functions defined in **<complex>** (§12.2).

12.5 Random Numbers

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of application areas is reflected in the wide selection of random number generators provided by the standard library in **<random>**. A random number generator consists of two parts:

[1] an *engine* that produces a sequence of random or pseudo-random values.

[2] a *distribution* that maps those values into a mathematical distribution in a range.

Examples of distributions are **uniform_int_distribution** (where all integers produced are equally likely), **normal_distribution** (“the bell curve”), and **exponential_distribution** (exponential growth); each for some specified range. For example:

```
using my_engine = default_random_engine;          // type of engine
using my_distribution = uniform_int_distribution<>;  // type of distribution

my_engine re {};                                  // the default engine
my_distribution one_to_six {1,6};                 // distribution that maps to the ints 1..6
auto die = bind(one_to_six,re);                   // make a generator

int x = die();                                    // roll the die: x becomes a value in [1:6]
```

The standard-library function **bind()** makes a function object that will invoke its first argument (here, **one_to_six**) given its second argument (here, **re**) as its argument (§11.5.1). Thus a call **die()** is equivalent to a call **one_to_six(re)**.

Thanks to its uncompromising attention to generality and performance one expert has deemed the standard-library random number component “what every random number library wants to be when it grows up.” However, it can hardly be deemed “novice friendly.” The **using** statements makes what is being done a bit more obvious. Instead, I could just have written:

```
auto die = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```

Which version is the more readable depends entirely on the context and the reader.

For novices (of any background) the fully general interface to the random number library can be a serious obstacle. A simple uniform random number generator is often sufficient to get started. For example:

```
Rand_int rnd {1,10};        // make a random number generator for [1:10]
int x = rnd();              // x is a number in [1:10]
```

So, how could we get that? We have to get something like **die()** inside a class **Rand_int**:

```
class Rand_int {
public:
    Rand_int(int low, int high) :dist{low,high} { }
    int operator()() { return dist(re); }        // draw an int
private:
    default_random_engine re;
    uniform_int_distribution<> dist;
};
```

That definition is still "expert level," but the *use* of **Rand_int()** is manageable in the first week of a C++ course for novices. For example:

```
int main()
{
    constexpr int max = 8;
    Rand_int rnd {0,max};                    // make a uniform random number generator

    vector<int> histogram(max+1);            // make a vector of appropriate size
    for (int i=0; i!=200; ++i)
        ++histogram[rnd()];                  // fill histogram with the frequencies of numbers [0:max]

    for (int i = 0; i!=histogram.size(); ++i) {   // write out a bar graph
        cout << i << '\t';
        for (int j=0; j!=histogram[i]; ++j) cout << '*';
        cout << endl;
    }
}
```

The output is a (reassuringly boring) uniform distribution (with reasonable statistical variation):

```
0    *********************
1    ****************
2    *******************
3    *******************
4    ****************
5    ***********************
6    **************************
7    ***********
8    **********************
9    *************************
```

There is no standard graphics library for C++, so I use "ASCII graphics." Obviously, there are lots of open source and commercial graphics and GUI libraries for C++, but in this book I restrict myself to ISO standard facilities.

12.6 Vector Arithmetic

The vector described in §9.2 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to vector would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides (in <valarray>) a vector-like template, called valarray, that is less general and more amenable to optimization for numerical computation:

```
template<typename T>
class valarray {
    // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for valarrays. For example:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;        // numeric array operators *, +, /, and =
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

For more details, see §12.6. In particular, valarray offers stride access to help implement multidimensional computations.

12.7 Numeric Limits

In <limits>, the standard library provides classes that describe the properties of built-in types – such as the maximum exponent of a float or the number of bytes in an int; see §12.7. For example, we can assert that a char is signed:

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

Note that the second assert (only) works because numeric_limits<int>::max() is a constexpr function (§1.7).

12.8 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 40 of [Stroustrup,2013].

[2] Numerical problems are often subtle. If you are not 100% certain about the mathematical aspects of a numerical problem, either take expert advice, experiment, or do both; §12.1.

[3] Don't try to do serious numeric computation using only the bare language; use libraries; §12.1.
[4] Consider **accumulate()**, **inner_product()**, **partial_sum()**, and **adjacent_difference()** before you write a loop to compute a value from a sequence; §12.3.
[5] Use **std::complex** for complex arithmetic; §12.4.
[6] Bind an engine to a distribution to get a random number generator; §12.5.
[7] Be careful that your random numbers are sufficiently random; §12.5.
[8] Use **valarray** for numeric computation when run-time efficiency is more important than flexibility with respect to operations and element types; §12.6.
[9] Properties of numeric types are accessible through **numeric_limits**; §12.7.
[10] Use **numeric_limits** to check that the numeric types are adequate for their use; §12.7.

This page intentionally left blank

13

Concurrency

Keep it simple:
as simple as possible,
but no simpler.
– A. Einstein

- Introduction
- Tasks and **thread**s
- Passing Arguments
- Returning Results
- Sharing Data
- Waiting for Events
- Communicating Tasks
  future and **promise**; **packaged_task**; **async()**
- Advice

13.1 Introduction

Concurrency – the execution of several tasks simultaneously – is widely used to improve throughput (by using several processors for a single computation) or to improve responsiveness (by allowing one part of a program to progress while another is waiting for a response). All modern programming languages provide support for this. The support provided by the C++ standard library is a portable and type-safe variant of what has been used in C++ for more than 20 years and is almost universally supported by modern hardware. The standard-library support is primarily aimed at supporting systems-level concurrency rather than directly providing sophisticated higher-level concurrency models; those can be supplied as libraries built using the standard-library facilities.

The standard library directly supports concurrent execution of multiple threads in a single address space. To allow that, C++ provides a suitable memory model and a set of atomic operations. The atomic operations allows lock-free programming [Dechev,2012]. The memory model

ensures that as long as a programmer avoids data races (uncontrolled concurrent access to mutable data), everything works as one would naively expect. However, most users will see concurrency only in terms of the standard library and libraries built on top of that. This section briefly gives examples of the main standard-library concurrency support facilities: **thread**s, **mutex**es, **lock()** operations, **packaged_task**s, and **future**s. These features are built directly upon what operating systems offer and do not incur performance penalties compared with those. Neither do they guarantee significant performance improvements compared to what the operating system offers.

Do not consider concurrency a panacea. If a task can be done sequentially, it is often simpler and faster to do so.

13.2 Tasks and threads

We call a computation that can potentially be executed concurrently with other computations a *task*. A *thread* is the system-level representation of a task in a program. A task to be executed concurrently with other tasks is launched by constructing a **std::thread** (found in **<thread>**) with the task as its argument. A task is a function or a function object:

```
void f();                   // function

struct F {                  // function object
    void operator()();      // F's call operator (§5.5)
};

void user()
{
    thread t1 {f};          // f() executes in separate thread
    thread t2 {F()};        // F()() executes in separate thread

    t1.join();              // wait for t1
    t2.join();              // wait for t2
}
```

The **join()**s ensure that we don't exit **user()** until the threads have completed. To "join" a **thread** means to "wait for the thread to terminate."

Threads of a program share a single address space. In this, threads differ from processes, which generally do not directly share data. Since threads share an address space, they can communicate through shared objects (§13.5). Such communication is typically controlled by locks or other mechanisms to prevent data races (uncontrolled concurrent access to a variable).

Programming concurrent tasks can be *very* tricky. Consider possible implementations of the tasks **f** (a function) and **F** (a function object):

```
void f() { cout << "Hello "; }

struct F {
    void operator()() { cout << "Parallel World!\n"; }
};
```

This is an example of a bad error: Here, **f** and **F()** each use the object **cout** without any form of

synchronization. The resulting output would be unpredictable and could vary between different executions of the program because the order of execution of the individual operations in the two tasks is not defined. The program may produce "odd" output, such as

```
PaHeralllel o World!
```

When defining tasks of a concurrent program, our aim is to keep tasks completely separate except where they communicate in simple and obvious ways. The simplest way of thinking of a concurrent task is as a function that happens to run concurrently with its caller. For that to work, we just have to pass arguments, get a result back, and make sure that there is no use of shared data in between (no data races).

13.3 Passing Arguments

Typically, a task needs data to work upon. We can easily pass data (or pointers or references to the data) as arguments. Consider:

```
void f(vector<double>& v);      // function do something with v

struct F {                      // function object: do something with v
        vector<double>& v;
        F(vector<double>& vv) :v{vv} { }
        void operator()();      // application operator; §5.5
};

int main()
{
        vector<double> some_vec {1,2,3,4,5,6,7,8,9};
        vector<double> vec2 {10,11,12,13,14};

        thread t1 {f,ref(some_vec)};    // f(some_vec) executes in a separate thread
        thread t2 {F{vec2}};            // F(vec2)() executes in a separate thread

        t1.join();
        t2.join();
}
```

Obviously, `F{vec2}` saves a reference to the argument vector in `F`. `F` can now use that vector and hopefully no other task accesses `vec2` while `F` is executing. Passing `vec2` by value would eliminate that risk.

The initialization with `{f,ref(some_vec)}` uses a `thread` variadic template constructor that can accept an arbitrary sequence of arguments (§5.6). The `ref()` is a type function from `<functional>` that unfortunately is needed to tell the variadic template to treat `some_vec` as a reference, rather than as an object. The compiler checks that the first argument can be invoked given the following arguments and builds the necessary function object to pass to the thread. Thus, if `F::operator()()` and `f()` perform the same algorithm, the handling of the two tasks are roughly equivalent: in both cases, a function object is constructed for the `thread` to execute.

13.4 Returning Results

In the example in §13.3, I pass the arguments by non-**const** reference. I only do that if I expect the task to modify the value of the data referred to (§1.8). That's a somewhat sneaky, but not uncommon, way of returning a result. A less obscure technique is to pass the input data by **const** reference and to pass the location of a place to deposit the result as a separate argument:

```
void f(const vector<double>& v, double* res);      // take input from v;  place result in *res

class F {
public:
        F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
        void operator()();                  // place result in *res
private:
        const vector<double>& v;            // source of input
        double* res;                        // target for output
};

int main()
{
        vector<double> some_vec;
        vector<double> vec2;
        // ...

        double res1;
        double res2;

        thread t1 {f,cref(some_vec),&res1};      // f(some_vec,&res1) executes in a separate thread
        thread t2 {F{vec2,&res2}};               // F{vec2,&res2}() executes in a separate thread

        t1.join();
        t2.join();

        cout << res1 << ' ' << res2 << '\n';
}
```

This works and the technique is very common, but I don't consider returning results through arguments particularly elegant, so I return to this topic in §13.7.1.

13.5 Sharing Data

Sometimes tasks need to share data. In that case, the access has to be synchronized so that at most one task at a time has access. Experienced programmers will recognize this as a simplification (e.g., there is no problem with many tasks simultaneously reading immutable data), but consider how to ensure that at most one task at a time has access to a given set of objects.

The fundamental element of the solution is a **mutex**, a "mutual exclusion object." A **thread** acquires a mutex using a **lock()** operation:

```
mutex m; // controlling mutex
int sh;     // shared data

void f()
{
        unique_lock<mutex> lck {m}; // acquire mutex
        sh += 7;                    // manipulate shared data
}       // release mutex implicitly
```

The **unique_lock**'s constructor acquires the mutex (through a call **m.lock()**). If another thread has already acquired the mutex, the thread waits (''blocks'') until the other thread completes its access. Once a thread has completed its access to the shared data, the **unique_lock** releases the **mutex** (with a call **m.unlock()**). When a **mutex** is released, **thread**s waiting for it resume executing (''are woken up''). The mutual exclusion and locking facilities are found in **<mutex>**.

The correspondence between the shared data and a **mutex** is conventional: the programmer simply has to know which **mutex** is supposed to correspond to which data. Obviously, this is error-prone, and equally obviously we try to make the correspondence clear through various language means. For example:

```
class Record {
public:
        mutex rm;
        // ...
};
```

It doesn't take a genius to guess that for a **Record** called **rec**, **rec.rm** is a **mutex** that you are supposed to acquire before accessing the other data of **rec**, though a comment or a better name might have helped a reader.

It is not uncommon to need to simultaneously access several resources to perform some action. This can lead to deadlock. For example, if **thread1** acquires **mutex1** and then tries to acquire **mutex2** while **thread2** acquires **mutex2** and then tries to acquire **mutex1**, then neither task will ever proceed further. The standard library offers help in the form of an operation for acquiring several locks simultaneously:

```
void f()
{
        // ...
        unique_lock<mutex> lck1 {m1,defer_lock};    // defer_lock: don't yet try to acquire the mutex
        unique_lock<mutex> lck2 {m2,defer_lock};
        unique_lock<mutex> lck3 {m3,defer_lock};
        // ...
        lock(lck1,lck2,lck3);                       // acquire all three locks
        // ... manipulate shared data ...
} // implicitly release all mutexes
```

This **lock()** will proceed only after acquiring all its **mutex** arguments and will never block (''go to sleep'') while holding a **mutex**. The destructors for the individual **unique_lock**s ensure that the **mutex**es are released when a **thread** leaves the scope.

Communicating through shared data is pretty low level. In particular, the programmer has to devise ways of knowing what work has and has not been done by various tasks. In that regard, use of shared data is inferior to the notion of call and return. On the other hand, some people are convinced that sharing must be more efficient than copying arguments and returns. That can indeed be so when large amounts of data are involved, but locking and unlocking are relatively expensive operations. On the other hand, modern machines are very good at copying data, especially compact data, such as **vector** elements. So don't choose shared data for communication because of "efficiency" without thought and preferably not without measurement.

13.6 Waiting for Events

Sometimes, a **thread** needs to wait for some kind of external event, such as another **thread** completing a task or a certain amount of time having passed. The simplest "event" is simply time passing. Using the time facilities found in **<chrono>** I can write:

```
using namespace std::chrono;      // see §11.4

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_cast<nanoseconds>(t1−t0).count() << " nanoseconds passed\n";
```

Note that I didn't even have to launch a **thread**; by default, **this_thread** refers to the one and only thread.

I used **duration_cast** to adjust the clock's units to the nanoseconds I wanted.

The basic support for communicating using external events is provided by **condition_variable**s found in **<condition_variable>**. A **condition_variable** is a mechanism allowing one **thread** to wait for another. In particular, it allows a **thread** to wait for some *condition* (often called an *event*) to occur as the result of work done by other **thread**s.

Using **condition_variable**s supports many forms of elegant and efficient sharing, but can be rather tricky. Consider the classical example of two **thread**s communicating by passing messages through a **queue**. For simplicity, I declare the **queue** and the mechanism for avoiding race conditions on that **queue** global to the producer and consumer:

```
class Message {      // object to be communicated
    // ...
};

queue<Message> mqueue;          // the queue of messages
condition_variable mcond;       // the variable communicating events
mutex mmutex;                   // the locking mechanism
```

The types **queue**, **condition_variable**, and **mutex** are provided by the standard library.

The **consumer()** reads and processes **Message**s:

```
void consumer()
{
    while(true) {
        unique_lock<mutex> lck{mmutex};        // acquire mmutex
        while (mcond.wait(lck)) /* do nothing */;  // release lck and wait;
                                                   // re-acquire lck upon wakeup
        auto m = mqueue.front();               // get the message
        mqueue.pop();
        lck.unlock();                          // release lck
        // ... process m ...
    }
}
```

Here, I explicitly protect the operations on the **queue** and on the **condition_variable** with a **unique_lock** on the **mutex**. Waiting on **condition_variable** releases its lock argument until the wait is over (so that the queue is non-empty) and then reacquires it.

The corresponding **producer** looks like this:

```
void producer()
{
    while(true) {
        Message m;
        // ... fill the message ...
        unique_lock<mutex> lck {mmutex};       // protect operations
        mqueue.push(m);
        mcond.notify_one();                    // notify
    }                                          // release lock (at end of scope)
}
```

13.7 Communicating Tasks

The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks (work to potentially be done concurrently) rather than directly at the lower level of threads and locks:

[1] **future** and **promise** for returning a value from a task spawned on a separate thread
[2] **packaged_task** to help launch tasks and connect up the mechanisms for returning a result
[3] **async()** for launching of a task in a manner very similar to calling a function.

These facilities are found in **<future>**.

13.7.1 future and promise

The important point about **future** and **promise** is that they enable a transfer of a value between two tasks without explicit use of a lock; "the system" implements the transfer efficiently. The basic idea is simple: When a task wants to pass a value to another, it puts the value into a **promise**. Somehow, the implementation makes that value appear in the corresponding **future**, from which it can be read (typically by the launcher of the task). We can represent this graphically:

If we have a **future<X>** called **fx**, we can **get()** a value of type **X** from it:

```
X v = fx.get();   // if necessary, wait for the value to get computed
```

If the value isn't there yet, our thread is blocked until it arrives. If the value couldn't be computed, **get()** might throw an exception (from the system or transmitted from the task from which we were trying to **get()** the value).

The main purpose of a **promise** is to provide simple "put" operations (called **set_value()** and **set_exception()**) to match **future**'s **get()**. The names "future" and "promise" are historical; please don't blame or credit me. They are yet another fertile source of puns.

If you have a **promise** and need to send a result of type **X** to a **future**, you can do one of two things: pass a value or pass an exception. For example:

```
void f(promise<X>& px)  // a task: place the result in px
{
        // ...
        try {
                X res;
                // ... compute a value for res ...
                px.set_value(res);
        }
        catch (...) {           // oops: couldn't compute res
                px.set_exception(current_exception());          // pass the exception to the future's thread
        }
}
```

The **current_exception()** refers to the caught exception.

To deal with an exception transmitted through a **future**, the caller of **get()** must be prepared to catch it somewhere. For example:

```
void g(future<X>& fx)           // a task: get the result from fx
{
        // ...
        try {
                X v = fx.get();  // if necessary, wait for the value to get computed
                // ... use v ...
        }
        catch (...) {           // oops: someone couldn't compute v
                // ... handle error ...
        }
}
```

If the error doesn't need to be handled by **g()** itself, the code reduces to the minimal:

```
void g(future<X>& fx)          // a task: get the result from fx
{
    // ...
    X v = fx.get();   // if necessary, wait for the value to get computed
    // ... use v ...
}
```

13.7.2 packaged_task

How do we get a **future** into the task that needs a result and the corresponding **promise** into the thread that should produce that result? The **packaged_task** type is provided to simplify setting up tasks connected with **future**s and **promise**s to be run on **thread**s. A **packaged_task** provides wrapper code to put the return value or exception from the task into a **promise** (like the code shown in §13.7.1). If you ask it by calling **get_future**, a **packaged_task** will give you the **future** corresponding to its **promise**. For example, we can set up two tasks to each add half of the elements of a **vector<double>** using the standard-library **accumulate()** (§12.3):

```
double accum(double* beg, double* end, double init)
    // compute the sum of [beg:end) starting with the initial value init
{
    return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
    using Task_type = double(double*,double*,double);          // type of task

    packaged_task<Task_type> pt0 {accum};                      // package the task (i.e., accum)
    packaged_task<Task_type> pt1 {accum};

    future<double> f0 {pt0.get_future()};                      // get hold of pt0's future
    future<double> f1 {pt1.get_future()};                      // get hold of pt1's future

    double* first = &v[0];
    thread t1 {move(pt0),first,first+v.size()/2,0};            // start a thread for pt0
    thread t2 {move(pt1),first+v.size()/2,first+v.size(),0};   // start a thread for pt1

    // ...

    return f0.get()+f1.get();                          // get the results
}
```

The **packaged_task** template takes the type of the task as its template argument (here **Task_type**, an alias for **double(double*,double*,double)**) and the task as its constructor argument (here, **accum**). The **move()** operations are needed because a **packaged_task** cannot be copied. The reason that a **packaged_task** cannot be copied is that it is a resource handle: it owns its **promise** and is (indirectly) responsible for whatever resoures its task may own.

Please note the absence of explicit mention of locks in this code: we are able to concentrate on tasks to be done, rather than on the mechanisms used to manage their communication. The two tasks will be run on separate threads and thus potentially in parallel.

13.7.3 async()

The line of thinking I have pursued in this chapter is the one I believe to be the simplest yet still among the most powerful: Treat a task as a function that may happen to run concurrently with other tasks. It is far from the only model supported by the C++ standard library, but it serves well for a wide range of needs. More subtle and tricky models, e.g., styles of programming relying on shared memory, can be used as needed.

To launch tasks to potentially run asynchronously, we can use **async()**:

```
double comp4(vector<double>& v)
    // spawn many tasks if v is large enough
{
    if (v.size()<10000)         // is it worth using concurrency?
        return accum(v.begin(),v.end(),0.0);

    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum,v0,v0+sz/4,0.0);              // first quarter
    auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);         // second quarter
    auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0);       // third quarter
    auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);         // fourth quarter

    return f0.get()+f1.get()+f2.get()+f3.get();  // collect and combine the results
}
```

Basically, **async()** separates the "call part" of a function call from the "get the result part," and separates both from the actual execution of the task. Using **async()**, you don't have to think about threads and locks. Instead, you think just in terms of tasks that potentially compute their results asynchronously. There is an obvious limitation: Don't even think of using **async()** for tasks that share resources needing locking – with **async()** you don't even know how many **thread**s will be used because that's up to **async()** to decide based on what it knows about the system resources available at the time of a call. For example, **async()** may check whether any idle cores (processors) are available before deciding how many **thread**s to use.

Using a guess about the cost of computation relative to the cost of launching a **thread**, such as **v.size()<10000**, is very primitive and prone to gross mistakes about performance. However, this is not the place for a proper disussion about how to manage **thread**s. Don't take this estimate as more than a simple and probably poor guess.

Please note that **async()** is not just a mechanism specialized for parallel computation for increased performance. For example, it can also be used to spawn a task for getting information from a user, leaving the "main program" active with something else (§13.7.3).

13.8 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 41-42 of [Stroustrup,2013].
[2] Use concurrency to improve responsiveness or to improve throughput; §13.1.
[3] Work at the highest level of abstraction that you can afford; §13.1.
[4] Consider processes as an alternative to threads; §13.1.
[5] The standard-library concurrency facilities are type safe; §13.1.
[6] The memory model exists to save most programmers from having to think about the machine architecture level of computers; §13.1.
[7] The memory model makes memory appear roughly as naively expected; §13.1.
[8] Atomics allow for lock-free programming; §13.1.
[9] Leave lock-free programming to experts; §13.1.
[10] Sometimes, a sequential solution is simpler and faster than a concurrent solution; §13.1.
[11] Avoid data races; §13.1, §13.2.
[12] A **thread** is a type-safe interface to a system thread; §13.2.
[13] Use **join()** to wait for a **thread** to complete; §13.2.
[14] Avoid explicitly shared data whenever you can; §13.2.
[15] Use **unique_lock** to manage mutexes; §13.5.
[16] Use **lock()** to acquire multiple locks; §13.5.
[17] Use **condition_variable**s to manage communication among **thread**s; §13.6.
[18] Think in terms of tasks that can be executed concurrently, rather than directly in terms of **thread**s; §13.7.
[19] Value simplicity; §13.7.
[20] Prefer **packaged_task** and **future**s over direct use of **thread**s and **mutex**es; §13.7.
[21] Return a result using a **promise** and get a result from a **future**; §13.7.1.
[22] Use **packaged_task**s to handle exceptions thrown by tasks and to arrange for value return; §13.7.2.
[23] Use a **packaged_task** and a **future** to express a request to an external service and wait for its response; §13.7.2.
[24] Use **async()** to launch simple tasks; §13.7.3.

This page intentionally left blank

14

History and Compatibility

Hurry Slowly
(festina lente).
– Octavius, Caesar Augustus

- History
  Timeline; The Early Years; The ISO C++ Standards
- C++11 Extensions
  Language Features; Standard-Library Components; Deprecated Features; Casts
- C/C++ Compatibility
  C and C++ Are Siblings; Compatibility Problems
- Bibliography
- Advice

14.1 History

I invented C++, wrote its early definitions, and produced its first implementation. I chose and formulated the design criteria for C++, designed its major language features, developed or helped to develop many of the early libraries, and was responsible for the processing of extension proposals in the C++ standards committee.

C++ was designed to provide Simula's facilities for program organization [Dahl,1970] together with C's efficiency and flexibility for systems programming [Kernighan,1978]. Simula is the initial source of C++'s abstraction mechanisms. The class concept (with derived classes and virtual functions) was borrowed from it. However, templates and exceptions came to C++ later with different sources of inspiration.

The evolution of C++ was always in the context of its use. I spent a lot of time listening to users and seeking out the opinions of experienced programmers. In particular, my colleagues at AT&T Bell Laboratories were essential for the growth of C++ during its first decade.

This section is a brief overview; it does not try to mention every language feature and library component. Furthermore, it does not go into details. For more information, and in particular for more names of people who contributed, see [Stroustrup,1993], [Stroustrup,2007], and [Stroustrup,1994]. My two papers from the ACM History of Programming Languages conference and my *Design and Evolution of C++* book (known as "D&E") describe the design and evolution of C++ in detail and document influences from other programming languages.

Most of the documents produced as part of the ISO C++ standards effort are available online [WG21]. In my FAQ, I try to maintain a connection between the standard facilities and the people who proposed and refined those facilities [Stroustrup,2010]. C++ is not the work of a faceless, anonymous committee or of a supposedly omnipotent "dictator for life"; it is the work of many dedicated, experienced, hard-working individuals.

14.1.1 Timeline

The work that led to C++ started in the fall of 1979 under the name "C with Classes." Here is a simplified timeline:

1979 Work on "C with Classes" started. The initial feature set included classes and derived classes, public/private access control, constructors and destructors, and function declarations with argument checking. The first library supported non-preemptive concurrent tasks and random number generators.

1984 "C with Classes" was renamed to C++. By then, C++ had acquired virtual functions, function and operator overloading, references, and the I/O stream and complex number libraries.

1985 First commercial release of C++ (October 14). The library included I/O streams, complex numbers, and tasks (non-preemptive scheduling).

1985 *The C++ Programming Language* ("TC++PL," October 14) [Stroustrup,1986].

1989 *The Annotated C++ Reference Manual* ("the ARM") [Ellis,1989].

1991 *The C++ Programming Language, Second Edition* [Stroustrup,1991], presenting generic programming using templates and error handling based on exceptions (including the "Resource Acquisition Is Initialization" general resource management idiom).

1997 *The C++ Programming Language, Third Edition* [Stroustrup,1997] introduced ISO C++, including namespaces, **dynamic_cast**, and many refinements of templates. The standard library added the STL framework of generic containers and algorithms.

1998 ISO C++ standard [C++,1998].

2002 Work on a revised standard, colloquially named C++0x, started.

2003 A "bug fix" revision of the ISO C++ standard was issued. A C++ Technical Report introduced new standard-library components, such as regular expressions, unordered containers (hash tables), and resource management pointers, which later became part of C++0x.

2006 An ISO C++ Technical Report on Performance was issued to answer questions of cost, predictability, and techniques, mostly related to embedded systems programming [C++,2004].

2009 C++0x was feature complete. It provided uniform initialization, move semantics, variadic template arguments, lambda expressions, type aliases, a memory model suitable for

concurrency, and much more. The standard library added several components, including threads, locks, and most of the components from the 2003 Technical Report.

2011 ISO C++11 standard was formally approved [C++,2011].

2012 Work on future ISO C++ standards (referred to as C++14 and C++17) started.

2013 The first complete C++11 implementations emerged.

2013 *The C++ Programming Language, Fourth Edition* introduced C++11.

During development, C++11 was known as C++0x. As is not uncommon in large projects, we were overly optimistic about the completion date.

14.1.2 The Early Years

I originally designed and implemented the language because I wanted to distribute the services of a UNIX kernel across multiprocessors and local-area networks (what are now known as multicores and clusters). For that, I needed some event-driven simulations for which Simula would have been ideal, except for performance considerations. I also needed to deal directly with hardware and provide high-performance concurrent programming mechanisms for which C would have been ideal, except for its weak support for modularity and type checking. The result of adding Simula-style classes to C (Classic C; §14.3.1), "C with Classes," was used for major projects in which its facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions, templates, exceptions, and many, many details [Stroustrup,1982]. The first use of C++ outside a research organization started in July 1983.

The name C++ (pronounced "see plus plus") was coined by Rick Mascitti in the summer of 1983 and chosen as the replacement for "C with Classes" by me. The name signifies the evolutionary nature of the changes from C; "++" is the C increment operator. The slightly shorter name "C+" is a syntax error; it had also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language was not called D, because it was an extension of C, because it did not attempt to remedy problems by removing features, and because there already existed several would-be C successors named D. For yet another interpretation of the name C++, see the appendix of [Orwell,1949].

C++ was designed primarily so that my friends and I would not have to program in assembler, C, or various then-fashionable high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer. In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously. There was no "C++ project" either, or a "C++ design committee." Throughout, C++ evolved to cope with problems encountered by users and as a result of discussions among my friends, my colleagues, and me.

The very first design of C++ (then called "C with Classes") included function declarations with argument type checking and implicit conversions, classes with the **public**/**private** distinction between the interface and the implementation, derived classes, and constructors and destructors. I used macros to provide primitive parameterization. This was in non-experimental use by mid-1980. Late that year, I was able to present a set of language facilities supporting a coherent set of programming styles. In retrospect, I consider the introduction of constructors and destructors most significant. In the terminology of the time, "a constructor creates the execution environment for the member functions and the destructor reverses that." Here is the root of C++'s strategies for

resource management (causing a demand for exceptions) and the key to many techniques for making user code short and clear. If there were other languages at the time that supported multiple constructors capable of executing general code, I didn't (and don't) know of them. Destructors were new in C++.

C++ was released commercially in October 1985. By then, I had added inlining (§1.4, §4.2.1), **const**s (§1.7), function overloading (§1.4), references (§1.8), operator overloading (§4.2.1), and virtual functions (§4.4). Of these features, support for run-time polymorphism in the form of virtual functions was by far the most controversial. I knew its worth from Simula but found it impossible to convince most people in the systems programming world of its value. Systems programmers tended to view indirect function calls with suspicion, and people acquainted with other languages supporting object-oriented programming had a hard time believing that **virtual** functions could be fast enough to be useful in systems code. Conversely, many programmers with an object-oriented background had (and many still have) a hard time getting used to the idea that you use virtual function calls only to express a choice that must be made at run time. The resistance to virtual functions may be related to a resistance to the idea that you can get better systems through more regular structure of code supported by a programming language. Many C programmers seem convinced that what really matters is complete flexibility and careful individual crafting of every detail of a program. My view was (and is) that we need every bit of help we can get from languages and tools: the inherent complexity of the systems we are trying to build is always at the edge of what we can express.

Much of the design of C++ was done on the blackboards of my colleagues. In the early years, the feedback from Stu Feldman, Alexander Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy, and Dennis Ritchie was invaluable.

In the second half of the 1980s, I continued to add language features in response to user comments. The most important of those were templates [Stroustrup,1988] and exception handling [Koenig,1990], which were considered experimental at the time the standards effort started. In the design of templates, I was forced to decide among flexibility, efficiency, and early type checking. At the time, nobody knew how to simultaneously get all three. To compete with C-style code for demanding systems applications, I felt that I had to choose the first two properties. In retrospect, I think the choice was the correct one, and the search for better type checking of templates continues [DosReis,2006] [Gregor,2006] [Sutton,2011] [Stroustrup,2012a]. The design of exceptions focused on multilevel propagation of exceptions, the passing of arbitrary information to an error handler, and the integration between exceptions and resource management by using local objects with destructors to represent and release resources (what I clumsily called *Resource Acquisition Is Initialization*; §4.2.2).

I generalized C++'s inheritance mechanisms to support multiple base classes [Stroustrup,1987a]. This was called *multiple inheritance* and was considered difficult and controversial. I considered it far less important than templates or exceptions. Multiple inheritance of abstract classes (often called *interfaces*) is now universal in languages supporting static type checking and object-oriented programming.

The C++ language evolved hand in hand with some of the key library facilities presented in this book. For example, I designed the complex [Stroustrup,1984], vector, stack, and (I/O) stream [Stroustrup,1985] classes together with the operator overloading mechanisms. The first string and list classes were developed by Jonathan Shopiro and me as part of the same effort. Jonathan's

string and list classes were the first to see extensive use as part of a library. The string class from the standard C++ library has its roots in these early efforts. The task library described in [Stroustrup,1987b] was part of the first "C with Classes" program ever written in 1980. I wrote it and its associated classes to support Simula-style simulations. Unfortunately, we had to wait until 2011 (30 years!) to get concurrency support standardized and universally available (Chapter 13). The development of the template facility was influenced by a variety of **vector**, **map**, **list**, and **sort** templates devised by Andrew Koenig, Alex Stepanov, me, and others.

The most important innovation in the 1998 standard library was the inclusion of the STL, a framework of algorithms and containers, in the standard library (Chapter 9, Chapter 10). It was the work of Alex Stepanov (with Dave Musser, Meng Lee, and others) based on more than a decade's work on generic programming. The STL has been massively influential within the C++ community and beyond.

C++ grew up in an environment with a multitude of established and experimental programming languages (e.g., Ada [Ichbiah,1979], Algol 68 [Woodward,1974], and ML [Paulson,1996]). At the time, I was comfortable in about 25 languages, and their influences on C++ are documented in [Stroustrup,1994] and [Stroustrup,2007]. However, the determining influences always came from the applications I encountered. That was a deliberate policy to have the development of C++ "problem driven" rather than imitative.

14.1.3 The ISO C++ Standards

The explosive growth of C++ use caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground for a standardization effort [Stroustrup,1994]. The result was a conscious effort to maintain contact between implementers of C++ compilers and major users. This was done through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Labs made a major contribution to C++ and its wider community by allowing me to share drafts of revised versions of the C++ reference manual with implementers and users. Because many of those people worked for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from dozens of organizations read and commented on what became the generally accepted reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual* ("the ARM") [Ellis,1989]. The X3J16 committee of ANSI was convened in December 1989 at the initiative of Hewlett-Packard. In June 1991, this ANSI (American national) standardization of C++ became part of an ISO (international) standardization effort for C++ and named WG21. From 1990, these joint C++ standards committees have been the main forum for the evolution of C++ and the refinement of its definition. I served on these committees throughout. In particular, as the chairman of the working group for extensions (later called the evolution group), I was directly responsible for handling proposals for major changes to C++ and the addition of new language features. An initial draft standard for public review was produced in April 1995. The first ISO C++ standard (ISO/IEC 14882-1998) [C++,1998] was ratified by a 22-0 national vote in 1998. A "bug fix release" of this standard was issued in 2003, so you sometimes hear people refer to C++03, but

that is essentially the same language as C++98.

The current C++, C++11, known for years as C++0x, is the work of the members of WG21. The committee worked under increasingly onerous self-imposed processes and procedures. These processes probably led to a better (and more rigorous) specification, but they also limited innovation [Stroustrup,2007]. An initial draft standard for public review was produced in 2009. The second ISO C++ standard (ISO/IEC 14882-2011) [C++,2011] was ratified by a 21-0 national vote in August 2011.

One reason for the long gap between the two standards is that most members of the committee (including me) were under the mistaken impression that the ISO rules required a "waiting period" after a standard was issued before starting work on new features. Consequently, serious work on new language features did not start until 2002. Other reasons included the increased size of modern languages and their foundation libraries. In terms of pages of standards text, the language grew by about 30% and the standard library by about 100%. Much of the increase was due to more detailed specification, rather than new functionality. Also, the work on a new C++ standard obviously had to take great care not to compromise older code through incompatible changes. There are billions of lines of C++ code in use that the committee must not break.

C++11 added massively to the standard library and pushed to complete the feature set needed for a programming style that is a synthesis of the "paradigms" and idioms that have proven successful with C++98. The overall aims for the C++11 effort were:

- Make C++ a better language for systems programming and library building.
- Make C++ easier to teach and learn.

The aims are documented and detailed in [Stroustrup,2007].

A major effort was made to make concurrent systems programming type-safe and portable. This involved a memory model (§13.1) and a set of facilities for lock-free programming, which is primarily the work of Hans Boehm, Brian McKnight, and others. On top of that, we added the **thread**s library.

14.2 C++11 Extensions

Here, I list the language features and standard-library components that have been added to C++ for the C++11 standard.

14.2.1 Language Features

Looking at a list of language features can be quite bewildering. Remember that a language feature is not meant to be used in isolation. In particular, most features that are new in C++11 make no sense in isolation from the framework provided by older features.

[1] Uniform and general initialization using **{}**-lists (§1.5, §4.2.3)
[2] Type deduction from initializer: **auto** (§1.5)
[3] Prevention of narrowing (§1.5)
[4] Generalized and guaranteed constant expressions: **constexpr** (§1.7)
[5] Range-**for**-statement (§1.8)
[6] Null pointer keyword: **nullptr** (§1.8)

[7] Scoped and strongly typed **enums**: **enum class** (§2.5)
[8] Compile-time assertions: **static_assert** (§3.4.3)
[9] Language mapping of **{}**-list to **std::initializer_list** (§4.2.3)
[10] Rvalue references (enabling move semantics; §4.6.2)
[11] Nested template arguments ending with **>>** (no space between the **>**s)
[12] Lambdas (§5.5)
[13] Variadic templates (§5.6)
[14] Type and template aliases (§5.7)
[15] Unicode characters
[16] **long long** integer type
[17] Alignment controls: **alignas** and **alignof**
[18] The ability to use the type of an expression as a type in a declaration: **decltype**
[19] Raw string literals (§7.3)
[20] Generalized POD (“Plain Old Data”)
[21] Generalized **union**s
[22] Local classes as template arguments
[23] Suffix return type syntax
[24] A syntax for attributes and two standard attributes: **[[carries_dependency]]** and **[[noreturn]]**
[25] Preventing exception propagation: the **noexcept** specifier (§3.4.1)
[26] Testing for the possibility of a **throw** in an expression: the **noexcept** operator.
[27] C99 features: extended integral types (i.e., rules for optional longer integer types); concatenation of narrow/wide strings; **__STDC_HOSTED__**; **_Pragma(X)**; vararg macros and empty macro arguments
[28] **__func__** as the name of a string holding the name of the current function
[29] **inline** namespaces
[30] Delegating constructors
[31] In-class member initializers
[32] Control of defaults: **default** and **delete** (§4.6.5)
[33] Explicit conversion operators
[34] User-defined literals
[35] More explicit control of **template** instantiation: **extern template**s
[36] Default template arguments for function templates
[37] Inheriting constructors
[38] Override controls: **override** and **final** (§4.5.1)
[39] Simpler and more general SFINAE rule
[40] Memory model (§13.1)
[41] Thread-local storage: **thread_local**

For a more complete description of the changes to C++98 in C++11, see [Stroustrup,2013].

14.2.2 Standard-Library Components

The C++11 additions to the standard library come in two forms: new components (such as the regular expression matching library) and improvements to C++98 components (such as move constructors for containers).

[1] initializer_list constructors for containers (§4.2.3)
[2] Move semantics for containers (§4.6.2, §9.2)
[3] A singly-linked list: forward_list (§9.6)
[4] Hash containers: unordered_map, unordered_multimap, unordered_set, and unordered_multiset (§9.6, §9.5)
[5] Resource management pointers: unique_ptr, shared_ptr, and weak_ptr (§11.2.1)
[6] Concurrency support: thread (§13.2), mutexes (§13.5), locks (§13.5), and condition variables (§13.6)
[7] Higher-level concurrency support: packaged_thread, future, promise, and async() (§13.7)
[8] tuples (§11.3.3)
[9] Regular expressions: regex (§7.3)
[10] Random numbers: uniform_int_distribution, normal_distribution, random_engine, etc. (§12.5)
[11] Integer type names, such as int16_t, uint32_t, and int_fast64_t
[12] A fixed-sized contiguous sequence container: array (§11.3.1)
[13] Copying and rethrowing exceptions (§13.7.1)
[14] Error reporting using error codes: system_error
[15] emplace() operations for containers
[16] Wide use of constexpr functions
[17] Systematic use of noexcept functions
[18] Improved function adaptors: function and bind() (§11.5)
[19] string to numeric value conversions
[20] Scoped allocators
[21] Type traits, such as is_integral and is_base_of (§11.6.2)
[22] Time utilities: duration and time_point (§11.4)
[23] Compile-time rational arithmetic: ratio
[24] Abandoning a process: quick_exit
[25] More algorithms, such as move(), copy_if(), and is_sorted() (Chapter 10)
[26] Garbage collection ABI (§4.6.4)
[27] Low-level concurrency support: atomics

14.2.3 Deprecated Features

By deprecating a feature, the standards committee expresses the wish that the feature will go away. However, the committee does not have a mandate to immediately remove a heavily used feature – however redundant or dangerous it may be. Thus, a deprecation is a strong hint to avoid the feature. It may disappear in the future. Compilers are likely to issue warnings for uses of deprecated features. However, deprecated features are part of the standard and history shows that unfortunately they tend to remain supported "forever" for reasons of compatibility.

- Generation of the copy constructor and the copy assignment is deprecated for a class with a destructor.
- It is no longer allowed to assign a string literal to a char*. Instead of char* as a target for assignment and initializations with string literals, use const char* or auto.

- C++98 exception specifications are deprecated:

  ```
  void f() throw(X,Y);  // C++98; now deprecated
  ```

  The support facilities for exception specifications, **unexcepted_handler**, **set_unexpected()**, **get_unexpected()**, and **unexpected()**, are similarly deprecated. Instead, use **noexcept** (§3.4.1).
- Some C++ standard-library function objects and associated functions are deprecated. Most relate to argument binding. Instead use lambdas, **bind**, and **function** (§11.5).
- The **auto_ptr** is deprecated. Instead, use **unique_ptr** (§11.2.1).
- The use of the storage specifier **register** is deprecated.
- The use of **++** on a **bool** is deprecated.

In addition, the committee did remove the essentially unused **export** feature, because it was complex and not shipped by the major vendors.

14.2.4 Casts

C-style casts should have been deprecated in favor of *named casts*. The named casts are:

- **static_cast**: for reasonably well-behaved conversions, such as from a pointer to a base to its derived class.
- **reinterpret_cast**: For really nasty, non-portable conversions, such as conversion of an **int** to a pointer type.
- **const_cast**: For casting away **const**.

For example:

```
Widget* pw = static_cast<Widget*>(pv);                  // pv is a void* supposed to point to a Widget
auto dd = reintrepret_cast<Device_driver*>(0xFF00);     // 0xFF is supposed to point to a device driver
char* pc = const_cast<char*>("Casts are inherently dangerous");
```

A literal starting with **0x** is a hexadecimal (base 16) integer.

Programmers should seriously consider banning C-style casts from their own programs. Where explicit type conversion is necessary, a combination of named casts can do what a C-style cast can. The named casts should be preferred because they are more explicit and more visible.

Expricit type conversion can be completely avoided in most high-level code, so consider every cast (however expressed) a blemish on your design. Consider defining a function **narrow_cast<T>(v)** that checks if the value **v** can be represented as a **T** without loss of information (without *narrowing*) and throws an exception if it cannot For class hierachy navigation, prefer the checked **dynamic_cast** (§4.5.3).

14.3 C/C++ Compatibility

With minor exceptions, C++ is a superset of C (meaning C11; [C11]). Most differences stem from C++'s greater emphasis on type checking. Well-written C programs tend to be C++ programs as well. A compiler can diagnose every difference between C++ and C. The C99/C++11 incompatibilities are listed in §iso.C. At the time of writing, C11 is still very new and most C code is Classic C or C99 [C99].

14.3.1 C and C++ Are Siblings

Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. The resulting incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of libraries and tools for C and C++.

How can I call C and C++ siblings? Clearly, C++ is a descendant of C. However, look at a simplified family tree:

A solid line means a massive inheritance of features, a dashed line a borrowing of major features, and a dotted line a borrowing of minor features. From this, ISO C and ISO C++ emerge as the two major descendants of K&R C [Kernighan,1978], and as siblings. Each carries with it the key aspects of Classic C, and neither is 100% compatible with Classic C. I picked the term "Classic C" from a sticker that used to be affixed to Dennis Ritchie's terminal. It is K&R C plus enumerations and **struct** assignment. BCPL is defined by [Richards,1980] and C89 by [C90].

Incompatibilities are nasty for programmers in part because they create a combinatorial explosion of alternatives. Consider a simple Venn diagram:

The areas are not to scale. Both C++11 and C11 have most of K&R C as a subset. C++11 has most of C11 as a subset. There are features belonging to most of the distinct areas. For example:

C89 only	Call of undeclared function
C99 only	Variable-length arrays (VLAs)
C++ only	Templates
C89 and C99	Algol-style function definitions
C89 and C++	Use of the C99 keyword **restrict** as an identifier
C++ and C99	**//** comments
C89, C++, and C99	**struct**s
C++11 only	Move semantics (using rvalue references; **&&**)
C11 only	Type-generic expressions using the **_Generic** keyword
C++11 and C11	Atomics

Note that differences between C and C++ are not necessarily the result of changes to C made in C++. In several cases, the incompatibilities arise from features adopted incompatibly into C long after they were common in C++. Examples are the ability to assign a **T*** to a **void*** and the linkage of global **const**s [Stroustrup,2002]. Sometimes, a feature was even incompatibly adopted into C after it was part of the ISO C++ standard, such as details of the meaning of **inline**.

14.3.2 Compatibility Problems

There are many minor incompatibilities between C and C++. All can cause problems for a programmer, but all can be coped with in the context of C++. If nothing else, C code fragments can be compiled as C and linked to using the **extern "C"** mechanism.

The major problems for converting a C program to C++ are likely to be:

- Suboptimal design and programming style.
- A **void*** implicitly converted to a **T*** (that is, converted without a cast).
- C++ keywords used as identifiers in C code.
- Incomparible linkage between code fragments compiled as C and code fragments compiled as C++.

14.3.2.1 Style Problems

Natually, a C program is written in a C style, such as the style used in K&R [Kernighan,1988]. This implies widespread use of pointers and arrays, and probably many macros. These facilities are hard to use reliably in a large program. Resource management and error handling are often ad hoc, documented (rather than language and tool supported), and often incompletely documented and adhered to. A simple line-for-line conversion of a C program into a C++ program yields a program that is often a bit better checked. In fact, I have never converted a C program into C++ without finding some bug. However, the fundamental structure is unchanged, and so are the fundamental sources of errors. If you had incomplete error handling, resource leaks, or buffer overflows in the original C program, they will still be there in the C++ version. To obtain major benefits, you must make changes to the fundamental structure of the code:

[1] Don't think of C++ as C with a few features added. C++ can be used that way, but only suboptimally. To get really major advantages from C++ as compared to C, you need to apply different design and implementation styles.

[2] Use the C++ standard library as a teacher of new techniques and programming styles. Note the difference from the C standard library (e.g., **=** rather than **strcpy()** for copying and **==** rather than **strcmp()** for comparing).

[3] Macro substitution is almost never necessary in C++. Use **const** (§1.7), **constexpr** (§1.7), **enum** or **enum class** (§2.5) to define manifest constants, **inline** (§4.2.1) to avoid function-calling overhead, **template**s (Chapter 5) to specify families of functions and types, and **namespace**s (§3.3) to avoid name clashes.

[4] Don't declare a variable before you need it, and initialize it immediately. A declaration can occur anywhere a statement can (§1.9), in **for**-statement initializers (§1.8), and in conditions (§4.5.3).

[5] Don't use **malloc()**. The **new** operator (§4.2.2) does the same job better, and instead of **realloc()**, try a **vector** (§4.2.3, §10.1). Don't just replace **malloc()** and **free()** with "naked" **new** and **delete** (§4.2.2).

[6] Avoid **void***, unions, and casts, except deep within the implementation of some function or class. Their use limits the support you can get from the type system and can harm performance. In most cases, a cast is an indication of a design error.

[7] If you must use an explicit type conversion, use an appropriate named cast (e.g., **static_cast**; §14.2.3) for a more precise statement of what you are trying to do.

[8] Minimize the use of arrays and C-style strings. C++ standard-library **string**s (§7.2), **array**s (§11.3.1), and **vector**s (§9.2) can often be used to write simpler and more maintainable code compared to the traditional C style. In general, try not to build yourself what has already been provided by the standard library.

[9] Avoid pointer arithmetic except in very specialized code (such as a memory manager) and for simple array traversal (e.g., **++p**).

[10] Do not assume that something laboriously written in C style (avoiding C++ features such as classes, templates, and exceptions) is more efficient than a shorter alternative (e.g., using standard-library facilities). Often (but of course not always), the opposite is true.

14.3.2.2 void*

In C, a void* may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not. For example:

```
void f(int n)
{
    int* p = malloc(n*sizeof(int));  /* not C++; in C++, allocate using "new" */
    // ...
}
```

This is probably the single most difficult incompatibility to deal with. Note that the implicit conversion of a void* to a different pointer type is *not* in general harmless:

```
char ch;
void* pv = &ch;
int* pi = pv;        // not C++
*pi = 666;           // overwrite ch and other bytes near ch
```

If you use both languages, cast the result of malloc() to the right type. If you use only C++, avoid malloc().

14.3.2.3 C++ Keywords

C++ provides many more keywords than C does. If one of these appears as an identifier in a C program, that program must be modified to make it a C++ program:

C++ Keywords That Are Not C Keywords					
alignas	alignof	and	and_eq	asm	bitand
bitor	bool	catch	char16_t	char32_t	class
compl	const_cast	constexpr	decltype	delete	dynamic_cast
explicit	false	friend	inline	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or_eq	private	protected	public	reinterpret_cast	static_assert
static_cast	template	this	thread_local	throw	true
try	typeid	typename	using	virtual	wchar_t
xor	xor_eq				

In addition, the word export is reserved for future use. C99 adopted inline.

In C, some of the C++ keywords are macros defined in standard headers:

C++ Keywords That Are C Macros								
and	and_eq	bitand	bitor	bool	compl	false	not	not_eq
or	or_eq	true	wchar_t	xor	xor_eq			

This implies that in C they can be tested using #ifdef, redefined, etc.

14.3.2.4 Linkage

C and C++ can (and often is) implemented to use different linkage conventions. The most basic reason for that is C++'s greater emphasis on type checking. A practical reason is that C++ supports overloading, so that there can be two global functions called **open()**. This has to be reflected in the way the linker works.

To give a C++ function C linkage (so that it can be called from a C program fragment) or to allow a C function to be called from a C++ program fragment, declare it **extern "C"**. For example:

```
extern "C" double sqrt(double);
```

Now **sqrt(double)** can be called from a C or a C++ code fragment. The definition of **sqrt(double)** can also be compiled as a C function or as a C++ function.

Only one function of a given name in a scope can have C linkage (because C doesn't allow function overloading). A linkage specification does not affect type checking, so the C++ rules for function calls and argument checking still apply to a function declared **extern"C"**.

14.4 Bibliography

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.

[C,1999] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-1999.

[C,2011] ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-2011.

[C++,1998] ISO/IEC JTC1/SC22/WG21 (editor: Andrew Koenig): *International Standard – The C++ Language*. ISO/IEC 14882:1998.

[C++,2004] ISO/IEC JTC1/SC22/WG21 (editor: Lois Goldtwaite): *Technical Report on C++ Performance*. ISO/IEC TR 18015:2004(E)

[C++Math,2010] *International Standard – Extensions to the C++ Library to Support Mathematical Special Functions*. ISO/IEC 29124:2010.

[C++,2011] ISO/IEC JTC1/SC22/WG21 (editor: Pete Pecker): *International Standard – The C++ Language*. ISO/IEC 14882:2011.

[Cox,2007] Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/~rsc/regexp/regexp1.html.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dechev,2010] D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.

[DosReis,2006] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Friedl,1997]: Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.

[Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.

[Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

[Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.

[Maddock,2009] John Maddock: *Boost.Regex*. www.boost.org. 2009.

[Orwell,1949] George Orwell: *1984*. Secker and Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: *ML for the Working Programmer*. Cambridge University Press. Cambridge. 1996. ISBN 0-521-56543-X.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler*. Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.

[Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Sigplan Notices. January 1982. The first public description of "C with Classes."

[Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.

[Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference.

[Stroustrup,1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference. May 1987.

[Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. 1988.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN 0-201-53992-6.

[Stroustrup,1993] B. Stroustrup: *A History of C++: 1979-1991*. Proc. ACM History of Programming Languages conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

[Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover (''Special'') Edition. 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

[Stroustrup,2009] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.

[Stroustrup,2010] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12-0041. January 2012.

[Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

[Stroustrup,2013] B. Stroustrup: *The C++ Programming Language (Fourth Edition)*. Addison-Wesley. 2013. ISBN 0-321-56384-0.

[Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.

[WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co. ISBN 978-1933988771.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office. London. 1974.

14.5 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 1 and 44 of [Stroustrup,2013].

[2] The ISO C++ standard [C++,2011] defines C++.

[3] When learning C++, don't focus on language features in isolation; §14.2.1.

[4] By now, many people have been using C++ for a decade or two. Many more are using C++ in a single environment and have learned to live with the restrictions imposed by early compilers and first-generation libraries. Often, what an experienced C++ programmer has failed to notice over the years is not the introduction of new features as such, but rather the changes in relationships between features that make fundamental new programming techniques feasible. In other words, what you didn't think of when first learning C++ or found impractical just might be a superior approach today. You find out only by reexamining the basics. Take the opportunity offered by the new C++11 facilities to modernize your design and

programming techniques:

[1] Use constructors to establish invariants (§3.4.2).
[2] Use constructor/destructor pairs to simplify resource management (RAII; §4.2.2).
[3] Avoid "naked" **new** and **delete** (§4.2.2).
[4] Use containers and algorithms rather than built-in arrays and ad hoc code (Chapter 9, Chapter 10).
[5] Prefer standard-library facilities to locally developed code (Chapter 6).
[6] Use exceptions, rather than error codes, to report errors that cannot be handled locally (§3.4).
[7] Use move semantics to avoid copying large objects (§4.6).
[8] Use **unique_ptr** to reference objects of polymorphic type (§11.2.1).
[9] Use **shared_ptr** to reference shared objects, that is, objects without a single owner that is responsible for their destruction (§11.2.1).
[10] Use templates to maintain static type safety (eliminate casts) and avoid unnecessary use of class hierarchies (Chapter 5).

[5] Before using a new feature in production code, try it out by writing small programs to test the standards conformance and performance of the implementations you plan to use.
[6] For learning C++, use the most up-to-date and complete implementation of Standard C++ that you can get access to.
[7] The common subset of C and C++ is not the best initial subset of C++ to learn; §14.3.2.1.
[8] Prefer named casts, such as **static_cast** over C-style casts; §14.2.3.
[9] When converting a C program to C++, first make sure that function declarations (prototypes) and standard headers are used consistently; §14.3.2.
[10] When converting a C program to C++, rename variables that are C++ keywords; §14.3.2.3.
[11] For portability and type safety, if you must use C, write in the common subset of C and C++; §14.3.2.1.
[12] When converting a C program to C++, cast the result of **malloc()** to the proper type or change all uses of **malloc()** to uses of **new**; §14.3.2.2.
[13] When converting from **malloc()** and **free()** to **new** and **delete**, consider using **vector**, **push_back()**, and **reserve()** instead of **realloc()**; §14.3.2.1.
[14] In C++, there are no implicit conversions from **int**s to enumerations; use explicit type conversion where necessary.
[15] Use **<string>** to get **std::string** (**<string.h>** holds the C-style string functions).
[16] For each standard C header **<X.h>** that places names in the global namespace, the header **<cX>** places the names in namespace **std**.
[17] Use **extern "C"** when declaring C functions; §14.3.2.4.
[18] Prefer **string** over C-style strings (direct manipulation of zero-terminated arrays of **char**).
[19] Prefer **iostream**s over **stdio**.
[20] Prefer containers (e.g., **vector**) over built-in arrays.

This page intentionally left blank

I

Index

Knowledge is of two kinds.
We know a subject ourselves,
or we know where
we can find information on it.
– Samuel Johnson

Token

!=, not-equal operator 6
", string literal 3
$, regex 79
%
    modulus operator 6
    remainder operator 6
%=, operator 7
&
    address-of operator 10
    reference to 10
&&, rvalue reference 51
(, regex 79
(), call operator 64
(? pattern 82
), regex 79
*
    contents-of operator 10
    multiply operator 6
    pointer to 9
    regex 79
*=, scaling operator 7
*? lazy 80
+
    plus operator 6
    regex 79
    string concatenation 75
++, increment operator 7
+=
    operator 7
    string append 76
+? lazy 80
-, minus operator 6
--, decrement operator 7
., regex 79
/, divide operator 6
// comment 2
/=, scaling operator 7
: public 40
<<, output operator 2
<=, less-than-or-equal operator 6
<, less-than operator 6
=
    0 39
    and == 6
    auto 7
    initializer 6
    string assignment 77
==
    = and 6
    equal operator 6
    string 76

>, greater-than operator 6
>=, greater-than-or-equal operator 6
>> template arguments 159
?, regex 79
?? lazy 80
[, regex 79
[]
    array 122
    array of 9
    string 76
\, backslash 3
], regex 79
^, regex 79
_1, placeholders 126
_2, placeholders 126
{, regex 79
{}
    grouping 2
    initializer 6
{}? lazy 80
|, regex 79
}, regex 79
~, destructor 37
0
    = 39
    nullptr NULL 12
0x hexadecimal literal 161

A

abs() 134
abstract
    class 40
    type 39
accumulate() 135
acquisition RAII, resource 118
adaptor, function 125
address-of operator & 10
adjacent_difference() 135
aims, C++11 158
algorithm 107
    container 108, 115
    numerical 135
    standard library 114
<algorithm> 73, 114
alias, using 67
alignas 159
alignof 159
allocation 37
almost container 121
alnum, regex 81
alpha, regex 81
[[:alpha:]] letter 81
ANSI C++ 157
append +=, string 76
argument
    passing, function 52
    type 61
    value 61
arithmetic
    conversions, usual 6
    operator 6
    vector 138
ARM 157
array
    array vs. 123
    of [] 9
array 122
    [] 122
    data() 122
    initialize 122
    size() 122
    vs. array 123
    vs. vector 122
<array> 73
asin() 134
assembler 155
assertion static_assert 30
assignment
    =, string 77
    copy 49, 52
    move 51–52
associative array – see map
async() launch 150
at() 98
atan() 134
atan2() 134
AT&T Bell Laboratories 157
auto = 7
auto_ptr, deprecated 161

B

back_inserter() 108
backslash \ 3
base and derived class 40
basic_string 77
BCPL 162
begin() 100, 108
beginner, book for 1
Bell Laboratories, AT&T 157
bibliography 166
binary search 114
bind() 126
    and overloading 126
binder 125
bit-field, bitset and 123
bitset 123
    and bit-field 123
    and enum 123
blank, regex 81
block

as function body, try 99
try 28
body, function 2
book for beginner 1
bool 5
break 13

C

C 155
and C++ compatibility 161
Classic 162
difference from 161
K&R 162
macro, difference from 165
programmer 168
void * assignment, difference from 165
with Classes 154
with Classes language features 155
with Classes standard library 156
C++
ANSI 157
compatibility, C and 161
core language 2
history 153
ISO 157
meaning 155
programmer 168
pronunciation 155
standard, ISO 2
standard library 2
standardization 157
timeline 154
C++03 157
C++0x, C++11 155, 158
C++11
aims 158
C++0x 155, 158
language features 158
library components 159
C++98 157
standard library 157
C11 161
C89 and C99 161
C99, C89 and 161
call operator () 64
callback 128
capacity() 97
capture list 65
carries_dependency 159
cast 39
deprecated C-style 161
named 161
catch
clause 28
every exception 99
catch(...) 99
ceil() 134
char 5
character sets, multiple 77
chrono 125
<chrono> 73, 125, 146
class 34
concrete 34
scope 8
template 59
class
abstract 40
base and derived 40
hierarchy 42
Classic C 162
C-library header 73
clock timing 146
<cmath> 73, 134
cntrl, regex 81
code complexity, function and 4
comment, // 2
communication, task 147
comparison operator 6
compatibility, C and C++ 161
compilation
model, template 68
separate 24
compiler 2
compile-time
computation 128
evaluation 9
complete encapsulation 52
complex 35, 135
<complex> 73, 134–135
complexity, function and code 4
components, C++11 library 159
computation, compile-time 128
concatenation +, string 75
concept 63
concrete
class 34
type 34
concurrency 141
condition, declaration in 47
condition_variable 146
notify_one() 147
wait() 146
<condition_variable> 146
const, immutability 8
constant expression 9
const_cast 161
constexpr
function 9
immutability 8
const_iterator 112
constructor

and destructor 155
copy 49, 52
default 35
delegating 159
explicit 53
inheriting 159
initializer-list 38
invariant and 29
move 51–52
container 36, 59, 95
algorithm 108, 115
almost 121
object in 98
overview 103
return 109
sort() 129
specialized 121
standard library 103
contents-of operator * 10
conversion
explicit type 39, 161
narrowing 7
conversions, usual arithmetic 6
copy 48
and hierarchy 55
assignment 49, 52
constructor 49, 52
cost of 50
memberwise 52
copy() 114
copyif() 114
core language, C++ 2
cos() 134
cosh() 134
cost of copy 50
count() 114
count_if() 113–114
cout, output 2
<cstdlib> 73
C-style
cast, deprecated 161
error handling 134
string 12
Currying 125

D

\d, regex 81
\D, regex 81
d, regex 81
data race 142
data(), array 122
D&E 154
deadlock 145
deallocation 37
declaration 5
function 3
in condition 47
interface 23
declarator operator 11
decltype 159
decrement operator -- 7
default
constructor 35
operations 52
=default 53
definition implementation 24
delegating constructor 159
=delete 55
delete
an operation 55
naked 38
operator 37
deprecated
auto_ptr 161
C-style cast 161
exception specification 161
feature 160
deque 103
derived class, base and 40
destructor 37, 52
~ 37
constructor and 155
virtual 44
dictionary – see map
difference
from C 161
from C macro 165
from C void * assignment 165
digit, [[:digit:]] 81
digit, regex 81
[[:digit:]] digit 81
dispatch, tag 129
distribution, random 136
divide operator / 6
domain error 134
double 5
duck typing 68
duration 125
duration_cast 125
dynamic store 37
dynamic_cast 47
is instance of 47
is kind of 47

E

EDOM 134
element requirements 98
encapsulation, complete 52
end() 100, 108
engine, random 136

enum, bitset and 123
equal operator == 6
equal_range() 114, 124
ERANGE 134
erase() 101
errno 134
error
  domain 134
  handling 27
  handling, C-style 134
  range 134
  run-time 27
essential operations 52
evaluation
  compile-time 9
  partial 125
example
  find_all() 109
  Hello, World! 2
  Rand_int 137
  Vec 98
exception 27
  and main() 99
  catch every 99
  specification, deprecated 161
explicit type conversion 39, 161
explicit constructor 53
exponential_distribution 136
export removed 161
expr() 134
expression
  constant 9
  lambda 65
extern template 159

F

fabs() 134
facilities, standard library 72
feature, deprecated 160
features
  C with Classes language 155
  C++11 language 158
file, header 25
final 159
find() 108, 114
find_all() example 109
find_if() 113–114
first, pair member 124
floor() 134
fmod() 134
for
  statement 10
  statement, range 10
forward_list 103
<forward_list> 73
free store 37
frexp() 134
<fstream> 73
__func__ 159
function 2
  adaptor 125
  and code complexity 4
  argument passing 52
  body 2
  body, try block as 99
  constexpr 9
  declaration 3
  implementation of virtual 42
  mathematical 134
  object 64
  overloading 4
  template 62
  type 128
  value return 52
function 127
  and nullptr 127
fundamental type 5
future
  and promise 147
  member get() 147
<future> 73, 147

G

garbage collection 54
generic programming 62
get<>() 125
get(), future member 147
graph, regex 81
greater-than operator > 6
greater-than-or-equal operator >= 6
greedy match 80, 83
grouping, {} 2

H

half-open sequence 114
handle 38
  resource 49, 119
hash table 102
header
  C-library 73
  file 25
  standard library 73
heap 37
Hello, World! example 2
hexadecimal literal, 0x 161
hierarchy
  class 42
  copy and 55

navigation 47
history, C++ 153
HOPL 154

I

if statement 12
immutability
const 8
constexpr 8
implementation
definition 24
inheritance 46
iterator 111
of virtual function 42
string 77
in-class member initialization 159
#include 25
increment operator ++ 7
inheritance 40
implementation 46
interface 46
multiple 156
inheriting constructor 159
initialization, in-class member 159
initialize 38
array 122
initializer
= 6
{} 6
initializer-list constructor 38
initializer_list 38
inline 35
namespace 159
inlining 35
inner_product() 135
insert() 101
int 5
output bits of 123
interface
declaration 23
inheritance 46
invariant 29
and constructor 29
I/O, iterator and 112
<ios> 73
<iostream> 2, 73
iota() 135
is
instance of, dynamic_cast 47
kind of, dynamic_cast 47
ISO
C++ 157
C++ standard 2
ISO-14882 157
istream_iterator 112
iterator 108
and I/O 112
implementation 111
iterator 100, 112
<iterator> 130
iterator_category 129
iterator_traits 128, 130
iterator_type 129

J

join(), thread 142

K

key and value 101
K&R C 162

L

\L, regex 81
\l, regex 81
lambda expression 65
language
and library 71
features, C with Classes 155
features, C++11 158
launch, async() 150
lazy
*? 80
+? 80
?? 80
{}? 80
match 80, 83
ldexp() 134
leak, resource 47, 54, 118
less-than operator < 6
less-than-or-equal operator <= 6
letter, [[:alpha:]] 81
library
algorithm, standard 114
C with Classes standard 156
C++98 standard 157
components, C++11 159
container, standard 103
facilities, standard 72
language and 71
non-standard 71
standard 71
lifetime, scope and 8
<limits> 128, 138
linker 2
list, capture 65
list 100, 103
literal

", string 3
0x hexadecimal 161
raw string 78
user-defined 159
local scope 8
lock() 145
and RAII 145
log() 134
log10() 134
long long 159
lower, regex 81

M

macro, difference from C 165
main() 2
exception and 99
make_pair() 124
make_shared() 120
make_tuple() 125
make_unique() 120
management, resource 54, 117
map 101, 103
<map> 73
mapped type, value 101
match
greedy 80, 83
lazy 80, 83
mathematical
function 134
functions, standard 134
<math.h> 134
meaning, C++ 155
member initialization, in-class 159
memberwise copy 52
mem_fn() 126
<memory> 73, 118, 120
merge() 114
minus operator - 6
model, template compilation 68
modf() 134
modularity 23
modulus operator % 6
move 51
assignment 51–52
constructor 51–52
move() 52, 114
multimap 103
multiple
character sets 77
inheritance 156
multiply operator * 6
multiset 103
mutex 144
<mutex> 144

N

\n 3
naked
delete 38
new 38
named cast 161
namespace scope 8
namespace 26
inline 159
placeholders 126
std 72
narrowing 161
conversion 7
navigation, hierarchy 47
new
naked 38
operator 37
noexcept 28
noexcept() 159
non-standard library 71
noreturn 159
normal_distribution 136
notation, regular expression 79
not-equal operator != 6
notify_one(), condition_variable 147
NULL 0, nullptr 12
nullptr 11
function and 127
NULL 0 12
number, random 136
<numeric> 135
numerical algorithm 135
numeric_limits 138

O

object 5
function 64
in container 98
object-oriented programming 42
operation, delete an 55
operations
default 52
essential 52
operator
%= 7
+= 7
&, address-of 10
(), call 64
*, contents-of 10
--, decrement 7
/, divide 6
==, equal 6
>, greater-than 6
>=, greater-than-or-equal 6

++, increment 7
<, less-than 6
<=, less-than-or-equal 6
-, minus 6
%, modulus 6
*, multiply 6
!=, not-equal 6
<<, output 2
+, plus 6
%, remainder 6
*=, scaling 7
/=, scaling 7
arithmetic 6
comparison 6
declarator 11
delete 37
new 37
overloaded 36
user-defined 36
optimization, short-string 77
ostream_iterator 112
out_of_range 98
output
bits of int 123
cout 2
operator << 2
overloaded operator 36
overloading
bind() and 126
function 4
override 40
override 45
overview, container 103
ownership 118

P

packaged_task thread 149
pair 124
member first 124
member second 124
parameterized type 59
partial evaluation 125
partial_sum() 135
passing data to task 143
pattern, (? 82
phone_book example 96
placeholders
_1 126
_2 126
namespace 126
plus operator + 6
pointer
smart 118
to * 9
polymorphic type 40
pow() 134
precondition 29
predicate 64, 113
type 130
print, regex 81
program 2
programmer
C++ 168
C 168
programming
generic 62
object-oriented 42
promise
future and 147
member set_exception() 147
member set_value() 147
pronunciation, C++ 155
punct, regex 81
pure virtual 39
purpose, template 62
push_back() 38, 97, 101
push_front() 101

R

R" 78
race, data 142
RAII
lock() and 145
resource acquisition 118
RAII 38
Rand_int example 137
random number 136
random
distribution 136
engine 136
<random> 73, 136
range
checking Vec 98
error 134
for statement 10
raw string literal 78
reference
&&, rvalue 51
rvalue 52
to & 10
regex
| 79
$ 79
(79
) 79
* 79
+ 79
. 79
? 79
^ 79

[79
] 79
} 79
{ 79
alnum 81
alpha 81
blank 81
cntrl 81
\d 81
d 81
\D 81
digit 81
graph 81
\l 81
\L 81
lower 81
print 81
punct 81
regular expression 78
repetition 80
\S 81
\s 81
s 81
space 81
\U 81
\u 81
upper 81
w 81
\w 81
\W 81
xdigit 81
<regex> 73, 78
regular expression 78
regex_iterator 83
regex_search 78
regular
expression notation 79
expression <regex> 78
expression regex 78
reinterpret_cast 161
remainder operator % 6
removed, export 161
repetition, regex 80
replace() 114
string 76
replace_if() 114
requirement, template 63
requirements, element 98
reserve() 97
resource
acquisition RAII 118
handle 49, 119
leak 47, 54, 118
management 54, 117
safety 54
rethrow 30
return
function value 52
type, suffix 159
return
container 109
type, void 3
returning results from task 144
run-time error 27
rvalue
reference 52
reference && 51

S

\s, regex 81
s, regex 81
\S, regex 81
safety, resource 54
scaling
operator *= 7
operator /= 7
scope
and lifetime 8
class 8
local 8
namespace 8
search, binary 114
second, pair member 124
separate compilation 24
sequence 108
half-open 114
set 103
<set> 73
set_exception(), promise member 147
set_value(), promise member 147
shared_ptr 118
sharing data task 144
short-string optimization 77
Simula 153
sin() 134
sinh() 134
size of type 5
size(), array 122
sizeof 5
sizeof() 128
size_t 67
smart pointer 118
sort() 107, 114
container 129
space, regex 81
specialized container 121
sqrt() 134
<sstream> 73
standard
ISO C++ 2
library 71

library algorithm 114
library, C++ 2
library, C with Classes 156
library, C++98 157
library container 103
library facilities 72
library header 73
library std 72
mathematical functions 134
standardization, C++ 157
statement
for 10
if 12
range for 10
switch 13
while 12
static_assert 138
assertion 30
static_cast 39, 161
std 2
namespace 72
standard library 72
<stdexcept> 73
STL 157
store
dynamic 37
free 37
string
C-style 12
literal " 3
literal, raw 78
Unicode 77
string 75
[] 76
== 76
append += 76
assignment = 77
concatenation + 75
implementation 77
replace() 76
substr() 76
<string> 73, 75
subclass, superclass and 40
substr(), string 76
suffix return type 159
superclass and subclass 40
switch statement 13

T

table, hash 102
tag dispatch 129
tanh() 134
task
and thread 142
communication 147
passing data to 143
returning results from 144
sharing data 144
TC++PL 154
template
arguments, >> 159
compilation model 68
variadic 66
template 59
class 59
extern 159
function 62
purpose 62
requirement 63
thread
join() 142
packaged_task 149
task and 142
<thread> 73, 142
thread_local 159
time 125
timeline, C++ 154
time_point 125
timing, clock 146
try
block 28
block as function body 99
tuple 125
type 5
abstract 39
argument 61
concrete 34
conversion, explicit 39, 161
function 128
fundamental 5
parameterized 59
polymorphic 40
predicate 130
size of 5
typename 59, 110
<type_traits> 130
typing, duck 68

U

\u, regex 81
\U, regex 81
Unicode string 77
uniform_int_distribution 136
uninitialized 7
unique_copy() 107, 114
unique_lock 144, 146
unique_ptr 47, 118
unordered_map 102–103
<unordered_map> 73
unordered_multimap 103

unordered_multiset 103
unordered_set 103
unsigned 5
upper, regex 81
user-defined
    literal 159
    operator 36
using alias 67
usual arithmetic conversions 6
<utility> 73, 124–125

V

valarray 138
<valarray> 138
value 5
    argument 61
    key and 101
    mapped type 101
    return, function 52
value_type 67
variable 5
variadic template 66
Vec
    example 98
    range checking 98
vector arithmetic 138
vector 96, 103
    array vs. 122
<vector> 73
vector<bool> 121
virtual 39
    destructor 44
    function, implementation of 42
    function table vtbl 42
    pure 39
void
    * 165
    * assignment, difference from C 165
    return type 3
vtbl, virtual function table 42

W

w, regex 81
\w, regex 81
\W, regex 81
wait(), condition_variable 146
WG21 154
while statement 12

X

X3J16 157
xdigit, regex 81

Addison Wesley

More Guides from the Inventor of C++

ISBN-13: 978-0-321-56384-2

The C++ Programming Language, Fourth Edition, delivers meticulous, richly explained, and integrated coverage of the entire language—its facilities, abstraction mechanisms, standard libraries, and key design techniques. Throughout, Stroustrup presents concise, "pure C++11" examples, which have been carefully crafted to clarify both usage and program design.

Available in soft cover, hard cover, and eBook formats.

ISBN-13: 978-0-321-54372-1

Programming: Principles and Practice Using C++ is a general introduction to programming, including object-oriented programming and generic programming, and a solid introduction to the C++ programming language. Stroustrup presents modern C++ programming techniques from the start, introducing the C++ standard library to simplify programming tasks.

informIT

For more information and sample content visit informit.com/stroustrup

The C++ Programming Language

SPECIAL EDITION

Bjarne Stroustrup

The Creator of C++

The C++ Programming Language

Third Edition

Bjarne Stroustrup

AT&T Labs
Murray Hill, New Jersey

Addison-Wesley
An Imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts · Harlow, England · Menlo Park, California
Berkeley, California · Don Mills, Ontario · Sydney
Bonn · Amsterdam · Tokyo · Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial capital letters or all capital letters

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:
Corporate & Professional Publishing Group
Addison-Wesley Publishing Company
One Jacob Way
Reading, Massachusetts 01867

Library of Congress Cataloging-in-Publication Data

Stroustrup, Bjarne
The C++ Programming Language / Bjarne Stroustrup. — 3rd. ed.
p. cm.
Includes index.
ISBN 0-201-88954-4
1. C++ (Computer Programming Language) I. Title
QA76.73.C153S77 1997 97-20239
005.13’3—dc21 CIP

Copyright © 1997 by AT&T

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

This book was typeset in Times and Courier by the author.

ISBN 0-201-88954-4
Printed on recycled paper
1 2 3 4 5 6 7 8 9—CRW—0100999897
First printing, June 1997

Contents

Contents iii

Preface v

Preface to Second Edition vii

Preface to First Edition ix

Introductory Material 1

1 Notes to the Reader 3
2 A Tour of C++ 21
3 A Tour of the Standard Library 45

Part I: Basic Facilities 67

4 Types and Declarations 69
5 Pointers, Arrays, and Structures 87
6 Expressions and Statements 107
7 Functions 143
8 Namespaces and Exceptions 165
9 Source Files and Programs 197

Part II: Abstraction Mechanisms **221**

10 Classes 223
11 Operator Overloading 261
12 Derived Classes 301
13 Templates 327
14 Exception Handling 355
15 Class Hierarchies 389

Part III: The Standard Library **427**

16 Library Organization and Containers 429
17 Standard Containers 461
18 Algorithms and Function Objects 507
19 Iterators and Allocators 549
20 Strings 579
21 Streams 605
22 Numerics 657

Part IV: Design Using C++ **689**

23 Development and Design 691
24 Design and Programming 723
25 Roles of Classes 765

Appendices **791**

A The C++ Grammar 793
B Compatibility 815
C Technicalities 827

Index **869**

Preface

Programming is understanding.
– Kristen Nygaard

I find using C++ more enjoyable than ever. C++'s support for design and programming has improved dramatically over the years, and lots of new helpful techniques have been developed for its use. However, C++ is not *just* fun. Ordinary practical programmers have achieved significant improvements in productivity, maintainability, flexibility, and quality in projects of just about any kind and scale. By now, C++ has fulfilled most of the hopes I originally had for it, and also succeeded at tasks I hadn't even dreamt of.

This book introduces standard C++† and the key programming and design techniques supported by C++. Standard C++ is a far more powerful and polished language than the version of C++ introduced by the first edition of this book. New language features such as namespaces, exceptions, templates, and run-time type identification allow many techniques to be applied more directly than was possible before, and the standard library allows the programmer to start from a much higher level than the bare language.

About a third of the information in the second edition of this book came from the first. This third edition is the result of a rewrite of even larger magnitude. It offers something to even the most experienced C++ programmer; at the same time, this book is easier for the novice to approach than its predecessors were. The explosion of C++ use and the massive amount of experience accumulated as a result makes this possible.

The definition of an extensive standard library makes a difference to the way C++ concepts can be presented. As before, this book presents C++ independently of any particular implementation, and as before, the tutorial chapters present language constructs and concepts in a ''bottom up'' order so that a construct is used only after it has been defined. However, it is much easier to use a well-designed library than it is to understand the details of its implementation. Therefore, the standard library can be used to provide realistic and interesting examples well before a reader can be assumed to understand its inner workings. The standard library itself is also a fertile source of programming examples and design techniques.

† ISO/IEC 14882, Standard for the C++ Programming Language.

This book presents every major C++ language feature and the standard library. It is organized around language and library facilities. However, features are presented in the context of their use. That is, the focus is on the language as the tool for design and programming rather than on the language in itself. This book demonstrates key techniques that make C++ effective and teaches the fundamental concepts necessary for mastery. Except where illustrating technicalities, examples are taken from the domain of systems software. A companion, *The Annotated C++ Language Standard*, presents the complete language definition together with annotations to make it more comprehensible.

The primary aim of this book is to help the reader understand how the facilities offered by C++ support key programming techniques. The aim is to take the reader far beyond the point where he or she gets code running primarily by copying examples and emulating programming styles from other languages. Only a good understanding of the ideas behind the language facilities leads to mastery. Supplemented by implementation documentation, the information provided is sufficient for completing significant real-world projects. The hope is that this book will help the reader gain new insights and become a better programmer and designer.

Acknowledgments

In addition to the people mentioned in the acknowledgement sections of the first and second editions, I would like to thank Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl, Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger, and Chris Van Wyk for commenting on draft chapters of this third edition. Without their help and suggestions, this book would have been harder to understand, contained more errors, been slightly less complete, and probably been a little bit shorter.

I would also like to thank the volunteers on the C++ standards committees who did an immense amount of constructive work to make C++ what it is today. It is slightly unfair to single out individuals, but it would be even more unfair not to mention anyone, so I'd like to especially mention Mike Ball, Dag Brück, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, Josée Lajoie, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov, and Mike Vilot, as people who each directly cooperated with me over some part of C++ and its standard library.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the Second Edition

The road goes ever on and on.
– Bilbo Baggins

As promised in the first edition of this book, C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a great range of application areas. The C++ user-community has grown a hundredfold during the six years since the first edition of this book; many lessons have been learned, and many techniques have been discovered and/or validated by experience. Some of these experiences are reflected here.

The primary aim of the language extensions made in the last six years has been to enhance C++ as a language for data abstraction and object-oriented programming in general and to enhance it as a tool for writing high-quality libraries of user-defined types in particular. A ''high-quality library,'' is a library that provides a concept to a user in the form of one or more classes that are convenient, safe, and efficient to use. In this context, *safe* means that a class provides a specific type-safe interface between the users of the library and its providers; *efficient* means that use of the class does not impose significant overheads in run-time or space on the user compared with hand-written C code.

This book presents the complete C++ language. Chapters 1 through 10 give a tutorial introduction; Chapters 11 through 13 provide a discussion of design and software development issues; and, finally, the complete C++ reference manual is included. Naturally, the features added and resolutions made since the original edition are integral parts of the presentation. They include refined overloading resolution, memory management facilities, and access control mechanisms, type-safe linkage, ***const*** and ***static*** member functions, abstract classes, multiple inheritance, templates, and exception handling.

C++ is a general-purpose programming language; its core application domain is systems programming in the broadest sense. In addition, C++ is successfully used in many application areas that are not covered by this label. Implementations of C++ exist from some of the most modest microcomputers to the largest supercomputers and for almost all operating systems. Consequently, this book describes the C++ language itself without trying to explain a particular implementation, programming environment, or library.

This book presents many examples of classes that, though useful, should be classified as ''toys.'' This style of exposition allows general principles and useful techniques to stand out more

clearly than they would in a fully elaborated program, where they would be buried in details. Most of the useful classes presented here, such as linked lists, arrays, character strings, matrices, graphics classes, associative arrays, etc., are available in ‘‘bulletproof’’ and/or ‘‘goldplated’’ versions from a wide variety of commercial and non-commercial sources. Many of these ‘‘industrial strength’’ classes and libraries are actually direct and indirect descendants of the toy versions found here.

This edition provides a greater emphasis on tutorial aspects than did the first edition of this book. However, the presentation is still aimed squarely at experienced programmers and endeavors not to insult their intelligence or experience. The discussion of design issues has been greatly expanded to reflect the demand for information beyond the description of language features and their immediate use. Technical detail and precision have also been increased. The reference manual, in particular, represents many years of work in this direction. The intent has been to provide a book with a depth sufficient to make more than one reading rewarding to most programmers. In other words, this book presents the C++ language, its fundamental principles, and the key techniques needed to apply it. Enjoy!

Acknowledgments

In addition to the people mentioned in the acknowledgements section in the preface to the first edition, I would like to thank Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot, and Peter Weinberger for commenting on draft chapters of this second edition. Many people influenced the development of C++ from 1985 to 1991. I can mention only a few: Andrew Koenig, Brian Kernighan, Doug McIlroy, and Jonathan Shopiro. Also thanks to the many participants of the ‘‘external reviews’’ of the reference manual drafts and to the people who suffered through the first year of X3J16.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the First Edition

Language shapes the way we think,
and determines what we can think about.
– B.L.Whorf

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application. This technique for program construction is often called *data abstraction.* Objects of some user-defined types contain type information. Such objects can be used conveniently and safely in contexts in which their type cannot be determined at compile time. Programs using objects of such types are often called *object based*. When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.

The key concept in C++ is *class*. A class is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides much better facilities for type checking and for expressing modularity than C does. It also contains improvements that are not directly related to classes, including symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency.

C++ and its standard libraries are designed for portability. The current implementation will run on most systems that support C. C libraries can be used from a C++ program, and most tools that support programming in C can be used with C++.

This book is primarily intended to help serious programmers learn the language and use it for nontrivial projects. It provides a complete description of C++, many complete examples, and many more program fragments.

Acknowledgments

C++ could never have matured without the constant use, suggestions, and constructive criticism of many friends and colleagues. In particular, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, and Jon Shopiro provided important ideas for development of the language. Dave Presotto wrote the current implementation of the stream I/O library.

In addition, hundreds of people contributed to the development of C++ and its compiler by sending me suggestions for improvements, descriptions of problems they had encountered, and compiler errors. I can mention only a few: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, and Gary Walker.

Many people have also helped with the production of this book, in particular, Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro, and the participants in the C++ course held at Bell Labs, Columbus, Ohio, June 26-27, 1985.

Murray Hill, New Jersey *Bjarne Stroustrup*

Introduction

This introduction gives an overview of the major concepts and features of the C++ programming language and its standard library. It also provides an overview of this book and explains the approach taken to the description of the language facilities and their use. In addition, the introductory chapters present some background information about C++, the design of C++, and the use of C++.

Chapters

1 Notes to the Reader
2 A Tour of C++
3 A Tour of the Standard Library

‘‘... and you, Marcus, you have given me many things; now I shall give you this good advice. Be many people. Give up the game of being always Marcus Cocoza. You have worried too much about Marcus Cocoza, so that you have been really his slave and prisoner. You have not done anything without first considering how it would affect Marcus Cocoza’s happiness and prestige. You were always much afraid that Marcus might do a stupid thing, or be bored. What would it really have mattered? All over the world people are doing stupid things ... I should like you to be easy, your little heart to be light again. You must from now, be more than one, many people, as many as you can think of ...’’

– Karen Blixen
(‘‘The Dreamers’’ from ‘‘Seven Gothic Tales’’
written under the pseudonym Isak Dinesen,
Random House, Inc.
Copyright, Isac Dinesen, 1934 renewed 1961)

1

Notes to the Reader

"The time has come," the Walrus said,
"to talk of many things."
– L.Carroll

Structure of this book — how to learn C++ — the design of C++ — efficiency and structure — philosophical note — historical note — what C++ is used for — C and C++ — suggestions for C programmers — suggestions for C++ programmers — thoughts about programming in C++ — advice — references.

1.1 The Structure of This Book

This book consists of six parts:

Introduction: Chapters 1 through 3 give an overview of the C++ language, the key programming styles it supports, and the C++ standard library.

Part I: Chapters 4 through 9 provide a tutorial introduction to C++'s built-in types and the basic facilities for constructing programs out of them.

Part II: Chapters 10 through 15 are a tutorial introduction to object-oriented and generic programming using C++.

Part III: Chapters 16 through 22 present the C++ standard library.

Part IV: Chapters 23 through 25 discuss design and software development issues.

Appendices: Appendices A through E provide language-technical details.

Chapter 1 provides an overview of this book, some hints about how to use it, and some background information about C++ and its use. You are encouraged to skim through it, read what appears interesting, and return to it after reading other parts of the book.

Chapters 2 and 3 provide an overview of the major concepts and features of the C++ programming language and its standard library. Their purpose is to motivate you to spend time on fundamental concepts and basic language features by showing what can be expressed using the complete

C++ language. If nothing else, these chapters should convince you that C++ isn't (just) C and that C++ has come a long way since the first and second editions of this book. Chapter 2 gives a high-level acquaintance with C++. The discussion focuses on the language features supporting data abstraction, object-oriented programming, and generic programming. Chapter 3 introduces the basic principles and major facilities of the standard library. This allows me to use standard library facilities in the following chapters. It also allows you to use library facilities in exercises rather than relying directly on lower-level, built-in features.

The introductory chapters provide an example of a general technique that is applied throughout this book: to enable a more direct and realistic discussion of some technique or feature, I occasionally present a concept briefly at first and then discuss it in depth later. This approach allows me to present concrete examples before a more general treatment of a topic. Thus, the organization of this book reflects the observation that we usually learn best by progressing from the concrete to the abstract – even where the abstract seems simple and obvious in retrospect.

Part I describes the subset of C++ that supports the styles of programming traditionally done in C or Pascal. It covers fundamental types, expressions, and control structures for C++ programs. Modularity – as supported by namespaces, source files, and exception handling – is also discussed. I assume that you are familiar with the fundamental programming concepts used in Part I. For example, I explain C++'s facilities for expressing recursion and iteration, but I do not spend much time explaining how these concepts are useful.

Part II describes C++'s facilities for defining and using new types. Concrete and abstract classes (interfaces) are presented here (Chapter 10, Chapter 12), together with operator overloading (Chapter 11), polymorphism, and the use of class hierarchies (Chapter 12, Chapter 15). Chapter 13 presents templates, that is, C++'s facilities for defining families of types and functions. It demonstrates the basic techniques used to provide containers, such as lists, and to support generic programming. Chapter 14 presents exception handling, discusses techniques for error handling, and presents strategies for fault tolerance. I assume that you either aren't well acquainted with object-oriented programming and generic programming or could benefit from an explanation of how the main abstraction techniques are supported by C++. Thus, I don't just present the language features supporting the abstraction techniques; I also explain the techniques themselves. Part IV goes further in this direction.

Part III presents the C++ standard library. The aim is to provide an understanding of how to use the library, to demonstrate general design and programming techniques, and to show how to extend the library. The library provides containers (such as ***list***, ***vector***, and ***map***; Chapter 16, Chapter 17), standard algorithms (such as ***sort***, ***find***, and ***merge***; Chapter 18, Chapter 19), strings (Chapter 20), Input/Output (Chapter 21), and support for numerical computation (Chapter 22).

Part IV discusses issues that arise when C++ is used in the design and implementation of large software systems. Chapter 23 concentrates on design and management issues. Chapter 24 discusses the relation between the C++ programming language and design issues. Chapter 25 presents some ways of using classes in design.

Appendix A is C++'s grammar, with a few annotations. Appendix B discusses the relation between C and C++ and between Standard C++ (also called ISO C++ and ANSI C++) and the versions of C++ that preceded it. Appendix C presents some language-technical examples. Appendix D explains the standard library's facilities supporting internationalization. Appendix E discusses the exception-safety guarantees and requirements of the standard library.

1.1.1 Examples and References

This book emphasizes program organization rather than the writing of algorithms. Consequently, I avoid clever or harder-to-understand algorithms. A trivial algorithm is typically better suited to illustrate an aspect of the language definition or a point about program structure. For example, I use a Shell sort where, in real code, a quicksort would be better. Often, reimplementation with a more suitable algorithm is an exercise. In real code, a call of a library function is typically more appropriate than the code used here for illustration of language features.

Textbook examples necessarily give a warped view of software development. By clarifying and simplifying the examples, the complexities that arise from scale disappear. I see no substitute for writing realistically-sized programs for getting an impression of what programming and a programming language are really like. This book concentrates on the language features, the basic techniques from which every program is composed, and the rules for composition.

The selection of examples reflects my background in compilers, foundation libraries, and simulations. Examples are simplified versions of what is found in real code. The simplification is necessary to keep programming language and design points from getting lost in details. There are no ‘‘cute’’ examples without counterparts in real code. Wherever possible, I relegated to Appendix C language-technical examples of the sort that use variables named ***x*** and ***y***, types called ***A*** and ***B***, and functions called ***f***( ) and ***g***( ).

In code examples, a proportional-width font is used for identifiers. For example:

```
#include<iostream>

int main()
{
    std::cout << "Hello, new world!\n";
}
```

At first glance, this presentation style will seem ‘‘unnatural’’ to programmers accustomed to seeing code in constant-width fonts. However, proportional-width fonts are generally regarded as better than constant-width fonts for presentation of text. Using a proportional-width font also allows me to present code with fewer illogical line breaks. Furthermore, my experiments show that most people find the new style more readable after a short while.

Where possible, the C++ language and library features are presented in the context of their use rather than in the dry manner of a manual. The language features presented and the detail in which they are described reflect my view of what is needed for effective use of C++. A companion, *The Annotated C++ Language Standard*, authored by Andrew Koenig and myself, is the complete definition of the language together with comments aimed at making it more accessible. Logically, there ought to be another companion, *The Annotated C++ Standard Library*. However, since both time and my capacity for writing are limited, I cannot promise to produce that.

References to parts of this book are of the form §2.3.4 (Chapter 2, section 3, subsection 4), §B.5.6 (Appendix B, subsection 5.6), and §6.6[10] (Chapter 6, exercise 10). Italics are used sparingly for emphasis (e.g., ‘‘a string literal is *not* acceptable’’), for first occurrences of important concepts (e.g., *polymorphism*), for nonterminals of the C++ grammar (e.g., *for-statement*), and for comments in code examples. Semi-bold italics are used to refer to identifiers, keywords, and numeric values from code examples (e.g., ***counter***, ***class***, and ***1712***).

1.1.2 Exercises

Exercises are found at the ends of chapters. The exercises are mainly of the write-a-program variety. Always write enough code for a solution to be compiled and run with at least a few test cases. The exercises vary considerably in difficulty, so they are marked with an estimate of their difficulty. The scale is exponential so that if a (∗1) exercise takes you ten minutes, a (∗2) might take an hour, and a (∗3) might take a day. The time needed to write and test a program depends more on your experience than on the exercise itself. A (∗1) exercise might take a day if you first have to get acquainted with a new computer system in order to run it. On the other hand, a (∗5) exercise might be done in an hour by someone who happens to have the right collection of programs handy.

Any book on programming in C can be used as a source of extra exercises for Part I. Any book on data structures and algorithms can be used as a source of exercises for Parts II and III.

1.1.3 Implementation Note

The language used in this book is ‘‘pure C++’’ as defined in the C++ standard [C++,1998]. Therefore, the examples ought to run on every C++ implementation. The major program fragments in this book were tried using several C++ implementations. Examples using features only recently adopted into C++ didn’t compile on every implementation. However, I see no point in mentioning which implementations failed to compile which examples. Such information would soon be out of date because implementers are working hard to ensure that their implementations correctly accept every C++ feature. See Appendix B for suggestions on how to cope with older C++ compilers and with code written for C compilers.

1.2 Learning C++

The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than an understanding of details; that understanding comes with time and practice.

C++ supports a variety of programming styles. All are based on strong static type checking, and most aim at achieving a high level of abstraction and a direct representation of the programmer’s ideas. Each style can achieve its aims effectively while maintaining run-time and space efficiency. A programmer coming from a different language (say C, Fortran, Smalltalk, Lisp, ML, Ada, Eiffel, Pascal, or Modula-2) should realize that to gain the benefits of C++, they must spend time learning and internalizing programming styles and techniques suitable to C++. The same applies to programmers used to an earlier and less expressive version of C++.

Thoughtlessly applying techniques effective in one language to another typically leads to awkward, poorly performing, and hard-to-maintain code. Such code is also most frustrating to write because every line of code and every compiler error message reminds the programmer that the language used differs from ‘‘the old language.’’ You can write in the style of Fortran, C, Smalltalk, etc., in any language, but doing so is neither pleasant nor economical in a language with a different philosophy. Every language can be a fertile source of ideas of how to write C++ programs.

However, ideas must be transformed into something that fits with the general structure and type system of C++ in order to be effective in the different context. Over the basic type system of a language, only Pyrrhic victories are possible.

C++ supports a gradual approach to learning. How you approach learning a new programming language depends on what you already know and what you aim to learn. There is no one approach that suits everyone. My assumption is that you are learning C++ to become a better programmer and designer. That is, I assume that your purpose in learning C++ is not simply to learn a new syntax for doing things the way you used to, but to learn new and better ways of building systems. This has to be done gradually because acquiring any significant new skill takes time and requires practice. Consider how long it would take to learn a new natural language well or to learn to play a new musical instrument well. Becoming a better system designer is easier and faster, but not as much easier and faster as most people would like it to be.

It follows that you will be using C++ – often for building real systems – before understanding every language feature and technique. By supporting several programming paradigms (Chapter 2), C++ supports productive programming at several levels of expertise. Each new style of programming adds another tool to your toolbox, but each is effective on its own and each adds to your effectiveness as a programmer. C++ is organized so that you can learn its concepts in a roughly linear order and gain practical benefits along the way. This is important because it allows you to gain benefits roughly in proportion to the effort expended.

In the continuing debate on whether one needs to learn C before C++, I am firmly convinced that it is best to go directly to C++. C++ is safer, more expressive, and reduces the need to focus on low-level techniques. It is easier for you to learn the trickier parts of C that are needed to compensate for its lack of higher-level facilities after you have been exposed to the common subset of C and C++ and to some of the higher-level techniques supported directly in C++. Appendix B is a guide for programmers going from C++ to C, say, to deal with legacy code.

Several independently developed and distributed implementations of C++ exist. A wealth of tools, libraries, and software development environments are also available. A mass of textbooks, manuals, journals, newsletters, electronic bulletin boards, mailing lists, conferences, and courses are available to inform you about the latest developments in C++, its use, tools, libraries, implementations, etc. If you plan to use C++ seriously, I strongly suggest that you gain access to such sources. Each has its own emphasis and bias, so use at least two. For example, see [Barton,1994], [Booch,1994], [Henricson,1997], [Koenig,1997], [Martin,1995].

1.3 The Design of C++

Simplicity was an important design criterion: where there was a choice between simplifying the language definition and simplifying the compiler, the former was chosen. However, great importance was attached to retaining a high degree of compatibility with C [Koenig,1989] [Stroustrup,1994] (Appendix B); this precluded cleaning up the C syntax.

C++ has no built-in high-level data types and no high-level primitive operations. For example, the C++ language does not provide a matrix type with an inversion operator or a string type with a concatenation operator. If a user wants such a type, it can be defined in the language itself. In fact, defining a new general-purpose or application-specific type is the most fundamental programming

activity in C++. A well-designed user-defined type differs from a built-in type only in the way it is defined, not in the way it is used. The C++ standard library described in Part III provides many examples of such types and their uses. From a user's point of view, there is little difference between a built-in type and a type provided by the standard library.

Features that would incur run-time or memory overheads even when not used were avoided in the design of C++. For example, constructs that would make it necessary to store ''housekeeping information'' in every object were rejected, so if a user declares a structure consisting of two 16-bit quantities, that structure will fit into a 32-bit register.

C++ was designed to be used in a traditional compilation and run-time environment, that is, the C programming environment on the UNIX system. Fortunately, C++ was never restricted to UNIX; it simply used UNIX and C as a model for the relationships between language, libraries, compilers, linkers, execution environments, etc. That minimal model helped C++ to be successful on essentially every computing platform. There are, however, good reasons for using C++ in environments that provide significantly more support. Facilities such as dynamic loading, incremental compilation, and a database of type definitions can be put to good use without affecting the language.

C++ type-checking and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. They do not provide secrecy or protection against someone who is deliberately breaking the rules. They can, however, be used freely without incurring run-time or space overheads. The idea is that to be useful, a language feature must not only be elegant; it must also be affordable in the context of a real program.

For a systematic and detailed description of the design of C++, see [Stroustrup,1994].

1.3.1 Efficiency and Structure

C++ was developed from the C programming language and, with few exceptions, retains C as a subset. The base language, the C subset of C++, is designed to ensure a very close correspondence between its types, operators, and statements and the objects that computers deal with directly: numbers, characters, and addresses. Except for the ***new***, ***delete***, ***typeid***, ***dynamic_cast***, and ***throw*** operators and the *try-block*, individual C++ expressions and statements need no run-time support.

C++ can use the same function call and return sequences as C – or more efficient ones. When even such relatively efficient mechanisms are too expensive, a C++ function can be substituted inline, so that we can enjoy the notational convenience of functions without run-time overhead.

One of the original aims for C was to replace assembly coding for the most demanding systems programming tasks. When C++ was designed, care was taken not to compromise the gains in this area. The difference between C and C++ is primarily in the degree of emphasis on types and structure. C is expressive and permissive. C++ is even more expressive. However, to gain that increase in expressiveness, you must pay more attention to the types of objects. Knowing the types of objects, the compiler can deal correctly with expressions when you would otherwise have had to specify operations in painful detail. Knowing the types of objects also enables the compiler to detect errors that would otherwise persist until testing – or even later. Note that using the type system to check function arguments, to protect data from accidental corruption, to provide new types, to provide new operators, etc., does not increase run-time or space overheads in C++.

The emphasis on structure in C++ reflects the increase in the scale of programs written since C was designed. You can make a small program (say, 1,000 lines) work through brute force even

when breaking every rule of good style. For a larger program, this is simply not so. If the structure of a 100,000-line program is bad, you will find that new errors are introduced as fast as old ones are removed. C++ was designed to enable larger programs to be structured in a rational way so that it would be reasonable for a single person to cope with far larger amounts of code. In addition, the aim was to have an average line of C++ code express much more than the average line of C or Pascal code. C++ has by now been shown to over-fulfill these goals.

Not every piece of code can be well-structured, hardware-independent, easy-to-read, etc. C++ possesses features that are intended for manipulating hardware facilities in a direct and efficient way without regard for safety or ease of comprehension. It also possesses facilities for hiding such code behind elegant and safe interfaces.

Naturally, the use of C++ for larger programs leads to the use of C++ by groups of programmers. C++'s emphasis on modularity, strongly typed interfaces, and flexibility pays off here. C++ has as good a balance of facilities for writing large programs as any language has. However, as programs get larger, the problems associated with their development and maintenance shift from being language problems to more global problems of tools and management. Part IV explores some of these issues.

This book emphasizes techniques for providing general-purpose facilities, generally useful types, libraries, etc. These techniques will serve programmers of small programs as well as programmers of large ones. Furthermore, because all nontrivial programs consist of many semi-independent parts, the techniques for writing such parts serve programmers of all applications.

You might suspect that specifying a program by using a more detailed type structure would lead to a larger program source text. With C++, this is not so. A C++ program declaring function argument types, using classes, etc., is typically a bit shorter than the equivalent C program not using these facilities. Where libraries are used, a C++ program will appear much shorter than its C equivalent, assuming, of course, that a functioning C equivalent could have been built.

1.3.2 Philosophical Note

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is ''close to the machine'' so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second purpose ideally requires a language that is ''close to the problem to be solved'' so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.

The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason, restricting language features with the intent of eliminating programmer errors is at best dangerous. As with natural languages, there are great benefits from being at least bilingual. A language provides a programmer with a set of conceptual tools; if these are inadequate for a task, they will simply be ignored. Good design and the absence of errors cannot be guaranteed merely by the presence or the absence of specific language features.

The type system should be especially helpful for nontrivial tasks. The C++ class concept has, in fact, proven itself to be a powerful conceptual tool.

1.4 Historical Note

I invented C++, wrote its early definitions, and produced its first implementation. I chose and formulated the design criteria for C++, designed all its major facilities, and was responsible for the processing of extension proposals in the C++ standards committee.

Clearly, C++ owes much to C [Kernighan,1978]. Except for closing a few serious loopholes in the type system (see Appendix B), C is retained as a subset. I also retained C's emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks. C in turn owes much to its predecessor BCPL [Richards,1980]; in fact, BCPL's `//` comment convention was (re)introduced in C++. The other main source of inspiration for C++ was Simula67 [Dahl,1970] [Dahl,1972]; the class concept (with derived classes and virtual functions) was borrowed from it. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68 [Woodward,1974].

Since the original edition of this book, the language has been extensively reviewed and refined. The major areas for revision were overload resolution, linking, and memory management facilities. In addition, several minor changes were made to increase C compatibility. Several generalizations and a few major extensions were added: these included multiple inheritance, ***static*** member functions, ***const*** member functions, ***protected*** members, templates, exception handling, run-time type identification, and namespaces. The overall theme of these extensions and revisions was to make C++ a better language for writing and using libraries. The evolution of C++ is described in [Stroustrup,1994].

The template facility was primarily designed to support statically typed containers (such as lists, vectors, and maps) and to support elegant and efficient use of such containers (generic programming). A key aim was to reduce the use of macros and casts (explicit type conversion). Templates were partly inspired by Ada's generics (both their strengths and their weaknesses) and partly by Clu's parameterized modules. Similarly, the C++ exception-handling mechanism was inspired partly by Ada [Ichbiah,1979], Clu [Liskov,1979], and ML [Wikström,1987]. Other developments in the 1985 to 1995 time span – such as multiple inheritance, pure virtual functions, and namespaces – were primarily generalizations driven by experience with the use of C++ rather than ideas imported from other languages.

Earlier versions of the language, collectively known as ''C with Classes'' [Stroustrup,1994], have been in use since 1980. The language was originally invented because I wanted to write some event-driven simulations for which Simula67 would have been ideal, except for efficiency considerations. ''C with Classes'' was used for major projects in which the facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions, templates, exceptions, and many details. The first use of C++ outside a research organization started in July 1983.

The name C++ (pronounced ''see plus plus'') was coined by Rick Mascitti in the summer of 1983. The name signifies the evolutionary nature of the changes from C; ''++'' is the C increment operator. The slightly shorter name ''C+'' is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is not called D, because it is an extension of C, and it does not attempt to remedy problems by removing features. For yet another interpretation of the name C++, see the appendix of [Orwell,1949].

C++ was designed primarily so that my friends and I would not have to program in assembler,

C, or various modern high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer. In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously. There was no ''C++ project'' either, or a ''C++ design committee.'' Throughout, C++ evolved to cope with problems encountered by users and as a result of discussions between my friends, my colleagues, and me.

Later, the explosive growth of C++ use caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground for a standardization effort [Stroustrup,1994]. The result was a conscious effort to maintain contact between implementers of C++ compilers and major users through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Laboratories made a major contribution to this by allowing me to share drafts of revised versions of the C++ reference manual with implementers and users. Because many of these people work for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from dozens of organizations read and commented on what became the generally accepted reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual* [Ellis,1989]. Finally, the X3J16 committee of ANSI was convened in December 1989 at the initiative of Hewlett-Packard. In June 1991, this ANSI (American national) standardization of C++ became part of an ISO (international) standardization effort for C++. From 1990, these joint C++ standards committees have been the main forum for the evolution of C++ and the refinement of its definition. I served on these committees throughout. In particular, as the chairman of the working group for extensions, I was directly responsible for the handling of proposals for major changes to C++ and the addition of new language features. An initial draft standard for public review was produced in April 1995. The ISO C++ standard (ISO/IEC 14882) was ratified in 1998.

C++ evolved hand-in-hand with some of the key classes presented in this book. For example, I designed complex, vector, and stack classes together with the operator overloading mechanisms. String and list classes were developed by Jonathan Shopiro and me as part of the same effort. Jonathan's string and list classes were the first to see extensive use as part of a library. The string class from the standard C++ library has its roots in these early efforts. The task library described in [Stroustrup,1987] and in §12.7[11] was part of the first ''C with Classes'' program ever written. I wrote it and its associated classes to support Simula-style simulations. The task library has been revised and reimplemented, notably by Jonathan Shopiro, and is still in extensive use. The stream library as described in the first edition of this book was designed and implemented by me. Jerry Schwarz transformed it into the iostreams library (Chapter 21) using Andrew Koenig's manipulator technique (§21.4.6) and other ideas. The iostreams library was further refined during standardization, when the bulk of the work was done by Jerry Schwarz, Nathan Myers, and Norihiro Kumagai. The development of the template facility was influenced by the ***vector***, ***map***, ***list***, and ***sort*** templates devised by Andrew Koenig, Alex Stepanov, me, and others. In turn, Alex Stepanov's work on generic programming using templates led to the containers and algorithms parts of the standard C++ library (§16.3, Chapter 17, Chapter 18, §19.2). The ***valarray*** library for numerical computation (Chapter 22) is primarily the work of Kent Budge.

1.5 Use of C++

C++ is used by hundreds of thousands of programmers in essentially every application domain. This use is supported by about a dozen independent implementations, hundreds of libraries, hundreds of textbooks, several technical journals, many conferences, and innumerable consultants. Training and education at a variety of levels are widely available.

Early applications tended to have a strong systems programming flavor. For example, several major operating systems have been written in C++ [Campbell,1987] [Rozier,1988] [Hamilton,1993] [Berg,1995] [Parrington,1995] and many more have key parts done in C++. I considered uncompromising low-level efficiency essential for C++. This allows us to use C++ to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed. Often, so is compactness of the resulting system. C++ was designed so that every language feature is usable in code under severe time and space constraints [Stroustrup,1994,§4.5].

Most applications have sections of code that are critical for acceptable performance. However, the largest amount of code is not in such sections. For most code, maintainability, ease of extension, and ease of testing is key. C++'s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over time. Examples are banking, trading, insurance, telecommunications, and military applications. For years, the central control of the U.S. long-distance telephone system has relied on C++ and every 800 call (that is, a call paid for by the called party) has been routed by a C++ program [Kamath,1993]. Many such applications are large and long-lived. As a result, stability, compatibility, and scalability have been constant concerns in the development of C++. Million-line C++ programs are not uncommon.

Like C, C++ wasn't specifically designed with numerical computation in mind. However, much numerical, scientific, and engineering computation is done in C++. A major reason for this is that traditional numerical work must often be combined with graphics and with computations relying on data structures that don't fit into the traditional Fortran mold [Budge,1992] [Barton,1994]. Graphics and user interfaces are areas in which C++ is heavily used. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are C++ programs. In addition, some of the most popular libraries supporting X for UNIX are written in C++. Thus, C++ is a common choice for the vast number of applications in which the user interface is a major part.

All of this points to what may be C++'s greatest strength: its ability to be used effectively for applications that require work in a variety of application areas. It is quite common to find an application that involves local and wide-area networking, numerics, graphics, user interaction, and database access. Traditionally, such application areas have been considered distinct, and they have most often been served by distinct technical communities using a variety of programming languages. However, C++ has been widely used in all of those areas. Furthermore, it is able to coexist with code fragments and programs written in other languages.

C++ is widely used for teaching and research. This has surprised some who – correctly – point out that C++ isn't the smallest or cleanest language ever designed. It is, however

- clean enough for successful teaching of basic concepts,
- realistic, efficient, and flexible enough for demanding projects,

– available enough for organizations and collaborations relying on diverse development and execution environments,
– comprehensive enough to be a vehicle for teaching advanced concepts and techniques, and
– commercial enough to be a vehicle for putting what is learned into non-academic use.

C++ is a language that you can grow with.

1.6 C and C++

C was chosen as the base language for C++ because it

[1] is versatile, terse, and relatively low-level;
[2] is adequate for most systems programming tasks;
[3] runs everywhere and on everything; and
[4] fits into the UNIX programming environment.

C has its problems, but a language designed from scratch would have some too, and we know C's problems. Importantly, working with C enabled ''C with Classes'' to be a useful (if awkward) tool within months of the first thought of adding Simula-like classes to C.

As C++ became more widely used, and as the facilities it provided over and above those of C became more significant, the question of whether to retain compatibility was raised again and again. Clearly some problems could be avoided if some of the C heritage was rejected (see, e.g., [Sethi,1981]). This was not done because

[1] there are millions of lines of C code that might benefit from C++, provided that a complete rewrite from C to C++ were unnecessary;
[2] there are millions of lines of library functions and utility software code written in C that could be used from/on C++ programs provided C++ were link-compatible with and syntactically very similar to C;
[3] there are hundreds of thousands of programmers who know C and therefore need only learn to use the new features of C++ and not relearn the basics; and
[4] C++ and C will be used on the same systems by the same people for years, so the differences should be either very large or very small so as to minimize mistakes and confusion.

The definition of C++ has been revised to ensure that a construct that is both legal C and legal C++ has the same meaning in both languages (with a few minor exceptions; see §B.2).

The C language has itself evolved, partly under the influence of the development of C++ [Rosler,1984]. The ANSI C standard [C,1990] contains a function declaration syntax borrowed from ''C with Classes.'' Borrowing works both ways. For example, the ***void**** pointer type was invented for ANSI C and first implemented in C++. As promised in the first edition of this book, the definition of C++ has been reviewed to remove gratuitous incompatibilities; C++ is now more compatible with C than it was originally. The ideal was for C++ to be as close to ANSI C as possible – but no closer [Koenig,1989]. One hundred percent compatibility was never a goal because that would compromise type safety and the smooth integration of user-defined and built-in types.

Knowing C is not a prerequisite for learning C++. Programming in C encourages many techniques and tricks that are rendered unnecessary by C++ language features. For example, explicit type conversion (casting) is less frequently needed in C++ than it is in C (§1.6.1). However, *good* C programs tend to be C++ programs. For example, every program in Kernighan and Ritchie, *The*

C Programming Language (2nd Edition) [Kernighan,1988], is a C++ program. Experience with any statically typed language will be a help when learning C++.

1.6.1 Suggestions for C Programmers

The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing some of the potential benefits of C++. Please take a look at Appendix B, which describes the differences between C and C++. Here are a few pointers to the areas in which C++ has better ways of doing something than C has:

[1] Macros are almost never necessary in C++. Use ***const*** (§5.4) or ***enum*** (§4.8) to define manifest constants, ***inline*** (§7.1.1) to avoid function-calling overhead, ***template***s (Chapter 13) to specify families of functions and types, and ***namespace***s (§8.2) to avoid name clashes.

[2] Don't declare a variable before you need it so that you can initialize it immediately. A declaration can occur anywhere a statement can (§6.3.1), in *for-statement* initializers (§6.3.3), and in conditions (§6.3.2.1).

[3] Don't use ***malloc***( ). The ***new*** operator (§6.2.6) does the same job better, and instead of ***realloc***( ), try a ***vector*** (§3.8).

[4] Try to avoid ***void****, pointer arithmetic, unions, and casts, except deep within the implementation of some function or class. In most cases, a cast is an indication of a design error. If you must use an explicit type conversion, try using one of the ''new casts'' (§6.2.7) for a more precise statement of what you are trying to do.

[5] Minimize the use of arrays and C-style strings. The C++ standard library ***string*** (§3.5) and ***vector*** (§3.7.1) classes can often be used to simplify programming compared to traditional C style. In general, try not to build yourself what has already been provided by the standard library.

To obey C linkage conventions, a C++ function must be declared to have C linkage (§9.2.4).

Most important, try thinking of a program as a set of interacting concepts represented as classes and objects, instead of as a bunch of data structures with functions twiddling their bits.

1.6.2 Suggestions for C++ Programmers

By now, many people have been using C++ for a decade. Many more are using C++ in a single environment and have learned to live with the restrictions imposed by early compilers and first-generation libraries. Often, what an experienced C++ programmer has failed to notice over the years is not the introduction of new features as such, but rather the changes in relationships between features that make fundamental new programming techniques feasible. In other words, what you didn't think of when first learning C++ or found impractical just might be a superior approach today. You find out only by re-examining the basics.

Read through the chapters in order. If you already know the contents of a chapter, you can be through in minutes. If you don't already know the contents, you'll have learned something unexpected. I learned a fair bit writing this book, and I suspect that hardly any C++ programmer knows every feature and technique presented. Furthermore, to use the language well, you need a perspective that brings order to the set of features and techniques. Through its organization and examples, this book offers such a perspective.

1.7 Thinking about Programming in C++

Ideally, you approach the task of designing a program in three stages. First, you gain a clear understanding of the problem (analysis), then you identify the key concepts involved in a solution (design), and finally you express that solution in a program (programming). However, the details of the problem and the concepts of the solution often become clearly understood only through the effort to express them in a program and trying to get it to run acceptably. This is where the choice of programming language matters.

In most applications, there are concepts that are not easily represented as one of the fundamental types or as a function without associated data. Given such a concept, declare a class to represent it in the program. A C++ class is a type. That is, it specifies how objects of its class behave: how they are created, how they can be manipulated, and how they are destroyed. A class may also specify how objects are represented, although in the early stages of the design of a program that should not be the major concern. The key to writing good programs is to design classes so that each cleanly represents a single concept. Often, this means that you must focus on questions such as: How are objects of this class created? Can objects of this class be copied and/or destroyed? What operations can be applied to such objects? If there are no good answers to such questions, the concept probably wasn't ''clean'' in the first place. It might then be a good idea to think more about the problem and its proposed solution instead of immediately starting to ''code around'' the problems.

The concepts that are easiest to deal with are the ones that have a traditional mathematical formalism: numbers of all sorts, sets, geometric shapes, etc. Text-oriented I/O, strings, basic containers, the fundamental algorithms on such containers, and some mathematical classes are part of the standard C++ library (Chapter 3, §16.1.2). In addition, a bewildering variety of libraries supporting general and domain-specific concepts are available.

A concept does not exist in a vacuum; there are always clusters of related concepts. Organizing the relationship between classes in a program – that is, determining the exact relationship between the different concepts involved in a solution – is often harder than laying out the individual classes in the first place. The result had better not be a muddle in which every class (concept) depends on every other. Consider two classes, A and B. Relationships such as ''A calls functions from B,'' ''A creates Bs,'' and ''A has a B member'' seldom cause major problems, while relationships such as ''A uses data from B'' can typically be eliminated.

One of the most powerful intellectual tools for managing complexity is hierarchical ordering, that is, organizing related concepts into a tree structure with the most general concept as the root. In C++, derived classes represent such structures. A program can often be organized as a set of trees or directed acyclic graphs of classes. That is, the programmer specifies a number of base classes, each with its own set of derived classes. Virtual functions (§2.5.5, §12.2.6) can often be used to define operations for the most general version of a concept (a base class). When necessary, the interpretation of these operations can be refined for particular special cases (derived classes).

Sometimes even a directed acyclic graph seems insufficient for organizing the concepts of a program; some concepts seem to be inherently mutually dependent. In that case, we try to localize cyclic dependencies so that they do not affect the overall structure of the program. If you cannot eliminate or localize such mutual dependencies, then you are most likely in a predicament that no programming language can help you out of. Unless you can conceive of some easily stated relationships between the basic concepts, the program is likely to become unmanageable.

One of the best tools for untangling dependency graphs is the clean separation of interface and implementation. Abstract classes (§2.5.4, §12.3) are C++'s primary tool for doing that.

Another form of commonality can be expressed through templates (§2.7, Chapter 13). A class template specifies a family of classes. For example, a list template specifies ''list of T,'' where ''T'' can be any type. Thus, a template is a mechanism for specifying how one type is generated given another type as an argument. The most common templates are container classes such as lists, vectors, and associative arrays (maps) and the fundamental algorithms using such containers. It is usually a mistake to express parameterization of a class and its associated functions with a type using inheritance. It is best done using templates.

Remember that much programming can be simply and clearly done using only primitive types, data structures, plain functions, and a few library classes. The whole apparatus involved in defining new types should not be used except when there is a real need.

The question ''How does one write good programs in C++?'' is very similar to the question ''How does one write good English prose?'' There are two answers: ''Know what you want to say'' and ''Practice. Imitate good writing.'' Both appear to be as appropriate for C++ as they are for English – and as hard to follow.

1.8 Advice

Here is a set of ''rules'' you might consider while learning C++. As you get more proficient you can evolve them into something suitable for your kind of applications and your style of programming. They are deliberately very simple, so they lack detail. Don't take them too literally. To write a good program takes intelligence, taste, and patience. You are not going to get it right the first time. Experiment!

[1] When you program, you create a concrete representation of the ideas in your solution to some problem. Let the structure of the program reflect those ideas as directly as possible:
  [a] If you can think of ''it'' as a separate idea, make it a class.
  [b] If you can think of ''it'' as a separate entity, make it an object of some class.
  [c] If two classes have a common interface, make that interface an abstract class.
  [d] If the implementations of two classes have something significant in common, make that commonality a base class.
  [e] If a class is a container of objects, make it a template.
  [f] If a function implements an algorithm for a container, make it a template function implementing the algorithm for a family of containers.
  [g] If a set of classes, templates, etc., are logically related, place them in a common namespace.

[2] When you define either a class that does not implement either a mathematical entity like a matrix or a complex number or a low-level type such as a linked list:
  [a] Don't use global data (use members).
  [b] Don't use global functions.
  [c] Don't use public data members.
  [d] Don't use friends, except to avoid [a] or [c].
  [e] Don't put a ''type field'' in a class; use virtual functions.
  [f] Don't use inline functions, except as a significant optimization.

More specific or detailed rules of thumb can be found in the ''Advice'' section of each chapter. Remember, this advice is only rough rules of thumb, not immutable laws. A piece of advice should be applied only ''where reasonable.'' There is no substitute for intelligence, experience, common sense, and good taste.

I find rules of the form ''never do this'' unhelpful. Consequently, most advice is phrased as suggestions of what to do, while negative suggestions tend not to be phrased as absolute prohibitions. I know of no major feature of C++ that I have not seen put to good use. The ''Advice'' sections do not contain explanations. Instead, each piece of advice is accompanied by a reference to the appropriate section of the book. Where negative advice is given, that section usually provides a suggested alternative.

1.8.1 References

There are few direct references in the text, but here is a short list of books and papers that are mentioned directly or indirectly.

[Barton,1994] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38 No. 10. October 1995.

[Booch,1994] Grady Booch: *Object-Oriented Analysis and Design*. Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.

[Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.

[C++,1998] X3 Secretariat: *International Standard – The C++ Language*. X3J16-14882. Information Technology Council (NSITC). Washington, DC, USA.

[Campbell,1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Coplien,1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-60734-4.

[Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.

[Dahl,1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction* in *Structured Programming*. Academic Press, New York. 1972.

[Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Gamma,1995] Erich Gamma, et al.: *Design Patterns*. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-63361-2.

[Goldberg,1983] A. Goldberg and D. Robson: *SMALLTALK-80 – The Language and Its Implementation*. Addison-Wesley. Reading, Mass. 1983.

[Griswold,1970] R. E. Griswold, et al.: *The Snobol4 Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1970.

[Griswold,1983] R. E. Griswold and M. T. Griswold: *The ICON Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1983.

[Hamilton,1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Objects.* Proc. 1993 Summer USENIX Conference. USENIX.

[Henricson,1997] Mats Henricson and Erik Nyquist: *Industrial Strength C++: Rules and Recommendations*. Prentice-Hall. Englewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.

[Ichbiah,1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14 No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72 No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice-Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (Second Edition).* Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol. 1 No. 7. July 1989.

[Koenig,1997] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Addison Wesley Longman. Reading, Mass. 1997. ISBN 0-201-42339-1.

[Knuth,1968] Donald Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Mass.

[Liskov,1979] Barbara Liskov et al.: *Clu Reference Manual*. MIT/LCS/TR-225. MIT Cambridge. Mass. 1979.

[Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.

[Orwell,1949] George Orwell: *1984.* Secker and Warburg. London. 1949.

[Parrington,1995] Graham Parrington et al.: *The Design and Implementation of Arjuna*. Computer Systems. Vol. 8 No. 3. Summer 1995.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler*. Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.

[Rosler,1984] L. Rosler: *The Evolution of C – Past and Future.* AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.

[Rozier,1988] M. Rozier, et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1 No. 4. Fall 1988.

[Sethi,1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations*. Software Practice & Experience. Vol. 11. 1981.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.

[Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

[Tarjan,1983] Robert E. Tarjan: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.

[Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley Developers Press. Reading, Mass. 1996. ISBN 0-201-48345-9.

[UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.

[Wikström,1987] Åke Wikström: *Functional Programming Using ML*. Prentice-Hall. Englewood Cliffs, New Jersey. 1987.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. England. 1974.

References to books relating to design and larger software development issues can be found at the end of Chapter 23.

2

A Tour of C++

The first thing we do, let´s
kill all the language lawyers.
– Henry VI, part II

What is C++? — programming paradigms — procedural programming — modularity — separate compilation — exception handling — data abstraction — user-defined types — concrete types — abstract types — virtual functions — object-oriented programming — generic programming — containers — algorithms — language and programming — advice.

2.1 What is C++? [tour.intro]

C++ is a general-purpose programming language with a bias towards systems programming that

- is a better C,
- supports data abstraction,
- supports object-oriented programming, and
- supports generic programming.

This chapter explains what this means without going into the finer details of the language definition. Its purpose is to give you a general overview of C++ and the key techniques for using it, *not* to provide you with the detailed information necessary to start programming in C++.

If you find some parts of this chapter rough going, just ignore those parts and plow on. All will be explained in detail in later chapters. However, if you do skip part of this chapter, do yourself a favor by returning to it later.

Detailed understanding of language features – even of *all* features of a language – cannot compensate for lack of an overall view of the language and the fundamental techniques for using it.

2.2 Programming Paradigms [tour.paradigm]

Object-oriented programming is a technique for programming – a paradigm for writing ''good'' programs for a set of problems. If the term ''object-oriented programming language'' means anything, it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran77 and object-oriented programs in C, but it is unnecessarily hard to do so because these languages do not directly support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time checks are also used to extend linguistic support for paradigms. Extra-linguistic facilities such as libraries and programming environments can provide further support for paradigms.

One language is not necessarily better than another because it possesses a feature the other does not. There are many examples to the contrary. The important issue is not so much what features a language possesses, but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

[1] All features must be cleanly and elegantly integrated into the language.

[2] It must be possible to use features in combination to achieve solutions that would otherwise require extra, separate features.

[3] There should be as few spurious and ''special-purpose'' features as possible.

[4] A feature's implementation should not impose significant overheads on programs that do not require it.

[5] A user should need to know only about the subset of the language explicitly used to write a program.

The first principle is an appeal to aesthetics and logic. The next two are expressions of the ideal of minimalism. The last two can be summarized as ''what you don't know won't hurt you.''

C++ was designed to support data abstraction, object-oriented programming, and generic programming in addition to traditional C programming techniques under these constraints. It was *not* meant to force one particular programming style upon all users.

The following sections consider some programming styles and the key language mechanisms supporting them. The presentation progresses through a series of techniques starting with procedural programming and leading up to the use of class hierarchies in object-oriented programming and generic programming using templates. Each paradigm builds on its predecessors, each adds something new to the C++ programmer's toolbox, and each reflects a proven design approach.

The presentation of language features is not exhaustive. The emphasis is on design approaches and ways of organizing programs rather than on language details. At this stage, it is far more important to gain an idea of what can be done using C++ than to understand exactly how it can be achieved.

2.3 Procedural Programming [tour.proc]

The original programming paradigm is:

> *Decide which procedures you want;*
> *use the best algorithms you can find.*

The focus is on the processing – the algorithm needed to perform the desired computation. Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways to pass arguments, ways to distinguish different kinds of arguments, different kinds of functions (e.g., procedures, routines, and macros), etc.

A typical example of ‘‘good style’’ is a square-root function. Given a double-precision floating-point argument, it produces a result. To do this, it performs a well-understood mathematical computation:

```
double sqrt(double arg)
{
        // code for calculating a square root
}

void f()
{
        double root2 = sqrt(2);
        // ...
}
```

Curly braces, `{ }`, express grouping in C++. Here, they indicate the start and end of the function bodies. The double slash, `//`, begins a comment that extends to the end of the line. The keyword ***void*** indicates that a function does not return a value.

From the point of view of program organization, functions are used to create order in a maze of algorithms. The algorithms themselves are written using function calls and other language facilities. The following subsections present a thumb-nail sketch of C++’s most basic facilities for expressing computation.

2.3.1 Variables and Arithmetic [tour.var]

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that ***inch*** is of type ***int***; that is, ***inch*** is an integer variable.

A *declaration* is a statement that introduces a name into the program. It specifies a type for that name. A *type* defines the proper use of a name or an expression.

C++ offers a variety of fundamental types, which correspond directly to hardware facilities. For example:

```
bool      // Boolean, possible values are true and false
char      // character, for example, 'a', 'z', and '9'
int       // integer, for example, 1, 42, and 1216
double    // double-precision floating-point number, for example, 3.14 and 299793.0
```

A ***char*** variable is of the natural size to hold a character on a given machine (typically a byte), and an ***int*** variable is of the natural size for integer arithmetic on a given machine (typically a word).

The arithmetic operators can be used for any combination of these types:

```
+         // plus, both unary and binary
-         // minus, both unary and binary
*         // multiply
/         // divide
%         // remainder
```

So can the comparison operators:

```
==        // equal
!=        // not equal
<         // less than
>         // greater than
<=        // less than or equal
>=        // greater than or equal
```

In assignments and in arithmetic operations, C++ performs all meaningful conversions between the basic types so that they can be mixed freely:

```
void some_function()      // function that doesn't return a value
{
    double d = 2.2;       // initialize floating-point number
    int i = 7;            // initialize integer
    d = d+i;              // assign sum to d
    i = d*i;              // assign product to i
}
```

As in C, = is the assignment operator and == tests equality.

2.3.2 Tests and Loops [tour.loop]

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

```
bool accept()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question

    char answer = 0;
    cin >> answer;                                   // read answer

    if (answer == 'y') return true;
    return false;
}
```

The << operator (''put to'') is used as an output operator; ***cout*** is the standard output stream. The >> operator (''get from'') is used as an input operator; ***cin*** is the standard input stream. The type of the right-hand operand of >> determines what input is accepted and is the target of the input operation. The ***\n*** character at the end of the output string represents a newline.

The example could be slightly improved by taking an 'n' answer into account:

```
bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n";    // write question

    char answer = 0;
    cin >> answer;                                   // read answer

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}
```

A *switch-statement* tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the ***default*** is chosen. The programmer need not provide a ***default***.

Few programs are written without loops. In this case, we might like to give the user a few tries:

```
bool accept3()
{
    int tries = 1;
    while (tries < 4) {
        cout << "Do you want to proceed (y or n)?\n";    // write question
        char answer = 0;
        cin >> answer;                                   // read answer

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "Sorry, I don't understand that.\n";
            tries = tries + 1;
        }
    }
    cout << "I'll take that for a no.\n";
    return false;
}
```

The *while-statement* executes until its condition becomes ***false***.

2.3.3 Pointers and Arrays [tour.ptr]

An array can be declared like this:

```
char v[10];    // array of 10 characters
```

Similarly, a pointer can be declared like this:

```
char* p;   // pointer to character
```

In declarations, `[]` means ''array of'' and `*` means ''pointer to.'' All arrays have *0* as their lower bound, so *v* has ten elements, `v[0]`...`v[9]`. A pointer variable can hold the address of an object of the appropriate type:

```
p = &v[3];      // p points to v's fourth element
```

Unary & is the address-of operator.

Consider copying ten elements from one array to another:

```
void another_function()
{
        int v1[10];
        int v2[10];
        // ...
        for (int i=0; i<10; ++i) v1[i]=v2[i];
}
```

This *for-statement* can be read as ''set *i* to zero, while *i* is less than *10*, copy the *i*th element and increment *i*.'' When applied to an integer variable, the increment operator ++ simply adds *1*.

2.4 Modular Programming [tour.module]

Over the years, the emphasis in the design of programs has shifted from the design of procedures and toward the organization of data. Among other things, this reflects an increase in program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

> *Decide which modules you want;*
> *partition the program so that data is hidden within modules.*

This paradigm is also known as the *data-hiding principle*. Where there is no grouping of procedures with related data, the procedural programming style suffices. Also, the techniques for designing ''good procedures'' are now applied for each procedure in a module. The most common example of a module is the definition of a stack. The main problems that have to be solved are:

[1] Provide a user interface for the stack (e.g., functions *push*( ) and *pop*( )).

[2] Ensure that the representation of the stack (e.g., an array of elements) can be accessed only through this user interface.

[3] Ensure that the stack is initialized before its first use.

C++ provides a mechanism for grouping related data, functions, etc., into separate namespaces. For example, the user interface of a ***Stack*** module could be declared and used like this:

```
namespace Stack {          // interface
    void push(char);
    char pop();
}

void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("impossible");
}
```

The ***Stack*****::** qualification indicates that the ***push*****()** and ***pop*****()** are those from the ***Stack*** namespace. Other uses of those names will not interfere or cause confusion.

The definition of the ***Stack*** could be provided in a separately-compiled part of the program:

```
namespace Stack {          // implementation
    const int max_size = 200;
    char v[max_size];
    int top = 0;

    void push(char c) { /* check for overflow and push c */ }
    char pop() { /* check for underflow and pop */ }
}
```

The key point about this ***Stack*** module is that the user code is insulated from the data representation of ***Stack*** by the code implementing ***Stack*****::*****push*****()** and ***Stack*****::*****pop*****()**. The user doesn't need to know that the ***Stack*** is implemented using an array, and the implementation can be changed without affecting user code.

Because data is only one of the things one might want to ''hide,'' the notion of data hiding is trivially extended to the notion of *information hiding*; that is, the names of functions, types, etc., can also be made local to a module. Consequently, C++ allows any declaration to be placed in a namespace (§8.2).

This ***Stack*** module is one way of representing a stack. The following sections use a variety of stacks to illustrate different programming styles.

2.4.1 Separate Compilation [tour.comp]

C++ supports C's notion of separate compilation. This can be used to organize a program into a set of semi-independent fragments.

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. Thus,

```
namespace Stack {          // interface
    void push(char);
    char pop();
}
```

would be placed in a file ***stack*****.*****h***, and users will *include* that file, called a *header file*, like this:

```
#include "stack.h"              // get the interface

void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("impossible");
}
```

To help the compiler ensure consistency, the file providing the implementation of the ***Stack*** module will also include the interface:

```
#include "stack.h"              // get the interface

namespace Stack {               // representation
    const int max_size = 200;
    char v[max_size];
    int top = 0;
}

void Stack::push(char c) { /* check for overflow and push c */ }

char Stack::pop() { /* check for underflow and pop */ }
```

The user code goes in a third file, say ***user.c***. The code in ***user.c*** and ***stack.c*** shares the stack interface information presented in ***stack.h***, but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:

Separate compilation is an issue in all real programs. It is not simply a concern in programs that present facilities, such as a ***Stack***, as modules. Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation (Chapter 8, Chapter 9).

2.4.2 Exception Handling [tour.except]

When a program is designed as a set of modules, error handling must be considered in light of these modules. Which module is responsible for handling what errors? Often, the module that detects an error doesn't know what action to take. The recovery action depends on the module that invoked

the operation rather than on the module that found the error while trying to perform the operation. As programs grow, and especially when libraries are used extensively, standards for handling errors (or, more generally, ‘‘exceptional circumstances’’) become important.

Consider again the ***Stack*** example. What *ought* to be done when we try to ***push***( ) one too many characters? The writer of the ***Stack*** module doesn’t know what the user would like to be done in this case, and the user cannot consistently detect the problem (if the user could, the overflow wouldn’t happen in the first place). The solution is for the ***Stack*** implementer to detect the overflow and then tell the (unknown) user. The user can then take appropriate action. For example:

```
namespace Stack {          // interface
    void push(char);
    char pop();

    class Overflow { };  // type representing overflow exceptions
}
```

When detecting an overflow, ***Stack***::***push***( ) can invoke the exception-handling code; that is, ‘‘throw an ***Overflow*** exception:’’

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    // push c
}
```

The ***throw*** transfers control to a handler for exceptions of type ***Stack***::***Overflow*** in some function that directly or indirectly called ***Stack***::***push***( ). To do that, the implementation will unwind the function call stack as needed to get back to the context of that caller. Thus, the ***throw*** acts as a multilevel ***return***. For example:

```
void f()
{
    // ...
    try { // exceptions here are handled by the handler defined below

        while (true) Stack::push('c');
    }
    catch (Stack::Overflow) {
        // oops: stack overflow; take appropriate action
    }
    // ...
}
```

The ***while*** loop will try to loop forever. Therefore, the ***catch***-clause providing a handler for ***Stack***::***Overflow*** will be entered after some call of ***Stack***::***push***( ) causes a ***throw***.

Use of the exception-handling mechanisms can make error handling more regular and readable. See §8.3 and Chapter 14 for further discussion and details.

2.5 Data Abstraction [tour.da]

Modularity is a fundamental aspect of all successful large programs. It remains a focus of all design discussions throughout this book. However, modules in the form described previously are not sufficient to express complex systems cleanly. Here, I first present a way of using modules to provide a form of user-defined types and then show how to overcome some problems with that approach by defining user-defined types directly.

2.5.1 Modules Defining Types [tour.types]

Programming with modules leads to the centralization of all data of a type under the control of a type manager module. For example, if we wanted many stacks – rather than the single one provided by the ***Stack*** module above – we could define a stack manager with an interface like this:

```
namespace Stack {
    struct Rep;                     // definition of stack layout is elsewhere
    typedef Rep& stack;

    stack create();                 // make a new stack
    void destroy(stack s);          // delete s

    void push(stack s, char c);     // push c onto s
    char pop(stack s);              // pop s
}
```

The declaration

```
struct Rep;
```

says that ***Rep*** is the name of a type, but it leaves the type to be defined later (§5.7). The declaration

```
typedef Rep& stack;
```

gives the name ***stack*** to a ‘‘reference to ***Rep***’’ (details in §5.5). The idea is that a stack is identified by its ***Stack*** : : ***stack*** and that further details are hidden from users.

A ***Stack*** : : ***stack*** acts much like a variable of a built-in type:

```
struct Bad_pop { };

void f()
{
    Stack::stack s1 = Stack::create();    // make a new stack
    Stack::stack s2 = Stack::create();    // make another new stack

    Stack::push(s1, ´c´);
    Stack::push(s2, ´k´);

    if (Stack::pop(s1) != ´c´) throw Bad_pop();
    if (Stack::pop(s2) != ´k´) throw Bad_pop();

    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

We could implement this ***Stack*** in several ways. It is important that a user doesn't need to know how we do it. As long as we keep the interface unchanged, a user will not be affected if we decide to re-implement ***Stack***.

An implementation might preallocate a few stack representations and let ***Stack*****::*****create*****()** hand out a reference to an unused one. ***Stack*****::*****destroy*****()** could then mark a representation ''unused'' so that ***Stack*****::*****create*****()** can recycle it:

```
namespace Stack {          // representation

    const int max_size = 200;

    struct Rep {
        char v[max_size];
        int top;
    };

    const int max = 16;  // maximum number of stacks

    Rep stacks[max];     // preallocated stack representations
    bool used[max];      // used[i] is true if stacks[i] is in use
}

void Stack::push(stack s, char c) { /* check s for overflow and push c */ }

char Stack::pop(stack s) { /* check s for underflow and pop */ }

Stack::stack Stack::create()
{
    // pick an unused Rep, mark it used, initialize it, and return a reference to it
}

void Stack::destroy(stack s) { /* mark s unused */ }
```

What we have done is to wrap a set of interface functions around the representation type. How the resulting ''stack type'' behaves depends partly on how we defined these interface functions, partly on how we presented the representation type to the users of ***Stack***s, and partly on the design of the representation type itself.

This is often less than ideal. A significant problem is that the presentation of such ''fake types'' to the users can vary greatly depending on the details of the representation type – and users ought to be insulated from knowledge of the representation type. For example, had we chosen to use a more elaborate data structure to identify a stack, the rules for assignment and initialization of ***Stack*****::*****stack***s would have changed dramatically. This may indeed be desirable at times. However, it shows that we have simply moved the problem of providing convenient ***Stack***s from the ***Stack*** module to the ***Stack*****::*****stack*** representation type.

More fundamentally, user-defined types implemented through a module providing access to an implementation type don't behave like built-in types and receive less and different support than do built-in types. For example, the time that a ***Stack*****::*****Rep*** can be used is controlled through ***Stack*****::*****create*****()** and ***Stack*****::*****destroy*****()** rather than by the usual language rules.

2.5.2 User-Defined Types [tour.udt]

C++ attacks this problem by allowing a user to directly define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*. I prefer the term *user-defined type*. A more reasonable definition of *abstract data type* would require a mathematical ''abstract'' specification. Given such a specification, what are called *types* here would be concrete examples of such truly abstract entities. The programming paradigm becomes:

> *Decide which types you want;*
> *provide a full set of operations for each type.*

Where there is no need for more than one object of a type, the data-hiding programming style using modules suffices.

Arithmetic types such as rational and complex numbers are common examples of user-defined types. Consider:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }   // construct complex from two scalars
    complex(double r) { re=r; im=0; }             // construct complex from one scalar
    complex( ) { re = im = 0; }                   // default complex: (0,0)

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);       // binary
    friend complex operator-(complex);                // unary
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);

    friend bool operator==(complex, complex);         // equal
    friend bool operator!=(complex, complex);         // not equal
    // ...
};
```

The declaration of class (that is, user-defined type) ***complex*** specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, ***re*** and ***im*** are accessible only to the functions specified in the declaration of class ***complex***. Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

A member function with the same name as its class is called a *constructor*. A constructor defines a way to initialize an object of its class. Class ***complex*** provides three constructors. One makes a ***complex*** from a ***double***, another takes a pair of ***double***s, and the third makes a ***complex*** with a default value.

Class ***complex*** can be used like this:

```
void f(complex z)
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b*complex(1,2.3);
    // ...
    if (c != b) c = -(b/a)+2*b;
}
```

The compiler converts operators involving *complex* numbers into appropriate function calls. For example, *c*!=*b* means *operator*!=(*c*,*b*) and *1/a* means *operator*/(*complex*(*1*),*a*).

Most, but not all, modules are better expressed as user-defined types.

2.5.3 Concrete Types [tour.concrete]

User-defined types can be designed to meet a wide variety of needs. Consider a user-defined *Stack* type along the lines of the *complex* type. To make the example a bit more realistic, this *Stack* type is defined to take its number of elements as an argument:

```
class Stack {
    char* v;
    int top;
    int max_size;
public:
    class Underflow { };        // used as exception
    class Overflow { };         // used as exception
    class Bad_size { };         // used as exception

    Stack(int s);               // constructor
    ~Stack();                   // destructor

    void push(char c);
    char pop();
};
```

The constructor *Stack*(*int*) will be called whenever an object of the class is created. This takes care of initialization. If any cleanup is needed when an object of the class goes out of scope, a complement to the constructor – called the *destructor* – can be declared:

```
Stack::Stack(int s)         // constructor
{
    top = 0;
    if (10000<s) throw Bad_size();
    max_size = s;
    v = new char[s];        // allocate elements on the free store (heap, dynamic store)
}

Stack::~Stack()             // destructor
{
    delete[] v;             // free the elements for possible reuse of their space (§6.2.6)
}
```

The constructor initializes a new ***Stack*** variable. To do so, it allocates some memory on the free store (also called the *heap* or *dynamic store*) using the ***new*** operator. The destructor cleans up by freeing that memory. This is all done without intervention by users of ***Stack***s. The users simply create and use ***Stack***s much as they would variables of built-in types. For example:

```
Stack s_var1(10);                    // global stack with 10 elements

void f(Stack& s_ref, int i)          // reference to Stack
{
    Stack s_var2(i);                 // local stack with i elements
    Stack* s_ptr = new Stack(20);    // pointer to Stack allocated on free store

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    // ...
}
```

This ***Stack*** type obeys the same rules for naming, scope, allocation, lifetime, copying, etc., as does a built-in type such as ***int*** and ***char***.

Naturally, the ***push***( ) and ***pop***( ) member functions must also be defined somewhere:

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    v[top] = c;
    top = top + 1;
}

char Stack::pop()
{
    if (top == 0) throw Underflow();
    top = top - 1;
    return v[top];
}
```

Types such as ***complex*** and ***Stack*** are called *concrete types*, in contrast to *abstract types*, where the interface more completely insulates a user from implementation details.

2.5.4 Abstract Types [tour.abstract]

One property was lost in the transition from ***Stack*** as a ''fake type'' implemented by a module (§2.5.1) to a proper type (§2.5.3). The representation is not decoupled from the user interface; rather, it is a part of what would be included in a program fragment using ***Stack***s. The representation is private, and therefore accessible only through the member functions, but it is present. If it changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types. In particular, we cannot have genuine local variables of a type without knowing the size of the type's representation.

For types that don't change often, and where local variables provide much-needed clarity and efficiency, this is acceptable and often ideal. However, if we want to completely isolate users of a

stack from changes to its implementation, this last ***Stack*** is insufficient. Then, the solution is to decouple the interface from the representation and give up genuine local variables.

First, we define the interface:

```
class Stack {
public:
        class Underflow { };        // used as exception
        class Overflow { };         // used as exception

        virtual void push(char c) = 0;
        virtual char pop() = 0;
};
```

The word ***virtual*** means ‘‘may be redefined later in a class derived from this one’’ in Simula and C++. A class derived from ***Stack*** provides an implementation for the ***Stack*** interface. The curious ***=0*** syntax says that some class derived from ***Stack*** *must* define the function. Thus, this ***Stack*** can serve as the interface to any class that implements its ***push*( )** and ***pop*( )** functions.

This ***Stack*** could be used like this:

```
void f(Stack& s_ref)
{
        s_ref.push(´c´);
        if (s_ref.pop() != ´c´) throw bad_stack();
}
```

Note how ***f*( )** uses the ***Stack*** interface in complete ignorance of implementation details. A class that provides the interface to a variety of other classes is often called a *polymorphic type*.

Not surprisingly, the implementation could consist of everything from the concrete class ***Stack*** that we left out of the interface ***Stack***:

```
class Array_stack : public Stack {      // Array_stack implements Stack
        char* p;
        int max_size;
        int top;
public:
        Array_stack(int s);
        ~Array_stack();

        void push(char c);
        char pop();
};
```

The ‘‘**:*public***’’ can be read as ‘‘is derived from,’’ ‘‘implements,’’ and ‘‘is a subtype of.’’

For a function like ***f*( )** to use a ***Stack*** in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

```
void g()
{
        Array_stack as(200);
        f(as);
}
```

Since *f*( ) doesn't know about ***Array_stack***s but only knows the ***Stack*** interface, it will work just as well for a different implementation of a ***Stack***. For example:

```
class List_stack : public Stack {        // List_stack implements Stack
        list<char> lc;                   // (standard library) list of characters (§3.7.3)
public:
        List_stack( ) { }

        void push(char c) { lc.push_front(c); }
        char pop( );
};

char List_stack::pop( )
{
        char x = lc.front( );            // get first element
        lc.pop_front( );                 // remove first element
        return x;
}
```

Here, the representation is a list of characters. The ***lc.push_front(c)*** adds ***c*** as the first element of ***lc***, the call ***lc.pop_front( )*** removes the first element, and ***lc.front( )*** denotes ***lc***'s first element.

A function can create a ***List_stack*** and have *f*( ) use it:

```
void h( )
{
        List_stack ls;
        f(ls);
}
```

2.5.5 Virtual Functions [tour.virtual]

How is the call ***s_set.pop( )*** in *f*( ) resolved to the right function definition? When *f*( ) is called from *h*( ), ***List_stack::pop( )*** must be called. When *f*( ) is called from *g*( ), ***Array_stack::pop( )*** must be called. To achieve this resolution, a ***Stack*** object must contain information to indicate the function to be called at run-time. A common implementation technique is for the compiler to convert the name of a ***virtual*** function into an index into a table of pointers to functions. That table is usually called ''a virtual function table'' or simply, a ***vtbl***. Each class with virtual functions has its own ***vtbl*** identifying its virtual functions. This can be represented graphically like this:

The functions in the ***vtbl*** allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. All the caller needs to know is the location of the ***vtbl*** in a ***Stack*** and the index used for each virtual function. This virtual call mechanism can be made essentially as efficient as the ''normal function call'' mechanism. Its space overhead is one pointer in each object of a class with virtual functions plus one ***vtbl*** for each such class.

2.6 Object-Oriented Programming [tour.oop]

Data abstraction is fundamental to good design and will remain a focus of design throughout this book. However, user-defined types by themselves are not flexible enough to serve our needs. This section first demonstrates a problem with simple user-defined data types and then shows how to overcome that problem by using class hierarchies.

2.6.1 Problems with Concrete Types [tour.problems]

A concrete type, like a ''fake type'' defined through a module, defines a sort of black box. Once the black box has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This situation can be ideal, but it can also lead to severe inflexibility. Consider defining a type ***Shape*** for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that we have

```
class Point{ /* ... */ };
class Color{ /* ... */ };
```

The `/*` and `*/` specify the beginning and end, respectively, of a comment. This comment notation can be used for multi-line comments and comments that end before the end of a line.

We might define a shape like this:

```
enum Kind { circle, triangle, square };     // enumeration (§4.8)

class Shape {
        Kind k;             // type field
        Point center;
        Color col;
        // ...

public:
        void draw();
        void rotate(int);
        // ...
};
```

The ''type field'' ***k*** is necessary to allow operations such as ***draw*( )** and ***rotate*( )** to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag ***k***). The function ***draw*( )** might be defined like this:

```
void Shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
        break;
    }
}
```

This is a mess. Functions such as ***draw*()** must ‘‘know about’’ all the kinds of shapes there are. Therefore, the code for any such function grows each time a new shape is added to the system. If we define a new shape, every operation on a shape must be examined and (possibly) modified. We are not able to add a new shape to a system unless we have access to the source code for every operation. Because adding a new shape involves ‘‘touching’’ the code of every important operation on shapes, doing so requires great skill and potentially introduces bugs into the code that handles other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed-sized framework presented by the definition of the general type ***Shape***.

2.6.2 Class Hierarchies [tour.hierarchies]

The problem is that there is no distinction between the general properties of every shape (that is, a shape has a color, it can be drawn, etc.) and the properties of a specific kind of shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. Languages with constructs that allow this distinction to be expressed and used support object-oriented programming. Other languages don’t.

The inheritance mechanism (borrowed for C++ from Simula) provides a solution. First, we specify a class that defines the general properties of all shapes:

```
class Shape {
    Point center;
    Color col;
    // ...
public:
    Point where() { return center; }
    void move(Point to) { center = to; /* ... */ draw(); }

    virtual void draw() = 0;
    virtual void rotate(int angle) = 0;
    // ...
};
```

As in the abstract type ***Stack*** in §2.5.4, the functions for which the calling interface can be defined – but where the implementation cannot be defined yet – are ***virtual***. In particular, the functions ***draw***( ) and ***rotate***( ) can be defined only for specific shapes, so they are declared ***virtual***.

Given this definition, we can write general functions manipulating vectors of pointers to shapes:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements angle degrees
{
    for (int i = 0; i<v.size(); ++i) v[i]->rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions):

```
class Circle : public Shape {
    int radius;
public:
    void draw() { /* ... */ }
    void rotate(int) {}   // yes, the null function
};
```

In C++, class ***Circle*** is said to be *derived* from class ***Shape***, and class ***Shape*** is said to be a *base* of class ***Circle***. An alternative terminology calls ***Circle*** and ***Shape*** subclass and superclass, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as *inheritance*.

The programming paradigm is:

> *Decide which classes you want;*
> *provide a full set of operations for each class;*
> *make commonality explicit by using inheritance.*

Where there is no such commonality, data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to a problem. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. In other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction, and the facilities needed for the support of object-oriented programming seem unnecessary.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When a system is designed – and even when the requirements for the system are written – commonality must be actively sought. Classes can be designed specifically as building blocks for other types, and existing classes can be examined to see if they exhibit similarities that can be exploited in a common base class.

For attempts to explain what object-oriented programming is without recourse to specific programming language constructs, see [Kerr,1987] and [Booch,1994] in §23.6.

Class hierarchies and abstract classes (§2.5.4) complement each other instead of being mutually exclusive (§12.5). In general, the paradigms listed here tend to be complementary and often

mutually supportive. For example, classes and modules contain functions, while modules contain classes and functions. The experienced designer applies a variety of paradigms as need dictates.

2.7 Generic Programming [tour.generic]

Someone who wants a stack is unlikely always to want a stack of characters. A stack is a general concept, independent of the notion of a character. Consequently, it ought to be represented independently.

More generally, if an algorithm can be expressed independently of representation details and if it can be done so affordably and without logical contortions, it ought to be done so.

The programming paradigm is:

> *Decide which algorithms you want;*
> *parameterize them so that they work for*
> *a variety of suitable types and data structures.*

2.7.1 Containers [tour.containers]

We can generalize a stack-of-characters type to a stack-of-anything type by making it a *template* and replacing the specific type ***char*** with a template parameter. For example:

```
template<class T> class Stack {
    T* v;
    int max_size;
    int top;
public:
    class Underflow { };
    class Overflow { };

    Stack(int s);     // constructor
    ~Stack();         // destructor

    void push(T);
    T pop();
};
```

The ***template<class T>*** prefix makes ***T*** a parameter of the declaration it prefixes.

The member functions might be defined similarly:

```
template<class T> void Stack<T>::push(T c)
{
    if (top == max_size) throw Overflow();
    v[top] = c;
    top = top + 1;
}
```

```
template<class T> T Stack<T>::pop()
{
    if (top == 0) throw Underflow();
    top = top - 1;
    return v[top];
}
```

Given these definitions, we can use stacks like this:

```
Stack<char> sc;          // stack of characters
Stack<complex> scplx;    // stack of complex numbers
Stack< list<int> > sli;  // stack of list of integers

void f()
{
    sc.push('c');
    if (sc.pop() != 'c') throw Bad_pop();

    scplx.push(complex(1,2));
    if (scplx.pop() != complex(1,2)) throw Bad_pop();
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates. A class holding a collection of elements of some type is commonly called a *container class*, or simply a *container*.

Templates are a compile-time mechanism so that their use incurs no run-time overhead compared to ''hand-written code.''

2.7.2 Generic Algorithms [tour.algorithms]

The C++ standard library provides a variety of containers, and users can write their own (Chapter 3, Chapter 17, Chapter 18). Thus, we find that we can apply the generic programming paradigm once more to parameterize algorithms by containers. For example, we want to sort, copy, and search ***vector***s, ***list***s, and arrays without having to write ***sort***( ), ***copy***( ), and ***search***( ) functions for each container. We also don't want to convert to a specific data structure accepted by a single sort function. Therefore, we must find a generalized way of defining our containers that allows us to manipulate one without knowing exactly which kind of container it is.

One approach, the approach taken for the containers and non-numerical algorithms in the C++ standard library (§3.8, Chapter 18) is to focus on the notion of a sequence and manipulate sequences through iterators.

Here is a graphical representation of the notion of a sequence:

A sequence has a beginning and an end. An iterator refers to an element, and provides an operation that makes the iterator refer to the next element of the sequence. The end of a sequence is an

iterator that refers one beyond the last element of the sequence. The physical representation of ‘‘the end’’ may be a sentinel element, but it doesn’t have to be. In fact, the point is that this notion of sequences covers a wide variety of representations, including lists and arrays.

We need some standard notation for operations such as ‘‘access an element through an iterator’’ and ‘‘make the iterator refer to the next element.’’ The obvious choices (once you get the idea) are to use the dereference operator * to mean ‘‘access an element through an iterator’’ and the increment operator ++ to mean ‘‘make the iterator refer to the next element.’’

Given that, we can write code like this:

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
    while (from != too_far) {
        *to = *from;      // copy element pointed to
        ++to;             // next input
        ++from;           // next output
    }
}
```

This copies any container for which we can define iterators with the right syntax and semantics.

C++’s built-in, low-level array and pointer types have the right operations for that, so we can write

```
char vc1[200]; // array of 200 characters
char vc2[500]; // array of 500 characters

void f()
{
    copy(&vc1[0],&vc1[200],&vc2[0]);
}
```

This copies *vc1* from its first element until its last into *vc2* starting at *vc2*’s first element.

All standard library containers (§16.3, Chapter 17) support this notion of iterators and sequences.

Two template parameters *In* and *Out* are used to indicate the types of the source and the target instead of a single argument. This was done because we often want to copy from one kind of container into another. For example:

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
    copy(&ac[0],&ac[200],lc.begin());
    copy(lc.begin(),lc.end(),vc.begin());
}
```

This copies the array to the *list* and the *list* to the *vector*. For a standard container, *begin*() is an iterator pointing to the first element.

2.8 Postscript [tour.post]

No programming language is perfect. Fortunately, a programming language does not have to be perfect to be a good tool for building great systems. In fact, a general-purpose programming language cannot be perfect for all of the many tasks to which it is put. What is perfect for one task is often seriously flawed for another because perfection in one area implies specialization. Thus, C++ was designed to be a good tool for building a wide variety of systems and to allow a wide variety of ideas to be expressed directly.

Not everything can be expressed directly using the built-in features of a language. In fact, that isn't even the ideal. Language features exist to support a variety of programming styles and techniques. Consequently, the task of learning a language should focus on mastering the native and natural styles for that language – not on the understanding of every little detail of all the language features.

In practical programming, there is little advantage in knowing the most obscure language features or for using the largest number of features. A single language feature in isolation is of little interest. Only in the context provided by techniques and by other features does the feature acquire meaning and interest. Thus, when reading the following chapters, please remember that the real purpose of examining the details of C++ is to be able to use them in concert to support good programming style in the context of sound designs.

2.9 Advice [tour.advice]

[1] Don't panic! All will become clear in time; §2.1.
[2] You don't have to know every detail of C++ to write good programs; §1.7.
[3] Focus on programming techniques, not on language features; §2.1.

.

3

A Tour of the Standard Library

Why waste time learning
when ignorance is instantaneous?
– Hobbes

Standard libraries — output — strings — input — vectors — range checking — lists — maps — container overview — algorithms — iterators — I/O iterators — traversals and predicates — algorithms using member functions — algorithm overview — complex numbers — vector arithmetic— standard library overview — advice.

3.1 Introduction [tour2.lib]

No significant program is written in just a bare programming language. First, a set of supporting libraries are developed. These then form the basis for further work.

Continuing Chapter 2, this chapter gives a quick tour of key library facilities to give you an idea what can be done using C++ and its standard library. Useful library types, such as ***string***, ***vector***, ***list***, and ***map***, are presented as well as the most common ways of using them. Doing this allows me to give better examples and to set better exercises in the following chapters. As in Chapter 2, you are strongly encouraged not to be distracted or discouraged by an incomplete understanding of details. The purpose of this chapter is to give you a taste of what is to come and to convey an understanding of the simplest uses of the most useful library facilities. A more detailed introduction to the standard library is given in §16.1.2.

The standard library facilities described in this book are part of every complete C++ implementation. In addition to the standard C++ library, most implementations offer ‘‘graphical user interface’’ systems, often referred to as GUIs or window systems, for interaction between a user and a program. Similarly, most application development environments provide ‘‘foundation libraries’’ that support corporate or industrial ‘‘standard’’ development and/or execution environments. I do not describe such systems and libraries. The intent is to provide a self-contained description of C++

as defined by the standard and to keep the examples portable, except where specifically noted. Naturally, a programmer is encouraged to explore the more extensive facilities available on most systems, but that is left to exercises.

3.2 Hello, world! [tour2.hello]

The minimal C++ program is

```
int main() { }
```

It defines a function called *main*, which takes no arguments and does nothing.

Every C++ program must have a function named *main*(). The program starts by executing that function. The *int* value returned by *main*(), if any, is the program's return value to ''the system.'' If no value is returned, the system will receive a value indicating successful completion. A nonzero value from *main*() indicates failure.

Typically, a program produces some output. Here is a program that writes out *Hello, world*!:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

The line *#include <iostream>* instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in *iostream*. Without these declarations, the expression

```
std::cout << "Hello, world!\n"
```

would make no sense. The operator << (''put to'') writes its second argument onto its first. In this case, the string literal "*Hello, world*!*n*" is written onto the standard output stream *std*::*cout*. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character \ followed by another character denotes a single special character. In this case, *n* is the newline character, so that the characters written are *Hello, world*! followed by a newline.

3.3 The Standard Library Namespace [tour2.name]

The standard library is defined in a namespace (§2.4, §8.2) called *std*. That is why I wrote *std*::*cout* rather than plain *cout*. I was being explicit about using the *standard cout*, rather than some other *cout*.

Every standard library facility is provided through some standard header similar to *<iostream>*. For example:

```
#include<string>
#include<list>
```

This makes the standard *string* and *list* available. To use them, the *std*:: prefix can be used:

```
std::string s = "Four legs Good; two legs Baaad!";
std::list<std::string> slogans;
```

For simplicity, I will rarely use the **std::** prefix explicitly in examples. Neither will I always **#include** the necessary headers explicitly. To compile and run the program fragments here, you must **#include** the appropriate headers (as listed in §3.7.5, §3.8.6, and Chapter 16). In addition, you must either use the **std::** prefix or make every name from **std** global (§8.2.3). For example:

```
#include<string>                    // make the standard string facilities accessible
using namespace std;                // make std names available without std:: prefix

string s = "Ignorance is bliss!";   // ok: string is std::string
```

It is generally in poor taste to dump every name from a namespace into the global namespace. However, to keep short the program fragments used to illustrate language and library features, I omit repetitive **#include**s and **std::** qualifications. In this book, I use the standard library almost exclusively, so if a name from the standard library is used, it either is a use of what the standard offers or part of an explanation of how the standard facility might be defined.

3.4 Output [tour2.ostream]

The iostream library defines output for every built-in type. Further, it is easy to define output of a user-defined type. By default, values output to **cout** are converted to a sequence of characters. For example,

```
void f()
{
    cout << 10;
}
```

will place the character **1** followed by the character **0** on the standard output stream. So will

```
void g()
{
    int i = 10;
    cout << i;
}
```

Output of different types can be combined in the obvious way:

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

If **i** has the value **10**, the output will be

```
the value of i is 10
```

A character constant is a character enclosed in single quotes. Note that a character constant is output as a character rather than as a numerical value. For example,

```
void k()
{
    cout << 'a';
    cout << 'b';
    cout << 'c';
}
```

will output *abc*.

People soon tire of repeating the name of the output stream when outputting several related items. Fortunately, the result of an output expression can itself be used for further output. For example:

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

This is equivalent to *h*( ). Streams are explained in more detail in Chapter 21.

3.5 Strings [tour2.string]

The standard library provides a *string* type to complement the string literals used earlier. The *string* type provides a variety of useful string operations, such as concatenation. For example:

```
string s1 = "Hello";
string s2 = "world";

void m1()
{
    string s3 = s1 + ", " + s2 + "!\n";

    cout << s3;
}
```

Here, *s3* is initialized to the character sequence

```
Hello, world!
```

followed by a newline. Addition of strings means concatenation. You can add strings, string literals, and characters to a string.

In many applications, the most common form of concatenation is adding something to the end of a string. This is directly supported by the += operation. For example:

```
void m2(string& s1, string& s2)
{
    s1 = s1 + '\n';  // append newline
    s2 += '\n';      // append newline
}
```

The two ways of adding to the end of a string are semantically equivalent, but I prefer the latter because it is more concise and likely to be more efficiently implemented.

Naturally, *string*s can be compared against each other and against string literals. For example:

```
string incantation;

void respond(const string& answer)
{
    if (answer == incantation) {
        // perform magic
    }
    else if (answer == "yes") {
        // ...
    }
    // ...
}
```

The standard library string class is described in Chapter 20. Among other useful features, it provides the ability to manipulate substrings. For example:

```
string name = "Niels Stroustrup";

void m3()
{
    string s = name.substr(6,10);        // s = "Stroustrup"
    name.replace(0,5,"Nicholas");        // name becomes "Nicholas Stroustrup"
}
```

The *substr*( ) operation returns a string that is a copy of the substring indicated by its arguments. The first argument is an index into the string (a position), and the second argument is the length of the desired substring. Since indexing starts from *0*, *s* gets the value *Stroustrup*.

The *replace*( ) operation replaces a substring with a value. In this case, the substring starting at *0* with length *5* is *Niels*; it is replaced by *Nicholas*. Thus, the final value of *name* is *Nicholas Stroustrup*. Note that the replacement string need not be the same size as the substring that it is replacing.

3.5.1 C-Style Strings [tour2.cstring]

A C-style string is a zero-terminated array of characters (§5.2.2). As shown, we can easily enter a C-style string into a *string*. To call functions that take C-style strings, we need to be able to extract the value of a *string* in the form of a C-style string. The *c_str*( ) function does that (§20.4.1). For example, we can print the *name* using the C output function *printf*( ) (§21.8) like this:

```
void f()
{
    printf("name: %s\n",name.c_str());
}
```

3.6 Input [tour2.istream]

The standard library offers ***istream***s for input. Like ***ostream***s, ***istream***s deal with character string representations of built-in types and can easily be extended to cope with user-defined types.

The operator >> (''get from'') is used as an input operator; ***cin*** is the standard input stream. The type of the right-hand operand of >> determines what input is accepted and what is the target of the input operation. For example,

```
void f()
{
    int i;
    cin >> i;   // read an integer into i

    double d;
    cin >> d;   // read a double-precision, floating-point number into d
}
```

reads a number, such as *1234*, from the standard input into the integer variable ***i*** and a floating-point number, such as *12.34e5*, into the double-precision, floating-point variable ***d***.

Here is an example that performs inch-to-centimeter and centimeter-to-inch conversions. You input a number followed by a character indicating the unit: centimeters or inches. The program then outputs the corresponding value in the other unit:

```
int main()
{
    const float factor = 2.54; // 1 inch equals 2.54 cm
    float x, in, cm;
    char ch = 0;

    cout << "enter length: ";

    cin >> x;          // read a floating-point number
    cin >> ch;         // read a suffix

    switch (ch) {
    case 'i':          // inch
        in = x;
        cm = x*factor;
        break;
    case 'c':          // cm
        in = x/factor;
        cm = x;
        break;
    default:
        in = cm = 0;
        break;
    }

    cout << in << " in = " << cm << " cm\n";
}
```

The *switch-statement* tests a value against a set of constants. The *break-statement*s are used to exit

the *switch-statement*. The case constants must be distinct. If the value tested does not match any of them, the ***default*** is chosen. The programmer need not provide a ***default***.

Often, we want to read a sequence of characters. A convenient way of doing that is to read into a ***string***. For example:

```
int main()
{
    string str;

    cout << "Please enter your name\n";
    cin >> str;
    cout << "Hello, " << str << "!\n";
}
```

If you type in

```
Eric
```

the response is

```
Hello, Eric!
```

By default, a whitespace character (§5.2.2) such as a space terminates the read, so if you enter

```
Eric Bloodaxe
```

pretending to be the ill-fated king of York, the response is still

```
Hello, Eric!
```

You can read a whole line using the ***getline()*** function. For example:

```
int main()
{
    string str;

    cout << "Please enter your name\n";
    getline(cin,str);
    cout << "Hello, " << str << "!\n";
}
```

With this program, the input

```
Eric Bloodaxe
```

yields the desired output:

```
Hello, Eric Bloodaxe!
```

The standard strings have the nice property of expanding to hold what you put in them, so if you enter a couple of megabytes of semicolons, the program will echo pages of semicolons back at you – unless your machine or operating system runs out of some critical resource first.

3.7 Containers [tour2.stl]

Much computing involves creating collections of various forms of objects and then manipulating such collections. Reading characters into a string and printing out the string is a simple example. A class with the main purpose of holding objects is commonly called a *container*. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program.

To illustrate the standard library's most useful containers, consider a simple program for keeping names and telephone numbers. This is the kind of program for which different approaches appear ''simple and obvious'' to people of different backgrounds.

3.7.1 Vector [tour2.vector]

For many C programmers, a built-in array of (name,number) pairs would seem to be a suitable starting point:

```
struct Entry {
    string name;
    int number;
};

Entry phone_book[1000];

void print_entry(int i)    // simple use
{
    cout << phone_book[i].name << ´ ´ << phone_book[i].number << ´\n´;
}
```

However, a built-in array has a fixed size. If we choose a large size, we waste space; if we choose a smaller size, the array will overflow. In either case, we will have to write low-level memory-management code. The standard library provides a ***vector*** (§16.3) that takes care of that:

```
vector<Entry> phone_book(1000);

void print_entry(int i)    // simple use, exactly as for array
{
    cout << phone_book[i].name << ´ ´ << phone_book[i].number << ´\n´;
}

void add_entries(int n) // increase size by n
{
    phone_book.resize(phone_book.size()+n);
}
```

The ***vector*** member function ***size()*** gives the number of elements.

Note the use of parentheses in the definition of ***phone_book***. We made a single object of type ***vector<Entry>*** and supplied its initial size as an initializer. This is very different from declaring a built-in array:

```
vector<Entry> book(1000);     // vector of 1000 elements
vector<Entry> books[1000];    // 1000 empty vectors
```

Should you make the mistake of using `[]` where you meant `()` when declaring a *vector*, your compiler will almost certainly catch the mistake and issue an error message when you try to use the *vector*.

A *vector* is a single object that can be assigned. For example:

```
void f(vector<Entry>& v)
{
    vector<Entry> v2 = phone_book;
    v = v2;
    // ...
}
```

Assigning a *vector* involves copying its elements. Thus, after the initialization and assignment in *f*( ), *v* and *v2* each holds a separate copy of every *Entry* in the phone book. When a *vector* holds many elements, such innocent-looking assignments and initializations can be prohibitively expensive. Where copying is undesirable, references or pointers should be used.

3.7.2 Range Checking [tour2.range]

The standard library *vector* does not provide range checking by default (§16.3.3). For example:

```
void f()
{
    int i = phone_book[1001].number;    // 1001 is out of range
    // ...
}
```

The initialization is likely to place some random value in *i* rather than giving an error. This is undesirable, so I will use a simple range-checking adaptation of *vector*, called *Vec*, in the following chapters. A *Vec* is like a *vector*, except that it throws an exception of type *out_of_range* if a subscript is out of range.

Techniques for implementing types such as *Vec* and for using exceptions effectively are discussed in §11.12, §8.3, and Chapter 14. However, the definition here is sufficient for the examples in this book:

```
template<class T> class Vec : public vector<T> {
public:
    Vec() : vector<T>() { }
    Vec(int s) : vector<T>(s) { }

    T& operator[](int i) { return at(i); }                  // range-checked
    const T& operator[](int i) const { return at(i); }      // range-checked
};
```

The *at*( ) operation is a *vector* subscript operation that throws an exception of type *out_of_range* if its argument is out of the *vector*'s range (§16.3.3).

Returning to the problem of keeping names and telephone numbers, we can now use a *Vec* to ensure that out-of-range accesses are caught. For example:

```
Vec<Entry> phone_book(1000);
```

```
void print_entry(int i)      // simple use, exactly as for vector
{
        cout << phone_book[i].name << ´ ´ << phone_book[i].number << ´\n´;
}
```

An out-of-range access will throw an exception that the user can catch. For example:

```
void f()
{
        try {
                for (int i = 0; i<10000; i++) print_entry(i);
        }
        catch (out_of_range) {
                cout << "range error\n";
        }
}
```

The exception will be thrown, and then caught, when *phone_book[i]* is tried with *i==1000*. If the user doesn't catch this kind of exception, the program will terminate in a well-defined manner rather than proceeding or failing in an undefined manner. One way to minimize surprises from exceptions is to use a *main()* with a *try-block* as its body:

```
int main()
try {
        // your code
}
catch (out_of_range) {
        cerr << "range error\n";
}
catch (...) {
        cerr << "unknown exception thrown\n";
}
```

This provides default exception handlers so that if we fail to catch some exception, an error message is printed on the standard error-diagnostic output stream *cerr* (§21.2.1).

3.7.3 List [tour2.list]

Insertion and deletion of phone book entries could be common. Therefore, a list could be more appropriate than a vector for representing a simple phone book. For example:

```
list<Entry> phone_book;
```

When we use a list, we tend not to access elements using subscripting the way we commonly do for vectors. Instead, we might search the list looking for an element with a given value. To do this, we take advantage of the fact that a *list* is a sequence as described in §3.8:

```
void print_entry(const string& s)
{
        typedef list<Entry>::const_iterator LI;
```

```
        for (LI i = phone_book.begin(); i != phone_book.end(); ++i) {
            Entry& e = *i;   // reference used as shorthand
            if (s == e.name) cout << e.name << ´ ´ << e.number << ´\n´;
        }
    }
```

The search for ***s*** starts at the beginning of the list and proceeds until either ***s*** is found or the end is reached. Every standard library container provides the functions ***begin*( )** and ***end*( )**, which return an iterator to the first and to one-past-the-last element, respectively (§16.3.2). Given an iterator ***i***, the next element is ***++i***. Given an iterator ***i***, the element it refers to is ****i***.

A user need not know the exact type of the iterator for a standard container. That iterator type is part of the definition of the container and can be referred to by name. When we don't need to modify an element of the container, ***const_iterator*** is the type we want. Otherwise, we use the plain ***iterator*** type (§16.3.1).

Adding elements to a ***list*** is easy:

```
void add_entry(Entry& e, list<Entry>::iterator i)
{
    phone_book.push_front(e);      // add at beginning
    phone_book.push_back(e);       // add at end
    phone_book.insert(i,e);        // add before the element 'i' refers to
}
```

3.7.4 Map [tour2.map]

Writing code to look up a name in a list of (name,number) pairs is really quite tedious. In addition, a linear search is quite inefficient for all but the shortest lists. Other data structures directly support insertion, deletion, and searching based on values. In particular, the standard library provides the ***map*** type (§17.4.1). A ***map*** is a container of pairs of values. For example:

```
map<string,int> phone_book;
```

In other contexts, a ***map*** is known as an associative array or a dictionary.

When indexed by a value of its first type (called the *key*) a ***map*** returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

```
void print_entry(const string& s)
{
    if (int i = phone_book[s]) cout << s << ´ ´ << i << ´\n´;
}
```

If no match was found for the key ***s***, a default value is returned from the ***phone_book***. The default value for an integer type in a ***map*** is ***0***. Here, I assume that ***0*** isn't a valid telephone number.

3.7.5 Standard Containers [tour2.stdcontainer]

A ***map***, a ***list***, and a ***vector*** can each be used to represent a phone book. However, each has strengths and weaknesses. For example, subscripting a ***vector*** is cheap and easy. On the other hand, inserting an element between two elements tends to be expensive. A ***list*** has exactly the

opposite properties. A ***map*** resembles a ***list*** of (key,value) pairs except that it is optimized for finding values based on keys.

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

Standard Container Summary	
vector<T>	A variable-sized vector (§16.3)
list<T>	A doubly-linked list (§17.2.2)
queue<T>	A queue (§17.3.2)
stack<T>	A stack (§17.3.1)
deque<T>	A double-ended queue (§17.2.3)
priority_queue<T>	A queue sorted by value (§17.3.3)
set<T>	A set (§17.4.3)
multiset<T>	A set in which a value can occur many times (§17.4.4)
map<key,val>	An associative array (§17.4.1)
multimap<key,val>	A map in which a key can occur many times (§17.4.2)

The standard containers are presented in §16.2, §16.3, and Chapter 17. The containers are defined in namespace ***std*** and presented in headers *<vector>*, *<list>*, *<map>*, etc. (§16.2).

The standard containers and their basic operations are designed to be similar from a notational point of view. Furthermore, the meanings of the operations are equivalent for the various containers. In general, basic operations apply to every kind of container. For example, ***push_back*( )** can be used (reasonably efficiently) to add elements to the end of a ***vector*** as well as for a ***list***, and every container has a ***size*( )** member function that returns its number of elements.

This notational and semantic uniformity enables programmers to provide new container types that can be used in a very similar manner to the standard ones. The range-checked vector, ***Vec*** (§3.7.2), is an example of that. Chapter 17 demonstrates how a ***hash_map*** can be added to the framework. The uniformity of container interfaces also allows us to specify algorithms independently of individual container types.

3.8 Algorithms [tour2.algorithms]

A data structure, such as a list or a vector, is not very useful on its own. To use one, we need operations for basic access such as adding and removing elements. Furthermore, we rarely just store objects in a container. We sort them, print them, extract subsets, remove elements, search for objects, etc. Consequently, the standard library provides the most common algorithms for containers in addition to providing the most common container types. For example, the following sorts a ***vector*** and places a copy of each unique ***vector*** element on a ***list***:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), le.begin());
}
```

The standard algorithms are described in Chapter 18. They are expressed in terms of sequences of elements (§2.7.2). A sequence is represented by a pair of iterators specifying the first element and the one-beyond-the-last element. In the example, ***sort*( )** sorts the sequence from ***ve*.*begin*( )** to ***ve*.*end*( )** – which just happens to be all the elements of a ***vector***. For writing, you need only to specify the first element to be written. If more than one element is written, the elements following that initial element will be overwritten.

If we wanted to add the new elements to the end of a container, we could have written:

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), back_inserter(le));    // append to le
}
```

A ***back_inserter*( )** adds elements at the end of a container, extending the container to make room for them (§19.2.4). C programmers will appreciate that the standard containers plus ***back_inserter*( )**s eliminate the need to use error-prone, explicit C-style memory management using ***realloc*( )** (§16.3.5). Forgetting to use a ***back_inserter*( )** when appending can lead to errors. For example:

```
void f(list<Entry>& ve, vector<Entry>& le)
{
    copy(ve.begin(), ve.end(), le);            // error: le not an iterator
    copy(ve.begin(), ve.end(), le.end());      // bad: writes beyond the end
    copy(ve.begin(), ve.end(), le.begin());    // overwrite elements
}
```

3.8.1 Use of Iterators [tour2.iteruse]

When you first encounter a container, a few iterators referring to useful elements can be obtained; ***begin*( )** and ***end*( )** are the best examples of this. In addition, many algorithms return iterators. For example, the standard algorithm ***find*** looks for a value in a sequence and returns an iterator to the element found. Using ***find***, we can write a function that counts the number of occurrences of a character in a ***string***:

```
int count(const string& s, char c)
{
    string::const_iterator i = find(s.begin(), s.end(), c);
    int n = 0;
    while (i != s.end()) {
        ++n;
        i = find(i+1, s.end(), c);
    }
    return n;
}
```

The ***find*** algorithm returns an iterator to the first occurrence of a value in a sequence or the one-past-the-end iterator. Consider what happens for a simple call of ***count***:

```
void f()
{
    string m = "Mary had a little lamb";
    int a_count = count(m, 'a');
}
```

The first call to *find*( ) finds the ´*a*´ in *Mary*. Thus, the iterator points to that character and not to *s*.*end*( ), so we enter the loop. In the loop, we start the search at *i+1*; that is, we start one past where we found the ´*a*´. We then loop finding the other three ´*a*´s. That done, *find*( ) reaches the end and returns *s*.*end*( ) so that the condition *i*!=*s*.*end*( ) fails and we exit the loop.

That call of *count*( ) could be graphically represented like this:

The arrows indicate the initial, intermediate, and final values of the iterator *i*.

Naturally, the *find* algorithm will work equivalently on every standard container. Consequently, we could generalize the *count*( ) function in the same way:

```
template<class C, class T> int count(const C& v, T val)
{
    typename C::const_iterator i = find(v.begin(), v.end(), val);  // "typename;" see §C.13.5
    int n = 0;
    while (i != v.end()) {
        ++n;
        ++i; // skip past the element we just found
        i = find(i, v.end(), val);
    }
    return n;
}
```

This works, so we can say:

```
void f(list<complex>& lc, vector<string>& vc, string s)
{
    int i1 = count(lc, complex(1,3));
    int i2 = count(vc, "Chrysippus");
    int i3 = count(s, 'x');
}
```

However, we don't have to define a *count* template. Counting occurrences of an element is so generally useful that the standard library provides that algorithm. To be fully general, the standard library *count* takes a sequence as its argument, rather than a container, so we would say:

```
void f(list<complex>& lc, vector<string>& vs, string s)
{
    int i1 = count(lc.begin(), lc.end(), complex(1,3));
    int i2 = count(vs.begin(), vs.end(), "Diogenes");
    int i3 = count(s.begin(), s.end(), 'x');
}
```

The use of a sequence allows us to use ***count*** for a built-in array and also to count parts of a container. For example:

```
void g(char cs[], int sz)
{
    int i1 = count(&cs[0], &cs[sz], 'z');        // 'z's in array
    int i2 = count(&cs[0], &cs[sz/2], 'z');      // 'z's in first half of array
}
```

3.8.2 Iterator Types [tour2.iter]

What are iterators really? Any particular iterator is an object of some type. There are, however, many different iterator types because an iterator needs to hold the information necessary for doing its job for a particular container type. These iterator types can be as different as the containers and the specialized needs they serve. For example, a ***vector***'s iterator is most likely an ordinary pointer because a pointer is quite a reasonable way of referring to an element of a ***vector***:

Alternatively, a ***vector*** iterator could be implemented as a pointer to the ***vector*** plus an index:

Using such an iterator would allow range checking (§19.3).

A list iterator must be something more complicated than a simple pointer to an element because an element of a list in general does not know where the next element of that list is. Thus, a list iterator might be a pointer to a link:

What is common for all iterators is their semantics and the naming of their operations. For example, applying ++ to any iterator yields an iterator that refers to the next element. Similarly, * yields the element to which the iterator refers. In fact, any object that obeys a few simple rules like these is an iterator (§19.2.1). Furthermore, users rarely need to know the type of a specific iterator; each container ''knows'' its iterator types and makes them available under the conventional names *iterator* and *const_iterator*. For example, *list<Entry>::iterator* is the general iterator type for *list<Entry>*. I rarely have to worry about the details of how that type is defined.

3.8.3 Iterators and I/O [tour2.ioiterators]

Iterators are a general and useful concept for dealing with sequences of elements in containers. However, containers are not the only place where we find sequences of elements. For example, an input stream produces a sequence of values and we write a sequence of values to an output stream. Consequently, the notion of iterators can be usefully applied to input and output.

To make an *ostream_iterator*, we need to specify which stream will be used and the type of objects written to it. For example, we can define an iterator that refers to the standard output stream, *cout*:

```
ostream_iterator<string> oo(cout);
```

The effect of assigning to **oo* is to write the assigned value to *cout*. For example:

```
int main()
{
        *oo = "Hello, ";        // meaning cout << "Hello, "
        ++oo;
        *oo = "world!\n";       // meaning cout << "world!\n"
}
```

This is yet another way of writing the canonical message to standard output. The *++oo* is done to mimic writing into an array through a pointer. This way wouldn't be my first choice for that simple task, but the utility of treating output as a write-only container will soon be obvious – if it isn't already.

Similarly, an *istream_iterator* is something that allows us to treat an input stream as a read-only container. Again, we must specify the stream to be used and the type of values expected:

```
istream_iterator<string> ii(cin);
```

Because input iterators invariably appear in pairs representing a sequence, we must provide an

istream_iterator to indicate the end of input. This is the default ***istream_iterator***:

```
istream_iterator<string> eos;
```

We could now read ***Hello, world!*** from input and write it out again like this:

```
int main()
{
    string s1 = *ii;
    ++ii;
    string s2 = *ii;

    cout << s1 << ´ ´ << s2 << ´\n´;
}
```

Actually, ***istream_iterator***s and ***ostream_iterator***s are not meant to be used directly. Instead, they are typically provided as arguments to algorithms. For example, we can write a simple program to read a file, sort the words read, eliminate duplicates, and write the result to another file:

```
int main()
{
    string from, to;
    cin >> from >> to;                      // get source and target file names

    ifstream is(from.c_str());              // input stream (c_str(); see §3.5)
    istream_iterator<string> ii(is);        // input iterator for stream
    istream_iterator<string> eos;           // input sentinel

    vector<string> b(ii,eos);               // b is a vector initialized from input
    sort(b.begin(),b.end());                // sort the buffer

    ofstream os(to.c_str());                // output stream
    ostream_iterator<string> oo(os,"\n");   // output iterator for stream

    unique_copy(b.begin(),b.end(),oo);      // copy buffer to output,
                                            // discard replicated values

    return !is.eof() && !os;                // return error state (§3.2, §21.3.3)
}
```

An ***ifstream*** is an ***istream*** that can be attached to a file, and an ***ofstream*** is an ***ostream*** that can be attached to a file. The ***ostream_iterator***'s second argument is used to delimit output values.

3.8.4 Traversals and Predicates [tour2.traverse]

Iterators allow us to write loops to iterate through a sequence. However, writing loops can be tedious, so the standard library provides ways for a function to be called for each element of a sequence.

Consider writing a program that reads words from input and records the frequency of their occurrence. The obvious representation of the strings and their associated frequencies is a ***map***:

```
map<string,int> histogram;
```

The obvious action to be taken for each string to record its frequency is:

```
void record(const string& s)
{
    histogram[s]++;     // record frequency of ''s''
}
```

Once the input has been read, we would like to output the data we have gathered. The *map* consists of a sequence of (string,int) pairs. Consequently, we would like to call

```
void print(const pair<const string,int>& r)
{
    cout << r.first << ´ ´ << r.second << ´\n´;
}
```

for each element in the map (the first element of a *pair* is called *first*, and the second element is called *second*). The first element of the *pair* is a *const string* rather than a plain *string* because all *map* keys are constants.

Thus, the main program becomes:

```
int main()
{
    istream_iterator<string> ii(cin);
    istream_iterator<string> eos;

    for_each(ii,eos,record);
    for_each(histogram.begin(),histogram.end(),print);
}
```

Note that we don't need to sort the *map* to get the output in order. A *map* keeps its elements ordered so that an iteration traverses the *map* in (increasing) order.

Many programming tasks involve looking for something in a container rather than simply doing something to every element. For example, the *find* algorithm (§18.5.2) provides a convenient way of looking for a specific value. A more general variant of this idea looks for an element that fulfills a specific requirement. For example, we might want to search a *map* for the first value larger than *42*. A *map* is a sequence of (key,value) pairs, so we search that list for a *pair<const string,int>* where the *int* is greater than *42*:

```
bool gt_42(const pair<const string,int>& r)
{
    return r.second>42;
}

void f(map<string,int>& m)
{
    typedef map<string,int>::const_iterator MI;
    MI i = find_if(m.begin(),m.end(),gt_42);
    // ...
}
```

Alternatively, we could count the number of words with a frequency higher than 42:

```
void g(const map<string,int>& m)
{
    int c42 = count_if(m.begin(),m.end(),gt_42);
    // ...
}
```

A function, such as *gt_42*( ), that is used to control the algorithm is called a *predicate*. A predicate is called for each element and returns a Boolean value, which the algorithm uses to perform its intended action. For example, *find_if*( ) searches until its predicate returns *true* to indicate that an element of interest has been found. Similarly, *count_if*( ) counts the number of times its predicate is *true*.

The standard library provides a few useful predicates and some templates that are useful for creating more (§18.4.2).

3.8.5 Algorithms Using Member Functions [tour2.memp]

Many algorithms apply a function to elements of a sequence. For example, in §3.8.4

```
for_each(ii,eos,record);
```

calls *record*( ) to read strings from input.

Often, we deal with containers of pointers and we really would like to call a member function of the object pointed to, rather than a global function on the pointer. For example, we might want to call the member function *Shape*::*draw*( ) for each element of a *list<Shape*>*. To handle this specific example, we simply write a nonmember function that invokes the member function. For example:

```
void draw(Shape* p)
{
    p->draw();
}
void f(list<Shape*>& sh)
{
    for_each(sh.begin(),sh.end(),draw);
}
```

By generalizing this technique, we can write the example like this:

```
void g(list<Shape*>& sh)
{
    for_each(sh.begin(),sh.end(),mem_fun(&Shape::draw));
}
```

The standard library *mem_fun*( ) template (§18.4.4.2) takes a pointer to a member function (§15.5) as its argument and produces something that can be called for a pointer to the member's class. The result of *mem_fun*(&*Shape*::*draw*) takes a *Shape** argument and returns whatever *Shape*::*draw*( ) returns.

The *mem_fun*( ) mechanism is important because it allows the standard algorithms to be used for containers of polymorphic objects.

3.8.6 Standard Library Algorithms [tour2.algolist]

What is an algorithm? A general definition of an algorithm is ‘‘a finite set of rules which gives a sequence of operations for solving a specific set of problems [and] has five important features: Finiteness ... Definiteness ... Input ... Output ... Effectiveness’’ [Knuth,1968,§1.1]. In the context of the C++ standard library, an algorithm is a set of templates operating on sequences of elements.

The standard library provides dozens of algorithms. The algorithms are defined in namespace *std* and presented in the *<algorithm>* header. Here are a few I have found particularly useful:

Selected Standard Algorithms	
for_each()	Invoke function for each element (§18.5.1)
find()	Find first occurrence of arguments (§18.5.2)
find_if()	Find first match of predicate (§18.5.2)
count()	Count occurrences of element (§18.5.3)
count_if()	Count matches of predicate (§18.5.3)
replace()	Replace element with new value (§18.6.4)
replace_if()	Replace element that matches predicate with new value (§18.6.4)
copy()	Copy elements (§18.6.1)
unique_copy()	Copy elements that are not duplicates (§18.6.1)
sort()	Sort elements (§18.7.1)
equal_range()	Find all elements with equivalent values (§18.7.2)
merge()	Merge sorted sequences (§18.7.3)

These algorithms, and many more (see Chapter 18), can be applied to elements of containers, *string*s, and built-in arrays.

3.9 Math [tour2.math]

Like C, C++ wasn’t designed primarily with numerical computation in mind. However, a lot of numerical work is done in C++, and the standard library reflects that.

3.9.1 Complex Numbers [tour2.complex]

The standard library supports a family of complex number types along the lines of the *complex* class described in §2.5.2. To support complex numbers where the scalars are single-precision, floating-point numbers (*float*s), double precision numbers (*double*s), etc., the standard library *complex* is a template:

```
template<class scalar> class complex {
public:
    complex(scalar re, scalar im);
    // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
// standard exponentiation function from <complex>:
template<class C> complex<C> pow(const complex<C>&, int);

void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld = fl+sqrt(db);
    db += fl*3;
    fl = pow(1/fl,2);
    // ...
}
```

For more details, see §22.5.

3.9.2 Vector Arithmetic [tour2.valarray]

The ***vector*** described in §3.7.1 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to ***vector*** would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides a vector, called ***valarray***, that is less general and more amenable to optimization for numerical computation:

```
template<class T> class valarray {
    // ...
    T& operator[](size_t);
    // ...
};
```

The type ***size_t*** is the unsigned integer type that the implementation uses for array indices.

The usual arithmetic operations and the most common mathematical functions are supported for ***valarray***s. For example:

```
// standard absolute value function from <valarray>:
template<class T> valarray<T> abs(const valarray<T>&);

void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}
```

For more details, see §22.4.

3.9.3 Basic Numeric Support [tour2.basicnum]

Naturally, the standard library contains the most common mathematical functions – such as ***log***( ), ***pow***( ), and ***cos***( ) – for floating-point types; see §22.3. In addition, classes that describe the properties of built-in types – such as the maximum exponent of a ***float*** – are provided; see §22.2.

3.10 Standard Library Facilities [tour2.post]

The facilities provided by the standard library can be classified like this:

[1] Basic run-time language support (e.g., for allocation and run-time type information); see §16.1.3.

[2] The C standard library (with very minor modifications to minimize violations of the type system); see §16.1.2.

[3] Strings and I/O streams (with support for international character sets and localization); see Chapter 20 and Chapter 21.

[4] A framework of containers (such as ***vector***, ***list***, and ***map***) and algorithms using containers (such as general traversals, sorts, and merges); see Chapter 16, Chapter 17, Chapter 18, and Chapter 19.

[5] Support for numerical computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization); see Chapter 22.

The main criterion for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be easy to learn. Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

Every algorithm works with every container without the use of conversions. This framework, conventionally called the STL [Stepanov,1994], is extensible in the sense that users can easily provide containers and algorithms in addition to the ones provided as part of the standard and have these work directly with the standard containers and algorithms.

3.11 Advice [tour2.advice]

[1] Don't reinvent the wheel; use libraries.

[2] Don't believe in magic; understand what your libraries do, how they do it, and at what cost they do it.

[3] When you have a choice, prefer the standard library to other libraries.

[4] Do not think that the standard library is ideal for everything.

[5] Remember to ***#include*** the headers for the facilities you use; §3.3.

[6] Remember that standard library facilities are defined in namespace ***std***; §3.3.

[7] Use ***string*** rather than ***char****; §3.5, §3.6.

[8] If in doubt use a range-checked vector (such as ***Vec***); §3.7.2.

[9] Prefer ***vector<T>***, ***list<T>***, and ***map<key, value>*** to ***T***[ ]; §3.7.1, §3.7.3, §3.7.4.

[10] When adding elements to a container, use ***push_back***( ) or ***back_inserter***( ); §3.7.3, §3.8.

[11] Use ***push_back***( ) on a ***vector*** rather than ***realloc***( ) on an array; §3.8.

[12] Catch common exceptions in ***main***( ); §3.7.2.

Part I

Basic Facilities

This part describes C++'s built-in types and the basic facilities for constructing programs out of them. The C subset of C++ is presented together with C++'s additional support for traditional styles of programming. It also discusses the basic facilities for composing a C++ program out of logical and physical parts.

Chapters

4 Types and Declarations
5 Pointers, Arrays, and Structures
6 Expressions and Statements
7 Functions
8 Namespaces and Exceptions
9 Source Files and Programs

4

Types and Declarations

Accept nothing short of perfection!
– anon

Perfection is achieved
only on the point of collapse.
– C. N. Parkinson

Types — fundamental types — Booleans — characters — character literals — integers — integer literals — floating-point types — floating-point literals — sizes — ***void*** — enumerations — declarations — names — scope — initialization — objects — ***typedef***s — advice — exercises.

4.1 Types [dcl.type]

Consider

```
x = y+f(2);
```

For this to make sense in a C++ program, the names ***x***, ***y***, and ***f*** must be suitably declared. That is, the programmer must specify that entities named ***x***, ***y***, and ***f*** exist and that they are of types for which **=** (assignment), **+** (addition), and **( )** (function call), respectively, are meaningful.

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example, the declarations

```
float x;          // x is a floating-point variable
int y = 7;        // y is an integer variable with the initial value 7
float f(int);     // f is a function taking an argument of type int and returning a floating-point number
```

would make the example meaningful. Because ***y*** is declared to be an ***int***, it can be assigned to, used in arithmetic expressions, etc. On the other hand, ***f*** is declared to be a function that takes an ***int*** as its argument, so it can be called given a suitable argument.

This chapter presents fundamental types (§4.1.1) and declarations (§4.9). Its examples just demonstrate language features; they are not intended to do anything useful. More extensive and realistic examples are saved for later chapters after more of C++ has been described. This chapter simply provides the most basic elements from which C++ programs are constructed. You must know these elements, plus the terminology and simple syntax that goes with them, in order to complete a real project in C++ and especially to read code written by others. However, a thorough understanding of every detail mentioned in this chapter is not a requirement for understanding the following chapters. Consequently, you may prefer to skim through this chapter, observing the major concepts, and return later as the need for understanding of more details arises.

4.1.1 Fundamental Types [dcl.fundamental]

C++ has a set of fundamental types corresponding to the most common basic storage units of a computer and the most common ways of using them to hold data:

§4.2 A Boolean type (***bool***)
§4.3 Character types (such as ***char***)
§4.4 Integer types (such as ***int***)
§4.5 Floating-point types (such as ***double***)

In addition, a user can define

§4.8 Enumeration types for representing specific sets of values (***enum***)

There also is

§4.7 A type, ***void***, used to signify the absence of information

From these types, we can construct other types:

§5.1 Pointer types (such as ***int****)
§5.2 Array types (such as ***char***[])
§5.5 Reference types (such as ***double***&)
§5.7 Data structures and classes (Chapter 10)

The Boolean, character, and integer types are collectively called *integral types*. The integral and floating-point types are collectively called *arithmetic types*. Enumerations and classes (Chapter 10) are called *user-defined types* because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, other types are called *built-in types.*

The integral and floating-point types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations (§4.6). The assumption is that a computer provides bytes for holding characters, words for holding and computing integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with pointers and arrays present these machine-level notions to the programmer in a reasonably implementation-independent manner.

For most applications, one could simply use ***bool*** for logical values, ***char*** for characters, ***int*** for integer values, and ***double*** for floating-point values. The remaining fundamental types are

variations for optimizations and special needs that are best ignored until such needs arise. They must be known, however, to read old C and C++ code.

4.2 Booleans [dcl.bool]

A Boolean, ***bool***, can have one of the two values ***true*** or ***false***. A Boolean is used to express the results of logical operations. For example:

```
void f(int a, int b)
{
    bool b1 = a==b;     // = is assignment, == is equality
    // ...
}
```

If ***a*** and ***b*** have the same value, ***b1*** becomes ***true***; otherwise, ***b1*** becomes ***false***.

A common use of ***bool*** is as the type of the result of a function that tests some condition (a predicate). For example:

```
bool is_open(File*);

bool greater(int a, int b) { return a>b; }
```

By definition, ***true*** has the value ***1*** when converted to an integer and ***false*** has the value ***0***. Conversely, integers can be implicitly converted to ***bool*** values: nonzero integers convert to ***true*** and ***0*** converts to ***false***. For example:

```
bool b = 7;     // bool(7) is true, so b becomes true
int i = true;   // int(true) is 1, so i becomes 1
```

In arithmetic and logical expressions, ***bool***s are converted to ***int***s; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to ***bool***, a ***0*** is converted to ***false*** and a nonzero value is converted to ***true***.

```
void g()
{
    bool a = true;
    bool b = true;

    bool x = a+b;   // a+b is 2, so x becomes true
    bool y = a|b;   // a|b is 1, so y becomes true
}
```

A pointer can be implicitly converted to a ***bool*** (§C.6.2.5). A nonzero pointer converts to ***true***; zero-valued pointers convert to ***false***.

4.3 Character Types [dcl.char]

A variable of type ***char*** can hold a character of the implementation's character set. For example:

```
char ch = 'a';
```

Almost universally, a ***char*** has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Many problems arise from the fact that this set of characters is only partially standardized (§C.3).

Serious variations occur between character sets supporting different natural languages and also between different character sets supporting the same natural language in different ways. However, here we are interested only in how such differences affect the rules of C++. The larger and more interesting issue of how to program in a multi-lingual, multi-character-set environment is beyond the scope of this book, although it is alluded to in several places (§20.2, §21.7, §C.3.3).

It is safe to assume that the implementation character set includes the decimal digits, the 26 alphabetic characters of English, and some of the basic punctuation characters. It is not safe to assume that there are no more than 127 characters in an 8-bit character set (e.g., some sets provide 255 characters), that there are no more alphabetic characters than English provides (most European languages provide more), that the alphabetic characters are contiguous (EBCDIC leaves a gap between ´*i*´ and ´*j*´), or that every character used to write C++ is available (e.g., some national character sets do not provide { } [ ] | \; §C.3.1). Whenever possible, we should avoid making assumptions about the representation of objects. This general rule applies even to characters.

Each character constant has an integer value. For example, the value of ´*b*´ is ***98*** in the ASCII character set. Here is a small program that will tell you the integer value of any character you care to input:

```
#include <iostream>

int main()
{
    char c;
    std::cin >> c;
    std::cout << "the value of ´" << c << "´ is " << int(c) << ´\n´;
}
```

The notation ***int***(*c*) gives the integer value for a character *c*. The possibility of converting a ***char*** to an integer raises the question: is a ***char*** signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values ***0*** to ***255*** or as the values ***-127*** to ***127***. Unfortunately, which choice is made for a plain ***char*** is implementation-defined (§C.1, §C.3.4). C++ provides two types for which the answer is definite; ***signed char***, which can hold at least the values ***-127*** to ***127***, and ***unsigned char***, which can hold at least the values ***0*** to ***255***. Fortunately, the difference matters only for values outside the ***0*** to ***127*** range, and the most common characters are within that range.

Values outside that range stored in a plain ***char*** can lead to subtle portability problems. See §C.3.4 if you need to use more than one type of ***char*** or if you store integers in ***char*** variables.

A type ***wchar_t*** is provided to hold characters of a larger character set such as Unicode. It is a distinct type. The size of ***wchar_t*** is implementation-defined and large enough to hold the largest character set supported by the implementation's locale (see §21.7, §C.3.3). The strange name is a leftover from C. In C, ***wchar_t*** is a ***typedef*** (§4.9.7) rather than a built-in type. The suffix ***_t*** was added to distinguish standard ***typedef***s.

Note that the character types are integral types (§4.1.1) so that arithmetic and logical operations (§6.2) apply.

4.3.1 Character Literals [dcl.char.lit]

A character literal, often called a character constant, is a character enclosed in single quotes, for example, ´*a*´ and ´*0*´. The type of a character literal is ***char***. Such character literals are really symbolic constants for the integer value of the characters in the character set of the machine on which the C++ program is to run. For example, if you are running on a machine using the ASCII character set, the value of ´*0*´ is *48*. The use of character literals rather than decimal notation makes programs more portable. A few characters also have standard names that use the backslash \ as an escape character. For example, ***\n*** is a newline and ***\t*** is a horizontal tab. See §C.3.2 for details about escape characters.

Wide character literals are of the form ***L´ab´***, where the number of characters between the quotes and their meanings is implementation-defined to match the ***wchar_t*** type. A wide character literal has type ***wchar_t***.

4.4 Integer Types [dcl.int]

Like ***char***, each integer type comes in three forms: ‘‘plain’’ ***int***, ***signed int***, and ***unsigned int***. In addition, integers come in three sizes: ***short int***, ‘‘plain’’ ***int***, and ***long int***. A ***long int*** can be referred to as plain ***long***. Similarly, ***short*** is a synonym for ***short int***, ***unsigned*** for ***unsigned int***, and ***signed*** for ***signed int***.

The ***unsigned*** integer types are ideal for uses that treat storage as a bit array. Using an ***unsigned*** instead of an ***int*** to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables ***unsigned*** will typically be defeated by the implicit conversion rules (§C.6.1, §C.6.2.1).

Unlike plain ***char***s, plain ***int***s are always signed. The signed ***int*** types are simply more explicit synonyms for their plain ***int*** counterparts.

4.4.1 Integer Literals [dcl.int.lit]

Integer literals come in four guises: decimal, octal, hexadecimal, and character literals. Decimal literals are the most commonly used and look as you would expect them to:

```
0   1234   976   12345678901234567890
```

The compiler ought to warn about literals that are too long to represent.

A literal starting with zero followed by ***x*** (***0x***) is a hexadecimal (base 16) number. A literal starting with zero followed by a digit is an octal (base 8) number. For example:

decimal:	*0*	*2*	*63*	*83*
octal:	*00*	*02*	*077*	*0123*
hexadecimal:	*0x0*	*0x2*	*0x3f*	*0x53*

The letters ***a***, ***b***, ***c***, ***d***, ***e***, and ***f***, or their uppercase equivalents, are used to represent *10*, *11*, *12*, *13*, *14*, and *15*, respectively. Octal and hexadecimal notations are most useful for expressing bit patterns. Using these notations to express genuine numbers can lead to surprises. For example, on a machine on which an ***int*** is represented as a two’s complement 16-bit integer, ***0xffff*** is the negative decimal number *-1*. Had more bits been used to represent an integer, it would have been *65535*.

The suffix ***U*** can be used to write explicitly ***unsigned*** literals. Similarly, the suffix ***L*** can be used to write explicitly ***long*** literals. For example, ***3*** is an ***int***, 3U is an ***unsigned int***, and ***3L*** is a ***long int***. If no suffix is provided, the compiler gives an integer literal a suitable type based on its value and the implementation's integer sizes (§C.4).

It is a good idea to limit the use of nonobvious constants to a few well-commented ***const*** (§5.4) or enumerator (§4.8) initializers.

4.5 Floating-Point Types [dcl.float]

The floating-point types represent floating-point numbers. Like integers, floating-point types come in three sizes: ***float*** (single-precision), ***double*** (double-precision), and ***long double*** (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use ***double*** and hope for the best.

4.5.1 Floating-Point Literals [dcl.fp.lit]

By default, a floating-point literal is of type ***double***. Again, a compiler ought to warn about floating-point literals that are too large to be represented. Here are some floating-point literals:

```
1.23   .23   0.23   1.   1.0   1.2e10   1.23e-15
```

Note that a space cannot occur in the middle of a floating-point literal. For example, ***65.43 e-21*** is not a floating-point literal but rather four separate lexical tokens (causing a syntax error):

```
65.43   e   -   21
```

If you want a floating-point literal of type ***float***, you can define one using the suffix ***f*** or ***F***:

```
3.14159265f   2.0f   2.997925F
```

4.6 Sizes [dcl.size]

Some of the aspects of C++'s fundamental types, such as the size of an ***int***, are implementation-defined (§C.2). I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don't, they are forced to waste time finding and fixing obscure bugs. People who claim they don't care about portability usually do so because they use only a single system and feel they can afford the attitude that ''the language is what my compiler implements.'' This is a narrow and shortsighted view. If your program is a success, it is likely to be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favorite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies

when a program is written than to try to untangle the mess afterwards.

It is relatively easy to limit the impact of implementation-dependent language features. Limiting the impact of system-dependent library facilities is far harder. Using standard library facilities wherever feasible is one approach.

The reason for providing more than one integer type, more than one unsigned type, and more than one floating-point type is to allow the programmer to take advantage of hardware characteristics. On many machines, there are significant differences in memory requirements, memory access times, and computation speed between the different varieties of fundamental types. If you know a machine, it is usually easy to choose, for example, the appropriate integer type for a particular variable. Writing truly portable low-level code is harder.

Sizes of C++ objects are expressed in terms of multiples of the size of a ***char***, so by definition the size of a ***char*** is ***1***. The size of an object or type can be obtained using the ***sizeof*** operator (§6.2). This is what is guaranteed about sizes of fundamental types:

$$1 \equiv sizeof(char) \leq sizeof(short) \leq sizeof(int) \leq sizeof(long)$$

$$1 \leq sizeof(bool) \leq sizeof(long)$$

$$sizeof(char) \leq sizeof(wchar_t) \leq sizeof(long)$$

$$sizeof(float) \leq sizeof(double) \leq sizeof(long\ double)$$

$$sizeof(N) \equiv sizeof(signed\ N) \equiv sizeof(unsigned\ N)$$

where ***N*** can be ***char***, ***short int***, ***int***, or ***long int***. In addition, it is guaranteed that a ***char*** has at least 8 bits, a ***short*** at least 16 bits, and a ***long*** at least 32 bits. A ***char*** can hold a character of the machine's character set.

Here is a graphical representation of a plausible set of fundamental types and a sample string:

On the same scale (.2 inch to a byte), a megabyte of memory would stretch about three miles (five km) to the right.

The *char* type is supposed to be chosen by the implementation to be the most suitable type for holding and manipulating characters on a given computer; it is typically an 8-bit byte. Similarly, the *int* type is supposed to be chosen to be the most suitable for holding and manipulating integers on a given computer; it is typically a 4-byte (32-bit) word. It is unwise to assume more. For example, there are machines with 32 bit *char*s.

When needed, implementation-dependent aspects about an implementation can be found in *<limits>* (§22.2). For example:

```
#include <limits>

int main()
{
        cout << "largest float == " << numeric_limits<float>::max()
                << ", char is signed == " << numeric_limits<char>::is_signed << '\n';
}
```

The fundamental types can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information (§C.6).

If a value *v* can be represented exactly in a variable of type *T*, a conversion of *v* to *T* is value-preserving and no problem. The cases where conversions are not value-preserving are best avoided (§C.6.2.6).

You need to understand implicit conversion in some detail in order to complete a major project and especially to understand real code written by others. However, such understanding is not required to read the following chapters.

4.7 Void [dcl.void]

The type *void* is syntactically a fundamental type. It can, however, be used only as part of a more complicated type; there are no objects of type *void*. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. For example:

```
void x;          // error: there are no void objects
void f();        // function f does not return a value (§7.3)
void* pv;        // pointer to object of unknown type (§5.6)
```

When declaring a function, you must specify the type of the value returned. Logically, you would expect to be able to indicate that a function didn't return a value by omitting the return type. However, that would make the grammar (Appendix A) less regular and clash with C usage. Consequently, *void* is used as a ''pseudo return type'' to indicate that a function doesn't return a value.

4.8 Enumerations [dcl.enum]

An *enumeration* is a type that can hold a set of values specified by the user. Once defined, an enumeration is used very much like an integer type.

Named integer constants can be defined as members of an enumeration. For example,

```
enum { ASM, AUTO, BREAK };
```

defines three integer constants, called enumerators, and assigns values to them. By default, enumerator values are assigned increasing from ***0***, so ***ASM==0***, ***AUTO==1***, and ***BREAK==2***. An enumeration can be named. For example:

```
enum keyword { ASM, AUTO, BREAK };
```

Each enumeration is a distinct type. The type of an enumerator is its enumeration. For example, ***AUTO*** is of type ***keyword***.

Declaring a variable ***keyword*** instead of plain ***int*** can give both the user and the compiler a hint as to the intended use. For example:

```
void f(keyword key)
{
    switch (key) {
    case ASM:
        // do something
        break;
    case BREAK:
        // do something
        break;
    }
}
```

A compiler can issue a warning because only two out of three ***keyword*** values are handled.

An enumerator can be initialized by a *constant-expression* (§C.5) of integral type (§4.1.1). The range of an enumeration holds all the enumeration's enumerator values rounded up to the nearest larger binary power minus ***1***. The range goes down to ***0*** if the smallest enumerator is non-negative and to the nearest lesser negative binary power if the smallest enumerator is negative. This defines the smallest bit-field capable of holding the enumerator values. For example:

```
enum e1 { dark, light };                  // range 0:1
enum e2 { a = 3, b = 9 };                 // range 0:15
enum e3 { min = -10, max = 1000000 };     // range -1048576:1048575
```

A value of integral type may be explicitly converted to an enumeration type. The result of such a conversion is undefined unless the value is within the range of the enumeration. For example:

```
enum flag { x=1, y=2, z=4, e=8 };   // range 0:15

flag f1 = 5;          // type error: 5 is not of type flag
flag f2 = flag(5);    // ok: flag(5) is of type flag and within the range of flag

flag f3 = flag(z|e);  // ok: flag(12) is of type flag and within the range of flag
flag f4 = flag(99);   // undefined: 99 is not within the range of flag
```

The last assignment shows why there is no implicit conversion from an integer to an enumeration; most integer values do not have a representation in a particular enumeration.

The notion of a range of values for an enumeration differs from the enumeration notion in the Pascal family of languages. However, bit-manipulation examples that require values outside the set of enumerators to be well-defined have a long history in C and C++.

The *sizeof* an enumeration is the *sizeof* some integral type that can hold its range and not larger than *sizeof*(*int*), unless an enumerator cannot be represented as an *int* or as an *unsigned int*. For example, *sizeof*(*e1*) could be *1* or maybe *4* but not *8* on a machine where *sizeof*(*int*)==*4*.

By default, enumerations are converted to integers for arithmetic operations (§6.2). An enumeration is a user-defined type, so users can define their own operations, such as ++ and << for an enumeration (§11.2.3).

4.9 Declarations [dcl.dcl]

Before a name (identifier) can be used in a C++ program, it must be declared. That is, its type must be specified to inform the compiler to what kind of entity the name refers. Here are some examples illustrating the diversity of declarations:

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
extern int error_number;

char* name = "Njal";
char* season[] = { "spring", "summer", "fall", "winter" };

struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

As can be seen from these examples, a declaration can do more than simply associate a type with a name. Most of these *declarations* are also *definitions*; that is, they also define an entity for the name to which they refer. For ***ch***, that entity is the appropriate amount of memory to be used as a variable – that memory will be allocated. For ***day***, it is the specified function. For the constant ***pi***, it is the value *3.1415926535897932385*. For ***Date***, that entity is a new type. For ***Point***, it is the type ***complex<short>*** so that ***Point*** becomes a synonym for ***complex<short>***. Of the declarations above, only

```
double sqrt(double);
extern int error_number;
struct User;
```

are not also definitions; that is, the entity they refer to must be defined elsewhere. The code (body) for the function ***sqrt*** must be specified by some other declaration, the memory for the ***int*** variable ***error_number*** must be allocated by some other declaration of ***error_number***, and some other declaration of the type ***User*** must define what that type looks like. For example:

```
double sqrt(double d) { /* ... */ }
int error_number = 1;

struct User { /* ... */ };
```

There must always be exactly one definition for each name in a C++ program (for the effects of *#include*, see §9.2.3). However, there can be many declarations. All declarations of an entity must agree on the type of the entity referred to. So, this fragment has two errors:

```
int count;
int count; // error: redefinition

extern int error_number;
extern short error_number;      // error: type mismatch
```

and this has none (for the use of *extern* see §9.2):

```
extern int error_number;
extern int error_number;
```

Some definitions specify a ‘‘value’’ for the entities they define. For example:

```
struct Date { int d, m, y; };
typedef complex<short> Point;
int day(Date* p) { return p->d; }
const double pi = 3.1415926535897932385;
```

For types, templates, functions, and constants, the ‘‘value’’ is permanent. For nonconstant data types, the initial value may be changed later. For example:

```
void f()
{
    int count = 1;
    char* name = "Bjarne";
    // ...
    count = 2;
    name = "Marian";
}
```

Of the definitions, only

```
char ch;
string s;
```

do not specify values. See §4.9.5 and §10.4.2 for explanations of how and when a variable is assigned a default value. Any declaration that specifies a value is a definition.

4.9.1 The Structure of a Declaration [dcl.parts]

A declaration consists of four parts: an optional ‘‘specifier,’’ a base type, a declarator, and an optional initializer. Except for function and namespace definitions, a declaration is terminated by a semicolon. For example:

```
char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };
```

Here, the base type is ***char***, the declarator is ****kings[]***, and the initializer is **={...}**.

A specifier is an initial keyword, such as ***virtual*** (§2.5.5, §12.2.6) and ***extern*** (§9.2), that specifies some non-type attribute of what is being declared.

A declarator is composed of a name and optionally some declarator operators. The most common declarator operators are (§A.7.1):

*	*pointer*	*prefix*
*const	*constant pointer*	*prefix*
&	*reference*	*prefix*
[]	*array*	*postfix*
( )	*function*	*postfix*

Their use would be simple if they were all either prefix or postfix. However, *****, **[]**, and **( )** were designed to mirror their use in expressions (§6.2). Thus, ***** is prefix and **[]** and **( )** are postfix. The postfix declarator operators bind tighter than the prefix ones. Consequently, ****kings[]*** is a vector of pointers to something, and we have to use parentheses to express types such as ‘‘pointer to function;’’ see examples in §5.1. For full details, see the grammar in Appendix A.

Note that the type cannot be left out of a declaration. For example:

```
const c = 7;      // error: no type
gt(int a, int b) { return (a>b) ? a : b; } // error: no return type

unsigned ui;      // ok: 'unsigned' is the type 'unsigned int'
long li;          // ok: 'long' is the type 'long int'
```

In this, standard C++ differs from earlier versions of C and C++ that allowed the first two examples by considering ***int*** to be the type when none were specified (§B.2). This ‘‘implicit ***int***’’ rule was a source of subtle errors and confusion.

4.9.2 Declaring Multiple Names [dcl.multi]

It is possible to declare several names in a single declaration. The declaration simply contains a list of comma-separated declarators. For example, we can declare two integers like this:

```
int x, y;         // int x; int y;
```

Note that operators apply to individual names only – and not to any subsequent names in the same declaration. For example:

```
int* p, y;        // int* p;   int y; NOT int* y;
int x, *q;        // int x;    int* q;
int v[10], *pv;   // int v[10];     int* pv;
```

Such constructs make a program less readable and should be avoided.

4.9.3 Names [dcl.name]

A name (identifier) consists of a sequence of letters and digits. The first character must be a letter. The underscore character _ is considered a letter. C++ imposes no limit on the number of characters in a name. However, some parts of an implementation are not under the control of the compiler writer (in particular, the linker), and those parts, unfortunately, sometimes do impose limits. Some run-time environments also make it necessary to extend or restrict the set of characters accepted in an identifier. Extensions (e.g., allowing the character $ in a name) yield nonportable programs. A C++ keyword (Appendix A), such as ***new*** and ***int***, cannot be used as a name of a user-defined entity. Examples of names are:

```
hello          this_is_a_most_unusually_long_name
DEFINED        foO        bAr        u_name        HorseSense
var0           var1       CLASS      _class        ___
```

Examples of character sequences that cannot be used as identifiers are:

```
012            a fool     $sys       class         3var
pay.due        foo~bar    .name      if
```

Names starting with an underscore are reserved for special facilities in the implementation and the run-time environment, so such names should not be used in application programs.

When reading a program, the compiler always looks for the longest string of characters that could make up a name. Hence, ***var10*** is a single name, not the name ***var*** followed by the number ***10***. Also, ***elseif*** is a single name, not the keyword ***else*** followed by the keyword ***if***.

Uppercase and lowercase letters are distinct, so ***Count*** and ***count*** are different names, but it is unwise to choose names that differ only by capitalization. In general, it is best to avoid names that differ only in subtle ways. For example, the uppercase o (***O***) and zero (***0***) can be hard to tell apart, as can the lowercase L (***l***) and one (***1***). Consequently, ***l0***, ***lO***, ***l1***, and ***ll*** are poor choices for identifier names.

Names from a large scope ought to have relatively long and reasonably obvious names, such as ***vector***, ***Window_with_border***, and ***Department_number***. However, code is clearer if names used only in a small scope have short, conventional names such as ***x***, ***i***, and ***p***. Classes (Chapter 10) and namespaces (§8.2) can be used to keep scopes small. It is often useful to keep frequently used names relatively short and reserve really long names for infrequently used entities. Choose names to reflect the meaning of an entity rather than its implementation. For example, ***phone_book*** is better than ***number_list*** even if the phone numbers happen to be stored in a ***list*** (§3.7). Choosing good names is an art.

Try to maintain a consistent naming style. For example, capitalize nonstandard library user-defined types and start nontypes with a lowercase letter (for example, ***Shape*** and ***current_token***). Also, use all capitals for macros (if you must use macros; for example, ***HACK***) and use underscores to separate words in an identifier. However, consistency is hard to achieve because programs are typically composed of fragments from different sources and several different reasonable styles are in use. Be consistent in your use of abbreviations and acronyms.

4.9.4 Scope [dcl.scope]

A declaration introduces a name into a scope; that is, a name can be used only in a specific part of the program text. For a name declared in a function (often called a *local name*), that scope extends from its point of declaration to the end of the block in which its declaration occurs. A *block* is a section of code delimited by a { } pair.

A name is called *global* if it is defined outside any function, class (Chapter 10), or namespace (§8.2). The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs. A declaration of a name in a block can hide a declaration in an enclosing block or a global name. That is, a name can be redefined to refer to a different entity within a block. After exit from the block, the name resumes its previous meaning. For example:

```
int x;                  // global x

void f()
{
    int x;              // local x hides global x
    x = 1;              // assign to local x

    {
        int x;          // hides first local x
        x = 2;          // assign to second local x
    }

    x = 3;              // assign to first local x
}

int* p = &x;            // take address of global x
```

Hiding names is unavoidable when writing large programs. However, a human reader can easily fail to notice that a name has been hidden. Because such errors are relatively rare, they can be very difficult to find. Consequently, name hiding should be minimized. Using names such as ***i*** and ***x*** for global variables or for local variables in a large function is asking for trouble.

A hidden global name can be referred to using the scope resolution operator ::. For example:

```
int x;

void f2()
{
    int x = 1;  // hide global x
    ::x = 2;    // assign to global x
    x = 2;      // assign to local x
    // ...
}
```

There is no way to use a hidden local name.

The scope of a name starts at its point of declaration; that is, after the complete declarator and before the initializer. This implies that a name can be used even to specify its own initial value. For example:

```
int x;
```

```
void f3()
{
    int x = x; // perverse: initialize x with its own (uninitialized) value
}
```

This is not illegal, just silly. A good compiler will warn if a variable is used before it has been set (see also §5.9[9]).

It is possible to use a single name to refer to two different objects in a block without using the `::` operator. For example:

```
int x = 11;

void f4()               // perverse:
{
    int y = x;          // use global x: y = 11
    int x = 22;
    y = x;              // use local x: y = 22
}
```

Function argument names are considered declared in the outermost block of a function, so

```
void f5(int x)
{
    int x;      // error
}
```

is an error because `x` is defined twice in the same scope. Having this be an error allows a not uncommon, subtle mistake to be caught.

4.9.5 Initialization [dcl.init]

If an initializer is specified for an object, that initializer determines the initial value of an object. If no initializer is specified, a global (§4.9.4), namespace (§8.2), or local static object (§7.1.2, §10.2.4) (collectively called *static objects*) is initialized to `0` of the appropriate type. For example:

```
int a;          // means ''int a = 0;''
double d;       // means ''double d = 0.0;''
```

Local variables (sometimes called *automatic objects*) and objects created on the free store (sometimes called *dynamic objects* or *heap objects*) are not initialized by default. For example:

```
void f()
{
    int x;      // x does not have a well-defined value
    // ...
}
```

Members of arrays and structures are default initialized or not depending on whether the array or structure is static. User-defined types may have default initialization defined (§10.4.2).

More complicated objects require more than one value as an initializer. This is handled by initializer lists delimited by `{` and `}` for C-style initialization of arrays (§5.2.1) and structures (§5.7). For user-defined types with constructors, function-style argument lists are used (§2.5.2, §10.2.3).

Note that an empty pair of parentheses `()` in a declaration always means ''function'' (§7.1). For example:

```
int a[] = { 1, 2 };          // array initializer
Point z(1,2);                // function-style initializer (initialization by constructor)
int f();                     // function declaration
```

4.9.6 Objects and Lvalues [dcl.objects]

We can allocate and use ''variables'' that do not have names, and it is possible to assign to strange-looking expressions (e.g., `*p[a+10]=7`). Consequently, there is a need for a name for ''something in memory.'' This is the simplest and most fundamental notion of an object. That is, an *object* is a contiguous region of storage; an *lvalue* is an expression that refers to an object. The word *lvalue* was originally coined to mean ''something that can be on the left-hand side of an assignment.'' However, not every lvalue may be used on the left-hand side of an assignment; an lvalue can refer to a constant (§5.5). An lvalue that has not been declared ***const*** is often called a *modifiable lvalue*. This simple and low-level notion of an object should not be confused with the notions of class object and object of polymorphic type (§15.4.3).

Unless the programmer specifies otherwise (§7.1.2, §10.4.8), an object declared in a function is created when its definition is encountered and destroyed when its name goes out of scope (§10.4.4). Such objects are called automatic objects. Objects declared in global or namespace scope and ***static***s declared in functions or classes are created and initialized once (only) and ''live'' until the program terminates (§10.4.9). Such objects are called static objects. Array elements and nonstatic structure or class members have their lifetimes determined by the object of which they are part.

Using the ***new*** and ***delete*** operators, you can create objects whose lifetimes are controlled directly (§6.2.6).

4.9.7 Typedef [dcl.typedef]

A declaration prefixed by the keyword ***typedef*** declares a new name for the type rather than a new variable of the given type. For example:

```
typedef char* Pchar;
Pchar p1, p2;        // p1 and p2 are char*s
char* p3 = p1;
```

A name defined like this, usually called a '' ***typedef***,'' can be a convenient shorthand for a type with an unwieldy name. For example, ***unsigned char*** is too long for really frequent use, so we could define a synonym, ***uchar***:

```
typedef unsigned char uchar;
```

Another use of a ***typedef*** is to limit the direct reference to a type to one place. For example:

```
typedef int int32;
typedef short int16;
```

If we now use ***int32*** wherever we need a potentially large integer, we can port our program to a machine on which ***sizeof(int)*** is ***2*** by redefining the single occurrence of ***int*** in our code:

```
typedef long int32;
```

For good and bad, ***typedef***s are synonyms for other types rather than distinct types. Consequently, ***typedef***s mix freely with the types for which they are synonyms. People who would like to have distinct types with identical semantics or identical representation should look at enumerations (§4.8) or classes (Chapter 10).

4.10 Advice [dcl.advice]

[1] Keep scopes small; §4.9.4.
[2] Don't use the same name in both a scope and an enclosing scope; §4.9.4.
[3] Declare one name (only) per declaration; §4.9.2.
[4] Keep common and local names short, and keep uncommon and nonlocal names longer; §4.9.3.
[5] Avoid similar-looking names; §4.9.3.
[6] Maintain a consistent naming style; §4.9.3.
[7] Choose names carefully to reflect meaning rather than implementation; §4.9.3.
[8] Use a ***typedef*** to define a meaningful name for a built-in type in cases in which the built-in type used to represent a value might change; §4.9.7.
[9] Use ***typedef***s to define synonyms for types; use enumerations and classes to define new types; §4.9.7.
[10] Remember that every declaration must specify a type (there is no ''implicit ***int***''); §4.9.1.
[11] Avoid unnecessary assumptions about the numeric value of characters; §4.3.1, §C.6.2.1.
[12] Avoid unnecessary assumptions about the size of integers; §4.6.
[13] Avoid unnecessary assumptions about the range of floating-point types; §4.6.
[14] Prefer a plain ***int*** over a ***short int*** or a ***long int***; §4.6.
[15] Prefer a ***double*** over a ***float*** or a ***long double***; §4.5.
[16] Prefer plain ***char*** over ***signed char*** and ***unsigned char***; §C.3.4.
[17] Avoid making unnecessary assumptions about the sizes of objects; §4.6.
[18] Avoid unsigned arithmetic; §4.4.
[19] View ***signed*** to ***unsigned*** and ***unsigned*** to ***signed*** conversions with suspicion; §C.6.2.6.
[20] View floating-point to integer conversions with suspicion; §C.6.2.6.
[21] View conversions to a smaller type, such as ***int*** to ***char***, with suspicion; §C.6.2.6.

4.11 Exercises [dcl.exercises]

1. (∗2) Get the ''Hello, world!'' program (§3.2) to run. If that program doesn't compile as written, look at §B.3.1.
2. (∗1) For each declaration in §4.9, do the following: If the declaration is not a definition, write a definition for it. If the declaration is a definition, write a declaration for it that is not also a definition.
3. (∗1.5) Write a program that prints the sizes of the fundamental types, a few pointer types, and a few enumerations of your choice. Use the ***sizeof*** operator.

4. (∗1.5) Write a program that prints out the letters ´***a***´..´***z***´ and the digits ´***0***´..´***9***´ and their integer values. Do the same for other printable characters. Do the same again but use hexadecimal notation.
5. (∗2) What, on your system, are the largest and the smallest values of the following types: ***char***, ***short***, ***int***, ***long***, ***float***, ***double***, ***long double***, and ***unsigned***.
6. (∗1) What is the longest local name you can use in a C++ program on your system? What is the longest external name you can use in a C++ program on your system? Are there any restrictions on the characters you can use in a name?
7. (∗2) Draw a graph of the integer and fundamental types where a type points to another type if all values of the first can be represented as values of the second on every standards-conforming implementation. Draw the same graph for the types on your favorite implementation.

5

Pointers, Arrays, and Structures

The sublime and the ridiculous
are often so nearly related that
it is difficult to class them separately.
– Tom Paine

Pointers — zero — arrays — string literals — pointers into arrays — constants — pointers and constants — references — ***void**** — data structures — advice — exercises.

5.1 Pointers [ptr.ptr]

For a type ***T***, ***T**** is the type ‘‘pointer to ***T***.’’ That is, a variable of type ***T**** can hold the address of an object of type ***T***. For example:

```
char c = 'a';
char* p = &c;          // p holds the address of c
```

or graphically:

Unfortunately, pointers to arrays and pointers to functions need a more complicated notation:

```
int* pi;               // pointer to int
char** ppc;            // pointer to pointer to char
int* ap[15];           // array of 15 pointers to ints
int (*fp)(char*);      // pointer to function taking a char* argument; returns an int
int* f(char*);         // function taking a char* argument; returns a pointer to int
```

See §4.9.1 for an explanation of the declaration syntax and Appendix A for the complete grammar.

The fundamental operation on a pointer is *dereferencing*, that is, referring to the object pointed to by the pointer. This operation is also called *indirection*. The dereferencing operator is (prefix) unary `*`. For example:

```
char c = 'a';
char* p = &c;    // p holds the address of c
char c2 = *p;    // c2 == 'a'
```

The variable pointed to by ***p*** is ***c***, and the value stored in ***c*** is ***'a'***, so the value of ****p*** assigned to ***c2*** is ***'a'***.

It is possible to perform some arithmetic operations on pointers to array elements (§5.3). Pointers to functions can be extremely useful; they are discussed in §7.7.

The implementation of pointers is intended to map directly to the addressing mechanisms of the machine on which the program runs. Most machines can address a byte. Those that can't tend to have hardware to extract bytes from words. On the other hand, few machines can directly address an individual bit. Consequently, the smallest object that can be independently allocated and pointed to using a built-in pointer type is a ***char***. Note that a ***bool*** occupies at least as much space as a ***char*** (§4.6). To store smaller values more compactly, you can use logical operations (§6.2.4) or bit fields in structures (§C.8.1).

5.1.1 Zero [ptr.zero]

Zero (***0***) is an ***int***. Because of standard conversions (§C.6.2.3), ***0*** can be used as a constant of any integral (§4.1.1), floating-point, pointer, or pointer-to-member type. The type of zero will be determined by context. Zero will typically (but not necessarily) be represented by the bit pattern *all-zeros* of the appropriate size.

No object is allocated with the address ***0***. Consequently, ***0*** acts as a pointer literal, indicating that a pointer doesn't refer to an object.

In C, it has been popular to define a macro ***NULL*** to represent the zero pointer. Because of C++'s tighter type checking, the use of plain ***0***, rather than any suggested ***NULL*** macro, leads to fewer problems. If you feel you must define ***NULL***, use

```
const int NULL = 0;
```

The ***const*** qualifier (§5.4) prevents accidental redefinition of ***NULL*** and ensures that ***NULL*** can be used where a constant is required.

5.2 Arrays [ptr.array]

For a type ***T***, ***T[size]*** is the type ''array of ***size*** elements of type ***T***.'' The elements are indexed from ***0*** to ***size-1***. For example:

```
float v[3];      // an array of three floats: v[0], v[1], v[2]
char* a[32];     // an array of 32 pointers to char: a[0] .. a[31]
```

The number of elements of the array, the array bound, must be a constant expression (§C.5). If you need variable bounds, use a ***vector*** (§3.7.1, §16.3). For example:

```
void f(int i)
{
    int v1[i];              // error: array size not a constant expression
    vector<int> v2(i);      // ok
}
```

Multidimensional arrays are represented as arrays of arrays. For example:

```
int d2[10][20]; // d2 is an array of 10 arrays of 20 integers
```

Using comma notation as used for array bounds in some other languages gives compile-time errors because comma (`,`) is a sequencing operator (§6.2.2) and is not allowed in constant expressions (§C.5). For example, try this:

```
int bad[5,2];       // error: comma not allowed in a constant expression
```

Multidimensional arrays are described in §C.7. They are best avoided outside low-level code.

5.2.1 Array Initializers [ptr.array.init]

An array can be initialized by a list of values. For example:

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
```

When an array is declared without a specific size, but with an initializer list, the size is calculated by counting the elements of the initializer list. Consequently, ***v1*** and ***v2*** are of type ***int[4]*** and ***char[4]***, respectively. If a size is explicitly specified, it is an error to give surplus elements in an initializer list. For example:

```
char v3[2] = { 'a', 'b', 0 };       // error: too many initializers
char v4[3] = { 'a', 'b', 0 };       // ok
```

If the initializer supplies too few elements, ***0*** is assumed for the remaining array elements. For example:

```
int v5[8] = { 1, 2, 3, 4 };
```

is equivalent to

```
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 };
```

Note that there is no array assignment to match the initialization:

```
void f()
{
    v4 = { 'c', 'd', 0 }; // error: no array assignment
}
```

When you need such assignments, use a ***vector*** (§16.3) or a ***valarray*** (§22.4) instead.

An array of characters can be conveniently initialized by a string literal (§5.2.2).

5.2.2 String Literals [ptr.string.literal]

A *string literal* is a character sequence enclosed within double quotes:

```
"this is a string"
```

A string literal contains one more character than it appears to have; it is terminated by the null character ´\0´, with the value 0. For example:

```
sizeof("Bohr")==5
```

The type of a string literal is ‘‘array of the appropriate number of **const** characters,’’ so **"Bohr"** is of type **const char[5]**.

A string literal can be assigned to a **char***. This is allowed because in previous definitions of C and C++ , the type of a string literal was **char***. Allowing the assignment of a string literal to a **char*** ensures that millions of lines of C and C++ remain valid. It is, however, an error to try to modify a string literal through such a pointer:

```
void f()
{
    char* p = "Plato";
    p[4] = ´e´;          // error: assignment to const; result is undefined
}
```

This kind of error cannot in general be caught until run-time, and implementations differ in their enforcement of this rule. Having string literals constant not only is obvious, but also allows implementations to do significant optimizations in the way string literals are stored and accessed.

If we want a string that we are guaranteed to be able to modify, we must copy the characters into an array:

```
void f()
{
    char p[] = "Zeno";      // p is an array of 5 char
    p[0] = ´R´;             // ok
}
```

A string literal is statically allocated so that it is safe to return one from a function. For example:

```
const char* error_message(int i)
{
    // ...
    return "range error";
}
```

The memory holding **range error** will not go away after a call of **error_message()**.

Whether two identical character literals are allocated as one is implementation-defined (§C.1). For example:

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";
```

```
void g()
{
    if (p == q) cout << "one!\n";   // result is implementation-defined
    // ...
}
```

Note that == compares addresses (pointer values) when applied to pointers, and not the values pointed to.

The empty string is written as a pair of adjacent double quotes, `""`, (and has the type *const char[1]*).

The backslash convention for representing nongraphic characters (§C.3.2) can also be used within a string. This makes it possible to represent the double quote (") and the escape character backslash (\) within a string. The most common such character by far is the newline character, ´*\n*´. For example:

```
cout<<"beep at end of message\a\n";
```

The escape character ´*\a*´ is the ASCII character ***BEL*** (also known as *alert*), which causes some kind of sound to be emitted.

It is not possible to have a ‘‘real’’ newline in a string:

```
"this is not a string
but a syntax error"
```

Long strings can be broken by whitespace to make the program text neater. For example:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

The compiler will concatenate adjacent strings, so ***alpha*** could equivalently have been initialized by the single string:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

It is possible to have the null character in a string, but most programs will not suspect that there are characters after it. For example, the string ***"Jens\000Munk"*** will be treated as ***"Jens"*** by standard library functions such as ***strcpy()*** and ***strlen()***; see §20.4.1.

A string with the prefix ***L***, such as ***L"angst"***, is a string of wide characters (§4.3, §C.3.3). Its type is *const wchar_t[]*.

5.3 Pointers into Arrays [ptr.into]

In C++, pointers and arrays are closely related. The name of an array can be used as a pointer to its initial element. For example:

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;          // pointer to initial element (implicit conversion)
int* p2 = &v[0];      // pointer to initial element
int* p3 = &v[4];      // pointer to one beyond last element
```

or graphically:

Taking a pointer to the element one beyond the end of an array is guaranteed to work. This is important for many algorithms (§2.7.2, §18.3). However, since such a pointer does not in fact point to an element of the array, it may not be used for reading or writing. The result of taking the address of the element before the initial element is undefined and should be avoided. On some machine architectures, arrays are often allocated on machine addressing boundaries, so ‘‘one before the initial element’’ simply doesn’t make sense.

The implicit conversion of an array name to a pointer to the initial element of the array is extensively used in function calls in C-style code. For example:

```
extern "C" int strlen(const char*); // from <string.h>

void f()
{
    char v[] = "Annemarie";
    char* p = v;      // implicit conversion of char[] to char*
    strlen(p);
    strlen(v);        // implicit conversion of char[] to char*
    v = p;            // error: cannot assign to array
}
```

The same value is passed to the standard library function *strlen*( ) in both calls. The snag is that it is impossible to avoid the implicit conversion. In other words, there is no way of declaring a function so that the array *v* is copied when the function is called. Fortunately, there is no implicit or explicit conversion from a pointer to an array.

The implicit conversion of the array argument to a pointer means that the size of the array is lost to the called function. However, the called function must somehow determine the size to perform a meaningful operation. Like other C standard library functions taking pointers to characters, *strlen*( ) relies on zero to indicate end-of-string; *strlen*(*p*) returns the number of characters up to and not including the terminating *0*. This is all pretty low-level. The standard library *vector* (§16.3) and *string* (Chapter 20) don’t suffer from this problem.

5.3.1 Navigating Arrays [ptr.navigate]

Efficient and elegant access to arrays (and similar data structures) is the key to many algorithms (see §3.8, Chapter 18). Access can be achieved either through a pointer to an array plus an index or through a pointer to an element. For example, traversing a character string using an index,

```
void fi(char v[])
{
    for (int i = 0; v[i]!=0; i++) use(v[i]);
}
```

is equivalent to a traversal using a pointer:

```
void fp(char v[])
{
    for (char* p = v; *p!=0; p++) use(*p);
}
```

The prefix `*` operator dereferences a pointer so that `*p` is the character pointed to by `p`, and `++` increments the pointer so that it refers to the next element of the array.

There is no inherent reason why one version should be faster than the other. With modern compilers, identical code should be generated for both examples (see §5.9[8]). Programmers can choose between the versions on logical and aesthetic grounds.

The result of applying the arithmetic operators `+`, `-`, `++`, or `--` to pointers depends on the type of the object pointed to. When an arithmetic operator is applied to a pointer `p` of type `T*`, `p` is assumed to point to an element of an array of objects of type `T`; `p+1` points to the next element of that array, and `p-1` points to the previous element. This implies that the integer value of `p+1` will be `sizeof(T)` larger than the integer value of `p`. For example, executing

```
#include <iostream>

int main ()
{
    int vi[10];
    short vs[10];

    std::cout << &vi[0] << ´ ´ << &vi[1] << ´\n´;
    std::cout << &vs[0] << ´ ´ << &vs[1] << ´\n´;
}
```

produced

```
0x7fffaef0  0x7fffaef4
0x7fffaedc  0x7fffaede
```

using a default hexadecimal notation for pointer values. This shows that on my implementation, `sizeof(short)` is `2` and `sizeof(int)` is `4`.

Subtraction of pointers is defined only when both pointers point to elements of the same array (although the language has no fast way of ensuring that is the case). When subtracting one pointer from another, the result is the number of array elements between the two pointers (an integer). One can add an integer to a pointer or subtract an integer from a pointer; in both cases, the result is a pointer value. If that value does not point to an element of the same array as the original pointer or one beyond, the result of using that value is undefined. For example:

```
void f()
{
    int v1[10];
    int v2[10];

    int i1 = &v1[5]-&v1[3];   // i1 = 2
    int i2 = &v1[5]-&v2[3];   // result undefined
```

```
        int* p1 = v2+2;             // p1 = &v2[2]
        int* p2 = v2-2;             // *p2 undefined
}
```

Complicated pointer arithmetic is usually unnecessary and often best avoided. Addition of pointers makes no sense and is not allowed.

Arrays are not self-describing because the number of elements of an array is not guaranteed to be stored with the array. This implies that to traverse an array that does not contain a terminator the way character strings do, we must somehow supply the number of elements. For example:

```
void fp(char v[], unsigned int size)
{
        for (int i=0; i<size; i++) use(v[i]);

        const int N = 7;
        char v2[N];
        for (int i=0; i<N; i++) use(v2[i]);

}
```

Note that most C++ implementations offer no range checking for arrays. This array concept is inherently low-level. A more advanced notion of arrays can be provided through the use of classes; see §3.7.1.

5.4 Constants [ptr.const]

C++ offers the concept of a user-defined constant, a ***const***, to express the notion that a value doesn't change directly. This is useful in several contexts. For example, many objects don't actually have their values changed after initialization, symbolic constants lead to more maintainable code than do literals embedded directly in code, pointers are often read through but never written through, and most function parameters are read but not written to.

The keyword ***const*** can be added to the declaration of an object to make the object declared a constant. Because it cannot be assigned to, a constant must be initialized. For example:

```
const int model = 90;               // model is a const
const int v[] = { 1, 2, 3, 4 };     // v[i] is a const
const int x;                        // error: no initializer
```

Declaring something ***const*** ensures that its value will not change within its scope:

```
void f( )
{
        model = 200;    // error
        v[2]++;         // error
}
```

Note that ***const*** modifies a type; that is, it restricts the ways in which an object can be used, rather than specifying how the constant is to be allocated. For example:

```
void g(const X* p)
{
    // can't modify *p here
}

void h()
{
    X val;      // val can be modified
    g(&val);
    // ...
}
```

Depending on how smart it is, a compiler can take advantage of an object being a constant in several ways. For example, the initializer for a constant is often (but not always) a constant expression (§C.5); if it is, it can be evaluated at compile time. Further, if the compiler knows every use of the *const*, it need not allocate space to hold it. For example:

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3);     // don't know the value of c3 at compile time
extern const int c4;        // don't know the value of c4 at compile time
const int* p = &c2;         // need to allocate space for c2
```

Given this, the compiler knows the values of *c1* and *c2* so that they can be used in constant expressions. Because the values of *c3* and *c4* are not known at compile time (using only the information available in this compilation unit; see §9.1), storage must be allocated for *c3* and *c4*. Because the address of *c2* is taken (and presumably used somewhere), storage must be allocated for *c2*. The simple and common case is the one in which the value of the constant is known at compile time and no storage needs to be allocated; *c1* is an example of that. The keyword *extern* indicates that *c4* is defined elsewhere (§9.2).

It is typically necessary to allocate store for an array of constants because the compiler cannot, in general, figure out which elements of the array are referred to in expressions. On many machines, however, efficiency improvements can be achieved even in this case by placing arrays of constants in read-only storage.

Common uses for *const*s are as array bounds and case labels. For example:

```
const int a = 42;
const int b = 99;
const int max = 128;

int v[max];

void f(int i)
{
    switch (i) {
    case a:
        // ...
```

```
        case b:
            // ...
        }
    }
```

Enumerators (§4.8) are often an alternative to *const*s in such cases.

The way *const* can be used with class member functions is discussed in §10.2.6 and §10.2.7.

Symbolic constants should be used systematically to avoid ''magic numbers'' in code. If a numeric constant, such as an array bound, is repeated in code, it becomes hard to revise that code because every occurrence of that constant must be changed to make a correct update. Using a symbolic constant instead localizes information. Usually, a numeric constant represents an assumption about the program. For example, *4* may represent the number of bytes in an integer, *128* the number of characters needed to buffer input, and *6.24* the exchange factor between Danish kroner and U.S. dollars. Left as numeric constants in the code, these values are hard for a maintainer to spot and understand. Often, such numeric values go unnoticed and become errors when a program is ported or when some other change violates the assumptions they represent. Representing assumptions as well-commented symbolic constants minimizes such maintenance problems.

5.4.1 Pointers and Constants [ptr.pc]

When using a pointer, two objects are involved: the pointer itself and the object pointed to. ''Prefixing'' a declaration of a pointer with *const* makes the object, but not the pointer, a constant. To declare a pointer itself, rather than the object pointed to, to be a constant, we use the declarator operator **const* instead of plain ***. For example:

```
void f1(char* p)
{
    char s[] = "Gorm";

    const char* pc = s;          // pointer to constant
    pc[3] = 'g';                 // error: pc points to constant
    pc = p;                      // ok

    char *const cp = s;          // constant pointer
    cp[3] = 'a';                 // ok
    cp = p;                      // error: cp is constant

    const char *const cpc = s;   // const pointer to const
    cpc[3] = 'a';                // error: cpc points to constant
    cpc = p;                     // error: cpc is constant
}
```

The declarator operator that makes a pointer constant is **const*. There is no *const** declarator operator, so a *const* appearing before the *** is taken to be part of the base type. For example:

```
char *const cp;      // const pointer to char
char const* pc;      // pointer to const char
const char* pc2;     // pointer to const char
```

Some people find it helpful to read such declarations right-to-left. For example, ''*cp* is a *const* pointer to a *char*'' and ''*pc2* is a pointer to a *char const*.''

An object that is a constant when accessed through one pointer may be variable when accessed in other ways. This is particularly useful for function arguments. By declaring a pointer argument *const*, the function is prohibited from modifying the object pointed to. For example:

```
char* strcpy(char* p, const char* q); // cannot modify *q
```

You can assign the address of a variable to a pointer to constant because no harm can come from that. However, the address of a constant cannot be assigned to an unrestricted pointer because this would allow the object's value to be changed. For example:

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c;    // ok
    const int* p2 = &a;    // ok
    int* p3 = &c;          // error: initialization of int* with const int*
    *p3 = 7;               // try to change the value of c
}
```

It is possible to explicitly remove the restrictions on a pointer to *const* by explicit type conversion (§10.2.7.1 and §15.4.2.1).

5.5 References [ptr.ref]

A *reference* is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators (Chapter 11) in particular. The notation *X*& means *reference to X*. For example:

```
void f()
{
    int i = 1;
    int& r = i;        // r and i now refer to the same int
    int x = r;         // x = 1

    r = 2;             // i = 2
}
```

To ensure that a reference is a name for something (that is, bound to an object), we must initialize the reference. For example:

```
int i = 1;
int& r1 = i;           // ok: r1 initialized
int& r2;               // error: initializer missing
extern int& r3;        // ok: r3 initialized elsewhere
```

Initialization of a reference is something quite different from assignment to it. Despite appearances, no operator operates on a reference. For example:

```
void g()
{
    int ii = 0;
    int& rr = ii;
    rr++;                    // ii is incremented to 1
    int* pp = &rr;           // pp points to ii
}
```

This is legal, but ***rr++*** does not increment the reference ***rr***; rather, ++ is applied to an ***int*** that happens to be ***ii***. Consequently, the value of a reference cannot be changed after initialization; it always refers to the object it was initialized to denote. To get a pointer to the object denoted by a reference ***rr***, we can write &***rr***.

The obvious implementation of a reference is as a (constant) pointer that is dereferenced each time it is used. It doesn't do much harm thinking about references that way, as long as one remembers that a reference isn't an object that can be manipulated the way a pointer is:

In some cases, the compiler can optimize away a reference so that there is no object representing that reference at run-time.

Initialization of a reference is trivial when the initializer is an lvalue (an object whose address you can take; see §4.9.6). The initializer for a ''plain'' ***T***& must be an lvalue of type ***T***.

The initializer for a ***const T***& need not be an lvalue or even of type ***T***. In such cases,

[1] first, implicit type conversion to ***T*** is applied if necessary (see §C.6);

[2] then, the resulting value is placed in a temporary variable of type ***T***; and

[3] finally, this temporary variable is used as the value of the initializer.

Consider:

```
double& dr = 1;          // error: lvalue needed
const double& cdr = 1;   // ok
```

The interpretation of this last initialization might be:

```
double temp = double(1);  // first create a temporary with the right value
const double& cdr = temp; // then use the temporary as the initializer for cdr
```

A temporary created to hold a reference initializer persists until the end of its reference's scope.

References to variables and references to constants are distinguished because the introduction of a temporary in the case of the variable is highly error-prone; an assignment to the variable would become an assignment to the – soon to disappear – temporary. No such problem exists for references to constants, and references to constants are often important as function arguments (§11.6).

A reference can be used to specify a function argument so that the function can change the value of an object passed to it. For example:

```
void increment(int& aa) { aa++; }

void f()
{
    int x = 1;
    increment(x);        // x = 2
}
```

The semantics of argument passing are defined to be those of initialization, so when called, ***increment***'s argument ***aa*** became another name for ***x***. To keep a program readable, it is often best to avoid functions that modify their arguments. Instead, you can return a value from the function explicitly or require a pointer argument:

```
int next(int p) { return p+1; }

void incr(int* p) { (*p)++; }

void g()
{
    int x = 1;
    increment(x);        // x = 2
    x = next(x);         // x = 3
    incr(&x);            // x = 4
}
```

The ***increment***(***x***) notation doesn't give a clue to the reader that ***x***'s value is being modified, the way ***x=next***(***x***) and ***incr***(&***x***) does. Consequently ''plain'' reference arguments should be used only where the name of the function gives a strong hint that the reference argument is modified.

References can also be used to define functions that can be used on both the left-hand and right-hand sides of an assignment. Again, many of the most interesting uses of this are found in the design of nontrivial user-defined types. As an example, let us define a simple associative array. First, we define struct ***Pair*** like this:

```
struct Pair {
    string name;
    double val;
};
```

The basic idea is that a ***string*** has a floating-point value associated with it. It is easy to define a function, ***value***(), that maintains a data structure consisting of one ***Pair*** for each different string that has been presented to it. To shorten the presentation, a very simple (and inefficient) implementation is used:

```
vector<Pair> pairs;

double& value(const string& s)
/*
    maintain a set of Pairs:
    search for s, return its value if found; otherwise make a new Pair and return the default value 0
*/
{
```

```
        for (int i = 0; i < pairs.size(); i++)
                if (s == pairs[i].name) return pairs[i].val;

        Pair p = { s, 0 };
        pairs.push_back(p); // add Pair at end (§3.7.3)

        return pairs[pairs.size()-1].val;
}
```

This function can be understood as an array of floating-point values indexed by character strings. For a given argument string, ***value*( )** finds the corresponding floating-point object (*not* the value of the corresponding floating-point object); it then returns a reference to it. For example:

```
int main() // count the number of occurrences of each word on input
{
        string buf;

        while (cin>>buf) value(buf)++;

        for (vector<Pair>::const_iterator p = pairs.begin(); p!=pairs.end(); ++p)
                cout << p->name << ": " << p->val << '\n';
}
```

Each time around, the ***while***-loop reads one word from the standard input stream ***cin*** into the string ***buf*** (§3.6) and then updates the counter associated with it. Finally, the resulting table of different words in the input, each with its number of occurrences, is printed. For example, given the input

```
aa bb bb aa aa bb aa aa
```

this program will produce:

```
aa: 5
bb: 3
```

It is easy to refine this into a proper associative array type by using a template class with the selection operator **[ ]** overloaded (§11.8). It is even easier just to use the standard library ***map*** (§17.4.1).

5.6 Pointer to Void [ptr.ptrtovoid]

A pointer of any type of object can be assigned to a variable of type ***void****, a ***void**** can be assigned to another ***void****, ***void****s can be compared for equality and inequality, and a ***void**** can be explicitly converted to another type. Other operations would be unsafe because the compiler cannot know what kind of object is really pointed to. Consequently, other operations result in compile-time errors. To use a ***void****, we must explicitly convert it to a pointer to a specific type. For example:

```
void f(int* pi)
{
        void* pv = pi;    // ok: implicit conversion of int* to void*
        *pv;              // error: can't dereference void*
        pv++;             // error: can't increment void* (the size of the object pointed to is unknown)
```

```
        int* pi2 = static_cast<int*>(pv);          // explicit conversion back to int*

        double* pd1 = pv;                          // error
        double* pd2 = pi;                          // error
        double* pd3 = static_cast<double*>(pv);    // unsafe
    }
```

In general, it is not safe to use a pointer that has been converted (''cast'') to a type that differs from the type the object pointed to. For example, a machine may assume that every ***double*** is allocated on an 8-byte boundary. If so, strange behavior could arise if ***pi*** pointed to an ***int*** that wasn't allocated that way. This form of explicit type conversion is inherently unsafe and ugly. Consequently, the notation used, ***static_cast***, was designed to be ugly.

The primary use for ***void**** is for passing pointers to functions that are not allowed to make assumptions about the type of the object and for returning untyped objects from functions. To use such an object, we must use explicit type conversion.

Functions using ***void**** pointers typically exist at the very lowest level of the system, where real hardware resources are manipulated. For example:

```
void* my_alloc(size_t n); // allocate n bytes from my special heap
```

Occurrences of ***void****s at higher levels of the system should be viewed with suspicion because they are likely indicators of design errors. Where used for optimization, ***void**** can be hidden behind a type-safe interface (§13.5, §24.4.2).

Pointers to functions (§7.7) and pointers to members (§15.5) cannot be assigned to ***void****s.

5.7 Structures [ptr.struct]

An array is an aggregate of elements of the same type. A ***struct*** is an aggregate of elements of (nearly) arbitrary types. For example:

```
struct address {
        char* name;          // "Jim Dandy"
        long int number;     // 61
        char* street;        // "South St"
        char* town;          // "New Providence"
        char state[2];       // 'N' 'J'
        long zip;            // 7974
};
```

This defines a new type called ***address*** consisting of the items you need in order to send mail to someone. Note the semicolon at the end. This is one of very few places in C++ where it is necessary to have a semicolon after a curly brace, so people are prone to forget it.

Variables of type ***address*** can be declared exactly as other variables, and the individual *members* can be accessed using the **.** (dot) operator. For example:

```
void f()
{
    address jd;
    jd.name = "Jim Dandy";
    jd.number = 61;
}
```

The notation used for initializing arrays can also be used for initializing variables of structure types. For example:

```
address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence", {´N´,´J´}, 7974
};
```

Using a constructor (§10.2.3) is usually better, however. Note that *jd.state* could not be initialized by the string *"NJ"*. Strings are terminated by the character *´\0´*. Hence, *"NJ"* has three characters – one more than will fit into *jd.state*.

Structure objects are often accessed through pointers using the `->` (structure pointer dereference) operator. For example:

```
void print_addr(address* p)
{
    cout << p->name << ´\n´
        << p->number << ´ ´ << p->street << ´\n´
        << p->town << ´\n´
        << p->state[0] << p->state[1] << ´ ´ << p->zip << ´\n´;
}
```

When *p* is a pointer, *p->m* is equivalent to `(*p).m`.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
address current;

address set_current(address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

Other plausible operations, such as comparison (`==` and `!=`), are not defined. However, the user can define such operators (Chapter 11).

The size of an object of a structure type is not necessarily the sum of the sizes of its members. This is because many machines require objects of certain types to be allocated on architecture-dependent boundaries or handle such objects much more efficiently if they are. For example, integers are often allocated on word boundaries. On such machines, objects are said to have to be *aligned* properly. This leads to ‘‘holes’’ in the structures. For example, on many machines,

sizeof(*address*) is *24*, and not *22* as might be expected. You can minimize wasted space by simply ordering members by size (largest member first). However, it is usually best to order members for readability and sort them by size only if there is a demonstrated need to optimize.

The name of a type becomes available for use immediately after it has been encountered and not just after the complete declaration has been seen. For example:

```
struct Link {
    Link* previous;
    Link* successor;
};
```

It is not possible to declare new objects of a structure type until the complete declaration has been seen. For example:

```
struct No_good {
    No_good member;    // error: recursive definition
};
```

This is an error because the compiler is not able to determine the size of ***No_good***. To allow two (or more) structure types to refer to each other, we can declare a name to be the name of a structure type. For example:

```
struct List;    // to be defined later

struct Link {
    Link* pre;
    Link* suc;
    List* member_of;
};

struct List {
    Link* head;
};
```

Without the first declaration of ***List***, use of ***List*** in the declaration of ***Link*** would have caused a syntax error.

The name of a structure type can be used before the type is defined as long as that use does not require the name of a member or the size of the structure to be known. For example:

```
class S;    // 'S' is the name of some type

extern S a;
S f();
void g(S);
S* h(S*);
```

However, many such declarations cannot be used unless the type ***S*** is defined:

```
void k(S* p)
{
    S a;            // error: S not defined; size needed to allocate
```

```
        f();            // error: S not defined; size needed to return value
        g(a);           // error: S not defined; size needed to pass argument
        p->m = 7;       // error: S not defined; member name not known

        S* q = h(p);    // ok: pointers can be allocated and passed
        q->m = 7;       // error: S not defined; member name not known
    }
```

A ***struct*** is a simple form of a ***class*** (Chapter 10).

For reasons that reach into the pre-history of C, it is possible to declare a ***struct*** and a non-structure with the same name in the same scope. For example:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

In that case, the plain name (***stat***) is the name of the non-structure, and the structure must be referred to with the prefix ***struct***. Similarly, the keywords ***class***, ***union*** (§C.8.2), and ***enum*** (§4.8) can be used as prefixes for disambiguation. However, it is best not to overload names to make that necessary.

5.7.1 Type Equivalence [ptr.equiv]

Two structures are different types even when they have the same members. For example,

```
struct S1 { int a; };
struct S2 { int a; };
```

are two different types, so

```
S1 x;
S2 y = x;  // error: type mismatch
```

Structure types are also different from fundamental types, so

```
S1 x;
int i = x;  // error: type mismatch
```

Every ***struct*** must have a unique definition in a program (§9.2.3).

5.8 Advice [ptr.advice]

[1] Avoid nontrivial pointer arithmetic; §5.3.
[2] Take care not to write beyond the bounds of an array; §5.3.1.
[3] Use ***0*** rather than ***NULL***; §5.1.1.
[4] Use ***vector*** and ***valarray*** rather than built-in (C-style) arrays; §5.3.1.
[5] Use ***string*** rather than zero-terminated arrays of ***char***; §5.3.
[6] Minimize use of plain reference arguments; §5.5.
[7] Avoid ***void**** except in low-level code; §5.6.
[8] Avoid nontrivial literals (‘‘magic numbers’’) in code. Instead, define and use symbolic constants; §4.8, §5.4.

5.9 Exercises [ptr.exercises]

1. (∗1) Write declarations for the following: a pointer to a character, an array of 10 integers, a reference to an array of 10 integers, a pointer to an array of character strings, a pointer to a pointer to a character, a constant integer, a pointer to a constant integer, and a constant pointer to an integer. Initialize each one.
2. (∗1.5) What, on your system, are the restrictions on the pointer types ***char****, ***int****, and ***void****? For example, may an ***int**** have an odd value? Hint: alignment.
3. (∗1) Use ***typedef*** to define the types ***unsigned char***, ***const unsigned char***, pointer to integer, pointer to pointer to ***char***, pointer to arrays of ***char***, array of 7 pointers to ***int***, pointer to an array of 7 pointers to ***int***, and array of 8 arrays of 7 pointers to ***int***.
4. (∗1) Write a function that swaps (exchanges the values of) two integers. Use ***int**** as the argument type. Write another swap function using ***int&*** as the argument type.
5. (∗1.5) What is the size of the array ***str*** in the following example:

```
char str[] = "a short string";
```

   What is the length of the string ***"a short string"***?
6. (∗1) Define functions ***f(char)***, ***g(char&)***, and ***h(const char&)***. Call them with the arguments ***´a´***, ***49***, ***3300***, ***c***, ***uc***, and ***sc***, where ***c*** is a ***char***, ***uc*** is an ***unsigned char***, and ***sc*** is a ***signed char***. Which calls are legal? Which calls cause the compiler to introduce a temporary variable?
7. (∗1.5) Define a table of the names of months of the year and the number of days in each month. Write out that table. Do this twice; once using an array of ***char*** for the names and an array for the number of days and once using an array of structures, with each structure holding the name of a month and the number of days in it.
8. (∗2) Run some tests to see if your compiler really generates equivalent code for iteration using pointers and iteration using indexing (§5.3.1). If different degrees of optimization can be requested, see if and how that affects the quality of the generated code.
9. (∗1.5) Find an example where it would make sense to use a name in its own initializer.
10. (∗1) Define an array of strings in which the strings contain the names of the months. Print those strings. Pass the array to a function that prints those strings.
11. (∗2) Read a sequence of words from input. Use ***Quit*** as a word that terminates the input. Print the words in the order they were entered. Don't print a word twice. Modify the program to sort the words before printing them.
12. (∗2) Write a function that counts the number of occurrences of a pair of letters in a ***string*** and another that does the same in a zero-terminated array of ***char*** (a C-style string). For example, the pair "ab" appears twice in "xabaacbaxabb".
13. (∗1.5) Define a ***struct Date*** to keep track of dates. Provide functions that read ***Date***s from input, write ***Date***s to output, and initialize a ***Date*** with a date.

.

6

Expressions and Statements

Premature optimization
is the root of all evil.
– D. Knuth

On the other hand,
we cannot ignore efficiency.
– Jon Bentley

Desk calculator example — input — command line arguments — expression summary — logical and relational operators — increment and decrement — free store — explicit type conversion — statement summary — declarations — selection statements — declarations in conditions — iteration statements — the infamous ***goto*** — comments and indentation — advice — exercises.

6.1 A Desk Calculator [expr.calculator]

Statements and expressions are introduced by presenting a desk calculator program that provides the four standard arithmetic operations as infix operators on floating-point numbers. The user can also define variables. For example, given the input

```
r = 2.5
area = pi * r * r
```

(pi is predefined) the calculator program will write

```
2.5
19.635
```

where `2.5` is the result of the first line of input and `19.635` is the result of the second.

The calculator consists of four main parts: a parser, an input function, a symbol table, and a driver. Actually, it is a miniature compiler in which the parser does the syntactic analysis, the input function handles input and lexical analysis, the symbol table holds permanent information, and the driver handles initialization, output, and errors. We could add many features to this calculator to make it more useful (§6.6[20]), but the code is long enough as it is, and most features would just add code without providing additional insight into the use of C++.

6.1.1 The Parser [expr.parser]

Here is a grammar for the language accepted by the calculator:

```
program:
    END                              // END is end-of-input
    expr_list  END

expr_list:
    expression  PRINT                // PRINT is semicolon
    expression  PRINT  expr_list

expression:
    expression + term
    expression - term
    term

term:
    term / primary
    term * primary
    primary

primary:
    NUMBER
    NAME
    NAME = expression
    - primary
    ( expression )
```

In other words, a program is a sequence of expressions separated by semicolons. The basic units of an expression are numbers, names, and the operators `*`, `/`, `+`, `-` (both unary and binary), and `=`. Names need not be declared before use.

The style of syntax analysis used is usually called *recursive descent*; it is a popular and straightforward top-down technique. In a language such as C++, in which function calls are relatively cheap, it is also efficient. For each production in the grammar, there is a function that calls other functions. Terminal symbols (for example, ***END***, ***NUMBER***, `+`, and `-`) are recognized by the lexical analyzer, ***get_token***`()`; and nonterminal symbols are recognized by the syntax analyzer functions, ***expr***`()`, ***term***`()`, and ***prim***`()`. As soon as both operands of a (sub)expression are known, the expression is evaluated; in a real compiler, code could be generated at this point.

The parser uses a function ***get_token***`()` to get input. The value of the most recent call of ***get_token***`()` can be found in the global variable ***curr_tok***. The type of ***curr_tok*** is the enumeration ***Token_value***:

```
enum Token_value {
    NAME,       NUMBER,     END,
    PLUS='+',   MINUS='-',  MUL='*',    DIV='/',
    PRINT=';',  ASSIGN='=', LP='(',     RP=')'
};

Token_value curr_tok = PRINT;
```

Representing each token by the integer value of its character is convenient and efficient and can be a help to people using debuggers. This works as long as no character used as input has a value used as an enumerator – and no character set I know of has a printing character with a single-digit integer value. I chose ***PRINT*** as the initial value for ***curr_tok*** because that is the value it will have after the calculator has evaluated an expression and displayed its value. Thus, I ''start the system'' in a normal state to minimize the chance of errors and the need for special startup code.

Each parser function takes a ***bool*** (§4.2) argument indicating whether the function needs to call ***get_token***( ) to get the next token. Each parser function evaluates ''its'' expression and returns the value. The function ***expr***( ) handles addition and subtraction. It consists of a single loop that looks for terms to add or subtract:

```
double expr(bool get)           // add and subtract
{
    double left = term(get);

    for (;;)                    // ''forever''
        switch (curr_tok) {
        case PLUS:
            left += term(true);
            break;
        case MINUS:
            left -= term(true);
            break;
        default:
            return left;
        }
}
```

This function really does not do much itself. In a manner typical of higher-level functions in a large program, it calls other functions to do the work.

The *switch-statement* tests the value of its condition, which is supplied in parentheses after the ***switch*** keyword, against a set of constants. The *break-statement*s are used to exit the *switch-statement*. The constants following the ***case*** labels must be distinct. If the value tested does not match any ***case*** label, the ***default*** is chosen. The programmer need not provide a ***default***.

Note that an expression such as *2-3+4* is evaluated as (*2-3*)*+4*, as specified in the grammar.

The curious notation ***for***(;;) is the standard way to specify an infinite loop; you could pronounce it ''forever.'' It is a degenerate form of a *for-statement* (§6.3.3); ***while***(***true***) is an alternative. The *switch-statement* is executed repeatedly until something different from + and - is found, and then the *return-statement* in the default case is executed.

The operators += and -= are used to handle the addition and subtraction; ***left=left+term***( ) and

left=left-term( ) could have been used without changing the meaning of the program. However, *left+=term*( ) and *left-=term*( ) not only are shorter but also express the intended operation directly. Each assignment operator is a separate lexical token, so *a + = 1;* is a syntax error because of the space between the + and the =.

Assignment operators are provided for the binary operators

```
+   -   *   /   %   &   |   ^   <<   >>
```

so that the following assignment operators are possible

```
=   +=   -=   *=   /=   %=   &=   |=   ^=   <<=  >>=
```

The % is the modulo, or remainder, operator; &, |, and ^ are the bitwise logical operators AND, OR, and exclusive OR; << and >> are the left shift and right shift operators; §6.2 summarizes the operators and their meanings. For a binary operator @ applied to operands of built-in types, an expression *x@=y* means *x=x@y*, except that *x* is evaluated once only.

Chapter 8 and Chapter 9 discuss how to organize a program as a set of modules. With one exception, the declarations for this calculator example can be ordered so that everything is declared exactly once and before it is used. The exception is *expr*( ), which calls *term*( ), which calls *prim*( ), which in turn calls *expr*( ). This loop must be broken somehow. A declaration

```
double expr(bool);
```

before the definition of *prim*( ) will do nicely.

Function *term*( ) handles multiplication and division in the same way *expr*( ) handles addition and subtraction:

```
double term(bool get)          // multiply and divide
{
    double left = prim(get);

    for ( ; ; )
        switch (curr_tok) {
        case MUL:
            left *= prim(true);
            break;
        case DIV:
            if (double d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");
        default:
            return left;
        }
}
```

The result of dividing by zero is undefined and usually disastrous. We therefore test for *0* before dividing and call *error*( ) if we detect a zero divisor. The function *error*( ) is described in §6.1.4.

The variable *d* is introduced into the program exactly where it is needed and initialized immediately. The scope of a name introduced in a condition is the statement controlled by that condition,

and the resulting value is the value of the condition (§6.3.2.1). Consequently, the division and assignment ***left/=d*** is done if and only if ***d*** is nonzero.

The function ***prim*( )** handling a *primary* is much like ***expr*( )** and ***term*( )**, except that because we are getting lower in the call hierarchy a bit of real work is being done and no loop is necessary:

```
double number_value;
string string_value;

double prim(bool get)            // handle primaries
{
    if (get) get_token();

    switch (curr_tok) {
    case NUMBER:                 // floating-point constant
    {   double v = number_value;
        get_token();
        return v;
    }
    case NAME:
    {   double& v = table[string_value];
        if (get_token() == ASSIGN) v = expr(true);
        return v;
    }
    case MINUS:              // unary minus
        return -prim(true);
    case LP:
    {   double e = expr(true);
        if (curr_tok != RP) return error("') expected");
        get_token();             // eat ')'
        return e;
    }
    default:
        return error("primary expected");
    }
}
```

When a ***NUMBER*** (that is, an integer or floating-point literal) is seen, its value is returned. The input routine ***get_token*( )** places the value in the global variable ***number_value***. Use of a global variable in a program often indicates that the structure is not quite clean – that some sort of optimization has been applied. So it is here. Ideally, a lexical token consists of two parts: a value specifying the kind of token (a ***Token_value*** in this program) and (when needed) the value of the token. Here, there is only a single, simple variable, ***curr_tok***, so the global variable ***number_value*** is needed to hold the value of the last ***NUMBER*** read. Eliminating this spurious global variable is left as an exercise (§6.6[21]). Saving the value of ***number_value*** in the local variable ***v*** before calling ***get_token*( )** is not really necessary. For every legal input, the calculator always uses one number in the computation before reading another from input. However, saving the value and displaying it correctly after an error helps the user.

In the same way that the value of the last ***NUMBER*** is kept in ***number_value***, the character string representation of the last ***NAME*** seen is kept in ***string_value***. Before doing anything to a

name, the calculator must first look ahead to see if it is being assigned to or simply read. In both cases, the symbol table is consulted. The symbol table is a *map* (§3.7.4, §17.4.1):

```
map<string,double> table;
```

That is, when *table* is indexed by a *string*, the resulting value is the *double* corresponding to the *string*. For example, if the user enters

```
radius = 6378.388;
```

the calculator will execute

```
double& v = table["radius"];
// ... expr() calculates the value to be assigned ...
v = 6378.388;
```

The reference *v* is used to hold on to the *double* associated with *radius* while *expr*( ) calculates the value *6378.388* from the input characters.

6.1.2 The Input Function [expr.input]

Reading input is often the messiest part of a program. This is because a program must communicate with a person, it must cope with that person's whims, conventions, and seemingly random errors. Trying to force the person to behave in a manner more suitable for the machine is often (rightly) considered offensive. The task of a low-level input routine is to read characters and compose higher-level tokens from them. These tokens are then the units of input for higher-level routines. Here, low-level input is done by *get_token*( ). Writing a low-level input routine need not be an everyday task. Many systems provide standard functions for this.

I build *get_token*( ) in two stages. First, I provide a deceptively simple version that imposes a burden on the user. Next, I modify it into a slightly less elegant, but much easier to use, version.

The idea is to read a character, use that character to decide what kind of token needs to be composed, and then return the *Token_value* representing the token read.

The initial statements read the first non-whitespace character into *ch* and check that the read operation succeeded:

```
Token_value get_token()
{
    char ch = 0;
    cin>>ch;

    switch (ch) {
    case 0:
        return curr_tok=END;      // assign and return
```

By default, operator >> skips whitespace (that is, spaces, tabs, newlines, etc.) and leaves the value of *ch* unchanged if the input operation failed. Consequently, *ch==0* indicates end of input.

Assignment is an operator, and the result of the assignment is the value of the variable assigned to. This allows me to assign the value *END* to *curr_tok* and return it in the same statement. Having a single statement rather than two is useful in maintenance. If the assignment and the return became separated in the code, a programmer might update the one and forget to update to the other.

Let us look at some of the cases separately before considering the complete function. The expression terminator ´;´, the parentheses, and the operators are handled simply by returning their values:

```
case ´;´ :
case ´*´ :
case ´/´ :
case ´+´ :
case ´-´ :
case ´(´ :
case ´)´ :
case ´=´ :
        return curr_tok=Token_value(ch);
```

Numbers are handled like this:

```
case ´0´ : case ´1´ : case ´2´ : case ´3´ : case ´4´ :
case ´5´ : case ´6´ : case ´7´ : case ´8´ : case ´9´ :
case ´.´ :
        cin.putback(ch);
        cin >> number_value;
        return curr_tok=NUMBER;
```

Stacking ***case*** labels horizontally rather than vertically is generally not a good idea because this arrangement is harder to read. However, having one line for each digit is tedious. Because operator >> is already defined for reading floating-point constants into a ***double***, the code is trivial. First the initial character (a digit or a dot) is put back into ***cin***. Then the constant can be read into ***number_value***.

A name is handled similarly:

```
default:                // NAME, NAME =, or error
        if (isalpha(ch)) {
                cin.putback(ch);
                cin>>string_value;
                return curr_tok=NAME;
        }
        error("bad token");
        return curr_tok=PRINT;
```

The standard library function ***isalpha***( ) (§20.4.2) is used to avoid listing every character as a separate ***case*** label. Operator >> applied to a string (in this case, ***string_value***) reads until it hits white-space. Consequently, a user must terminate a name by a space before an operator using the name as an operand. This is less than ideal, so we will return to this problem in §6.1.3.

Here, finally, is the complete input function:

```
Token_value get_token()
{
        char ch = 0;
        cin>>ch;
```

```
        switch (ch) {
        case 0:
                return curr_tok=END;
        case ´;´:
        case ´*´:
        case ´/´:
        case ´+´:
        case ´-´:
        case ´(´:
        case ´)´:
        case ´=´:
                return curr_tok=Token_value(ch);
        case ´0´: case ´1´: case ´2´: case ´3´: case ´4´:
        case ´5´: case ´6´: case ´7´: case ´8´: case ´9´:
        case ´.´:
                cin.putback(ch);
                cin >> number_value;
                return curr_tok=NUMBER;
        default:                        // NAME, NAME =, or error
                if (isalpha(ch)) {
                        cin.putback(ch);
                        cin>>string_value;
                        return curr_tok=NAME;
                }
                error("bad token");
                return curr_tok=PRINT;
        }
}
```

The conversion of an operator to its token value is trivial because the ***Token_value*** of an operator was defined as the integer value of the operator (§4.8).

6.1.3 Low-level Input [expr.low]

Using the calculator as defined so far reveals a few inconveniences. It is tedious to remember to add a semicolon after an expression in order to get its value printed, and having a name terminated by whitespace only is a real nuisance. For example, *x=7* is an identifier – rather than the identifier *x* followed by the operator = and the number *7*. Both problems are solved by replacing the type-oriented default input operations in ***get_token*( )** with code that reads individual characters.

First, we'll make a newline equivalent to the semicolon used to mark the end of expression:

```
Token_value get_token()
{
        char ch;

        do { // skip whitespace except '\n'
                if(!cin.get(ch)) return curr_tok = END;
        } while (ch!=´\n´ && isspace(ch));
```

```
switch (ch) {
case ´;´:
case ´\n´:
        return curr_tok=PRINT;
```

A *do-statement* is used; it is equivalent to a *while-statement* except that the controlled statement is always executed at least once. The call *cin.get(ch)* reads a single character from the standard input stream into *ch*. By default, *get()* does not skip whitespace the way *operator* >> does. The test *if* (!*cin.get(ch)*) fails if no character can be read from *cin*; in this case, *END* is returned to terminate the calculator session. The operator ! (NOT) is used because *get()* returns *true* in case of success.

The standard library function *isspace()* provides the standard test for whitespace (§20.4.2); *isspace(c)* returns a nonzero value if *c* is a whitespace character and zero otherwise. The test is implemented as a table lookup, so using *isspace()* is much faster than testing for the individual whitespace characters. Similar functions test if a character is a digit – *isdigit()* – a letter – *isalpha()* – or a digit or letter – *isalnum()*.

After whitespace has been skipped, the next character is used to determine what kind of lexical token is coming.

The problem caused by >> reading into a string until whitespace is encountered is solved by reading one character at a time until a character that is not a letter or a digit is found:

```
default:                    // NAME, NAME=, or error
        if (isalpha(ch)) {
                string_value = ch;
                while (cin.get(ch) && isalnum(ch)) string_value.push_back(ch);
                cin.putback(ch);
                return curr_tok=NAME;
        }
        error("bad token");
        return curr_tok=PRINT;
```

Fortunately, these two improvements could both be implemented by modifying a single local section of code. Constructing programs so that improvements can be implemented through local modifications only is an important design aim.

6.1.4 Error Handling [expr.error]

Because the program is so simple, error handling is not a major concern. The error function simply counts the errors, writes out an error message, and returns:

```
int no_of_errors;

double error(const string& s)
{
        no_of_errors++;
        cerr << "error: " << s << ´\n´;
        return 1;
}
```

The stream *cerr* is an unbuffered output stream usually used to report errors (§21.2.1).

The reason for returning a value is that errors typically occur in the middle of the evaluation of an expression, so we should either abort that evaluation entirely or return a value that is unlikely to cause subsequent errors. The latter is adequate for this simple calculator. Had *get_token*( ) kept track of the line numbers, *error*( ) could have informed the user approximately where the error occurred. This would be useful when the calculator is used noninteractively (§6.6[19]).

Often, a program must be terminated after an error has occurred because no sensible way of continuing has been devised. This can be done by calling *exit*( ), which first cleans up things like output streams and then terminates the program with its argument as the return value (§9.4.1.1).

More stylized error-handling mechanisms can be implemented using exceptions (see §8.3, Chapter 14), but what we have here is quite suitable for a 150-line calculator.

6.1.5 The Driver [expr.driver]

With all the pieces of the program in place, we need only a driver to start things. In this simple example, *main*( ) can do that:

```
int main()
{
    table["pi"] = 3.1415926535897932385;    // insert predefined names
    table["e"] = 2.7182818284590452354;

    while (cin) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
    }

    return no_of_errors;
}
```

Conventionally, *main*( ) should return zero if the program terminates normally and nonzero otherwise (§3.2). Returning the number of errors accomplishes this nicely. As it happens, the only initialization needed is to insert the predefined names into the symbol table.

The primary task of the main loop is to read expressions and write out the answer. This is achieved by the line:

```
cout << expr(false) << '\n';
```

The argument *false* tells *expr*( ) that it does not need to call *get_token*( ) to get a current token on which to work.

Testing *cin* each time around the loop ensures that the program terminates if something goes wrong with the input stream, and testing for *END* ensures that the loop is correctly exited when *get_token*( ) encounters end-of-file. A *break-statement* exits its nearest enclosing *switch-statement* or loop (that is, a *for-statement*, *while-statement*, or *do-statement*). Testing for *PRINT* (that is, for '\n' and ';') relieves *expr*( ) of the responsibility for handling empty expressions. A *continue-statement* is equivalent to going to the very end of a loop, so in this case

```
while (cin) {
        // ...
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
}
```

is equivalent to

```
while (cin) {
        // ...
        if (curr_tok != PRINT)
                cout << expr(false) << '\n';
}
```

6.1.6 Headers [expr.headers]

The calculator uses standard library facilities. Therefore, appropriate headers must be *#include*d to complete the program:

```
#include<iostream>    // I/O
#include<string>      // strings
#include<map>         // map
#include<cctype>      // isalpha(), etc.
```

All of these headers provide facilities in the *std* namespace, so to use the names they provide we must either use explicit qualification with *std*:: or bring the names into the global namespace by

```
using namespace std;
```

To avoid confusing the discussion of expressions with modularity issues, I did the latter. Chapter 8 and Chapter 9 discuss ways of organizing this calculator into modules using namespaces and how to organize it into source files. On many systems, standard headers have equivalents with a *.h* suffix that declare the classes, functions, etc., and place them in the global namespace (§9.2.1, §9.2.4, §B.3.1).

6.1.7 Command-Line Arguments [expr.command]

After the program was written and tested, I found it a bother to first start the program, then type the expressions, and finally quit. My most common use was to evaluate a single expression. If that expression could be presented as a command-line argument, a few keystrokes could be avoided.

A program starts by calling *main*() (§3.2, §9.4). When this is done, *main*() is given two arguments specifying the number of arguments, usually called *argc*, and an array of arguments, usually called *argv*. The arguments are character strings, so the type of *argv* is *char**[*argc+1*]. The name of the program (as it occurs on the command line) is passed as *argv*[*0*], so *argc* is always at least *1*. The list of arguments is zero-terminated; that is, *argv*[*argc*]==*0*. For example, for the command

```
dc 150/1.1934
```

the arguments have these values:

Because the conventions for calling ***main*( )** are shared with C, C-style arrays and strings are used.

It is not difficult to get hold of a command-line argument. The problem is how to use it with minimal reprogramming. The idea is to read from the command string in the same way that we read from the input stream. A stream that reads from a string is unsurprisingly called an ***istringstream***. Unfortunately, there is no elegant way of making ***cin*** refer to an ***istringstream*.** Therefore, we must find a way of getting the calculator input functions to refer to an ***istringstream***. Furthermore, we must find a way of getting the calculator input functions to refer to an ***istringstream*** or to ***cin*** depending on what kind of command-line argument we supply.

A simple solution is to introduce a global pointer ***input*** that points to the input stream to be used and have every input routine use that:

```
istream* input; // pointer to input stream

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:                                     // read from standard input
        input = &cin;
        break;
    case 2:                                     // read argument string
        input = new istringstream(argv[1]);
        break;
    default:
        error("too many arguments");
        return 1;
    }

    table["pi"] = 3.1415926535897932385;        // insert predefined names
    table["e"] = 2.7182818284590452354;

    while (*input) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
    }

    if (input != &cin) delete input;
    return no_of_errors;
}
```

An ***istringstream*** is a kind of ***istream*** that reads from its character string argument (§21.5.3). Upon reaching the end of its string, an ***istringstream*** fails exactly like other streams do when they hit the end of input (§3.6, §21.3.3). To use an ***istringstream***, you must include ***<sstream>***.

It would be easy to modify ***main()*** to accept several command-line arguments, but this does not appear to be necessary, especially as several expressions can be passed as a single argument:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

I use quotes because ; is the command separator on my UNIX systems. Other systems have different conventions for supplying arguments to a program on startup.

It was inelegant to modify all of the input routines to use ****input*** rather than ***cin*** to gain the flexibility to use alternative sources of input. The change could have been avoided had I shown foresight by introducing something like ***input*** from the start. A more general and useful view is to note that the source of input really should be the parameter of a calculator module. That is, the fundamental problem with this calculator example is that what I refer to as ''the calculator'' is only a collection of functions and data. There is no module (§2.4) or object (§2.5.2) that explicitly represents the calculator. Had I set out to design a calculator module or a calculator type, I would naturally have considered what its parameters should be (§8.5[3], §10.6[16]).

6.1.8 A Note on Style [expr.style]

To programmers unacquainted with associative arrays, the use of the standard library ***map*** as the symbol table seems almost like cheating. It is not. The standard library and other libraries are meant to be used. Often, a library has received more care in its design and implementation than a programmer could afford for a handcrafted piece of code to be used in just one program.

Looking at the code for the calculator, especially at the first version, we can see that there isn't much traditional C-style, low-level code presented. Many of the traditional tricky details have been replaced by uses of standard library classes such as ***ostream***, ***string***, and ***map*** (§3.4, §3.5, §3.7.4, Chapter 17).

Note the relative scarcity of arithmetic, loops, and even assignments. This is the way things ought to be in code that doesn't manipulate hardware directly or implement low-level abstractions.

6.2 Operator Summary [expr.operators]

This section presents a summary of expressions and some examples. Each operator is followed by one or more names commonly used for it and an example of its use. In these tables, a *class_name* is the name of a class, a *member* is a member name, an *object* is an expression yielding a class object, a *pointer* is an expression yielding a pointer, an *expr* is an expression, and an *lvalue* is an expression denoting a nonconstant object. A *type* can be a fully general type name (with *, ( ), etc.) only when it appears in parentheses; elsewhere, there are restrictions (§A.5).

The syntax of expressions is independent of operand types. The meanings presented here apply when the operands are of built-in types (§4.1.1). In addition, you can define meanings for operators applied to operands of user-defined types (§2.5.2, Chapter 11).

Operator Summary	
scope resolution	*class_name* : : *member*
scope resolution	*namespace_name* : : *member*
global	: : *name*
global	: : *qualified-name*
member selection	*object* . *member*
member selection	*pointer* -> *member*
subscripting	*pointer* [*expr*]
function call	*expr* (*expr_list*)
value construction	*type* (*expr_list*)
post increment	*lvalue* ++
post decrement	*lvalue* - -
type identification	***typeid*** (*type*)
run-time type identification	***typeid*** (*expr*)
run-time checked conversion	***dynamic_cast*** < *type* > (*expr*)
compile-time checked conversion	***static_cast*** < *type* > (*expr*)
unchecked conversion	***reinterpret_cast*** < *type* > (*expr*)
const conversion	***const_cast*** < *type* > (*expr*)
size of object	***sizeof*** *expr*
size of type	***sizeof*** (*type*)
pre increment	++ *lvalue*
pre decrement	- - *lvalue*
complement	~ *expr*
not	! *expr*
unary minus	- *expr*
unary plus	+ *expr*
address of	& *lvalue*
dereference	* *expr*
create (allocate)	***new*** *type*
create (allocate and initialize)	***new*** *type* (*expr-list*)
create (place)	***new*** (*expr-list*) *type*
create (place and initialize)	***new*** (*expr-list*) *type* (*expr-list*)
destroy (de-allocate)	***delete*** *pointer*
destroy array	***delete*** [ ] *pointer*
cast (type conversion)	(*type*) *expr*
member selection	*object* .* *pointer-to-member*
member selection	*pointer* ->* *pointer-to-member*
multiply	*expr* * *expr*
divide	*expr* / *expr*
modulo (remainder)	*expr* % *expr*

Operator Summary (continued)	
add (plus)	*expr* + *expr*
subtract (minus)	*expr* - *expr*
shift left	*expr* << *expr*
shift right	*expr* >> *expr*
less than	*expr* < *expr*
less than or equal	*expr* <= *expr*
greater than	*expr* > *expr*
greater than or equal	*expr* >= *expr*
equal	*expr* == *expr*
not equal	*expr* != *expr*
bitwise AND	*expr* & *expr*
bitwise exclusive OR	*expr* ^ *expr*
bitwise inclusive OR	*expr* | *expr*
logical AND	*expr* && *expr*
logical inclusive OR	*expr* || *expr*
simple assignment	*lvalue* = *expr*
multiply and assign	*lvalue* *= *expr*
divide and assign	*lvalue* /= *expr*
modulo and assign	*lvalue* %= *expr*
add and assign	*lvalue* += *expr*
subtract and assign	*lvalue* -= *expr*
shift left and assign	*lvalue* <<= *expr*
shift right and assign	*lvalue* >>= *expr*
AND and assign	*lvalue* &= *expr*
inclusive OR and assign	*lvalue* |= *expr*
exclusive OR and assign	*lvalue* ^= *expr*
conditional expression	*expr* ? *expr* : *expr*
throw exception	***throw*** *expr*
comma (sequencing)	*expr* , *expr*

Each box holds operators with the same precedence. Operators in higher boxes have higher precedence than operators in lower boxes. For example: `a+b*c` means `a+(b*c)` rather than `(a+b)*c` because `*` has higher precedence than `+`.

Unary operators and assignment operators are right-associative; all others are left-associative. For example, `a=b=c` means `a=(b=c)`, `a+b+c` means `(a+b)+c`, and `*p++` means `*(p++)`, *not* `(*p)++`.

A few grammar rules cannot be expressed in terms of precedence (also known as binding strength) and associativity. For example, `a=b<c?d=e:f=g` means `a=((b<c)?(d=e):(f=g))`, but you need to look at the grammar (§A.5) to determine that.

6.2.1 Results [expr.res]

The result types of arithmetic operators are determined by a set of rules known as ''the usual arithmetic conversions'' (§C.6.3). The overall aim is to produce a result of the ''largest'' operand type. For example, if a binary operator has a floating-point operand, the computation is done using floating-point arithmetic and the result is a floating-point value. If it has a ***long*** operand, the computation is done using long integer arithmetic, and the result is a ***long***. Operands that are smaller than an ***int*** (such as ***bool*** and ***char***) are converted to ***int*** before the operator is applied.

The relational operators, ==, <=, etc., produce Boolean results. The meaning and result type of user-defined operators are determined by their declarations (§11.2).

Where logically feasible, the result of an operator that takes an lvalue operand is an lvalue denoting that lvalue operand. For example:

```
void f(int x, int y)
{
    int j = x = y;          // the value of x=y is the value of x after the assignment
    int* p = &++x;          // p points to x
    int* q = &(x++);        // error: x++ is not an lvalue (it is not the value stored in x)
    int* pp = &(x>y?x:y);   // address of the int with the larger value
}
```

If both the second and third operands of ?: are lvalues and have the same type, the result is of that type and is an lvalue. Preserving lvalues in this way allows greater flexibility in using operators. This is particularly useful when writing code that needs to work uniformly and efficiently with both built-in and user-defined types (e.g., when writing templates or programs that generate C++ code).

The result of ***sizeof*** is of an unsigned integral type called ***size_t*** defined in ***<cstddef>***. The result of pointer subtraction is of a signed integral type called ***ptrdiff_t*** defined in ***<cstddef>***.

Implementations do not have to check for arithmetic overflow and hardly any do. For example:

```
void f()
{
    int i = 1;
    while (0 < i) i++;
    cout << "i has become negative!" << i << '\n';
}
```

This will (eventually) try to increase ***i*** past the largest integer. What happens then is undefined, but typically the value ''wraps around'' to a negative number (on my machine *-2147483648*). Similarly, the effect of dividing by zero is undefined, but doing so usually causes abrupt termination of the program. In particular, underflow, overflow, and division by zero do not throw standard exceptions (§14.10).

6.2.2 Evaluation Order [expr.evaluation]

The order of evaluation of subexpressions within an expression is undefined. In particular, you cannot assume that the expression is evaluated left to right. For example:

```
int x = f(2)+g(3);    // undefined whether f() or g() is called first
```

Better code can be generated in the absence of restrictions on expression evaluation order. However, the absence of restrictions on evaluation order can lead to undefined results. For example,

```
int i = 1;
v[i] = i++;        // undefined result
```

may be evaluated as either `v[1]=1` or `v[2]=1` or may cause some even stranger behavior. Compilers can warn about such ambiguities. Unfortunately, most do not.

The operators `,` (comma), && (logical and), and `||` (logical or) guarantee that their left-hand operand is evaluated before their right-hand operand. For example, `b=(a=2,a+1)` assigns `3` to `b`. Examples of the use of `||` and && can be found in §6.2.3. For built-in types, the second operand of && is evaluated only if its first operand is ***true***, and the second operand of `||` is evaluated only if its first operand is ***false***; this is sometimes called *short-circuit evaluation*. Note that the sequencing operator `,` (comma) is logically different from the comma used to separate arguments in a function call. Consider:

```
f1(v[i],i++);          // two arguments
f2( (v[i],i++) );      // one argument
```

The call of *f1* has two arguments, `v[i]` and `i++`, and the order of evaluation of the argument expressions is undefined. Order dependence of argument expressions is very poor style and has undefined behavior. The call of *f2* has one argument, the comma expression `(v[i],i++)`, which is equivalent to `i++`.

Parentheses can be used to force grouping. For example, `a*b/c` means `(a*b)/c` so parentheses must be used to get `a*(b/c)`; `a*(b/c)` may be evaluated as `(a*b)/c` only if the user cannot tell the difference. In particular, for many floating-point computations `a*(b/c)` and `(a*b)/c` are significantly different, so a compiler will evaluate such expressions exactly as written.

6.2.3 Operator Precedence [expr.precedence]

Precedence levels and associativity rules reflect the most common usage. For example,

```
if (i<=0 || max<i) // ...
```

means ‘‘if *i* is less than or equal to *0* or if *max* is less than *i*.’’ That is, it is equivalent to

```
if ( (i<=0) || (max<i) ) // ...
```

and not the legal but nonsensical

```
if (i <= (0||max) < i) // ...
```

However, parentheses should be used whenever a programmer is in doubt about those rules. Use of parentheses becomes more common as the subexpressions become more complicated, but complicated subexpressions are a source of errors. Therefore, if you start feeling the need for parentheses, you might consider breaking up the expression by using an extra variable.

There are cases when the operator precedence does not result in the ‘‘obvious’’ interpretation. For example:

```
if (i&mask == 0)        // oops! == expression as operand for &
```

This does not apply a mask to *i* and then test if the result is zero. Because == has higher precedence than &, the expression is interpreted as *i*&(*mask*==*0*). Fortunately, it is easy enough for a compiler to warn about most such mistakes. In this case, parentheses are important:

```
if ((i&mask) == 0) // ...
```

It is worth noting that the following does not work the way a mathematician might expect:

```
if (0 <= x <= 99) // ...
```

This is legal, but it is interpreted as (*0*<=*x*)<=*99*, where the result of the first comparison is either *true* or *false*. This Boolean value is then implicitly converted to *1* or *0*, which is then compared to *99*, yielding *true*. To test whether *x* is in the range *0*..*99*, we might use:

```
if (0<=x && x<=99) // ...
```

A common mistake for novices is to use = (assignment) instead of == (equals) in a condition:

```
if (a = 7)  // oops! constant assignment in condition
```

This is natural because = means ''equals'' in many languages. Again, it is easy for a compiler to warn about most such mistakes – and many do.

6.2.4 Bitwise Logical Operators [expr.logical]

The bitwise logical operators &, |, ^, ~, >>, and << are applied to objects of integer types – that is, *bool*, *char*, *short*, *int*, *long*, and their *unsigned* counterparts. The results are also integers.

A typical use of bitwise logical operators is to implement the notion of a small set (a bit vector). In this case, each bit of an unsigned integer represents one member of the set, and the number of bits limits the number of members. The binary operator & is interpreted as intersection, | as union, ^ as symmetric difference, and ~ as complement. An enumeration can be used to name the members of such a set. Here is a small example borrowed from an implementation of *ostream*:

```
enum ios_base::iostate {
        goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

The implementation of a stream can set and test its state like this:

```
state = goodbit;
// ...
if (state&(badbit|failbit)) // stream no good
```

The extra parentheses are necessary because & has higher precedence than |.

A function that reaches the end of input might report it like this:

```
state |= eofbit;
```

The |= operator is used to add to the state. A simple assignment, *state=eofbit*, would have cleared all other bits.

These stream state flags are observable from outside the stream implementation. For example, we could see how the states of two streams differ like this:

```
int diff = cin.rdstate()^cout.rdstate();    // rdstate() returns the state
```

Computing differences of stream states is not very common. For other similar types, computing differences is essential. For example, consider comparing a bit vector that represents the set of interrupts being handled with another that represents the set of interrupts waiting to be handled.

Please note that this bit fiddling is taken from the implementation of iostreams rather than from the user interface. Convenient bit manipulation can be very important, but for reliability, maintainability, portability, etc., it should be kept at low levels of a system. For more general notions of a set, see the standard library ***set*** (§17.4.3), ***bitset*** (§17.5.3), and ***vector<bool>*** (§16.3.11).

Using fields (§C.8.1) is really a convenient shorthand for shifting and masking to extract bit fields from a word. This can, of course, also be done using the bitwise logical operators. For example, one could extract the middle 16 bits of a 32-bit ***long*** like this:

```
unsigned short middle(long a) { return (a>>8)&0xffff; }
```

Do not confuse the bitwise logical operators with the logical operators: `&&`, `||`, and `!`. The latter return either ***true*** or ***false***, and they are primarily useful for writing the test in an ***if***, ***while***, or ***for*** statement (§6.3.2, §6.3.3). For example, `!0` (not zero) is the value ***true***, whereas `~0` (complement of zero) is the bit pattern all-ones, which in two's complement representation is the value `-1`.

6.2.5 Increment and Decrement [expr.incr]

The `++` operator is used to express incrementing directly, rather than expressing it indirectly using a combination of an addition and an assignment. By definition, `++lvalue` means `lvalue+=1`, which again means `lvalue=lvalue+1` provided ***lvalue*** has no side effects. The expression denoting the object to be incremented is evaluated once (only). Decrementing is similarly expressed by the `--` operator. The operators `++` and `--` can be used as both prefix and postfix operators. The value of `++x` is the new (that is, incremented) value of `x`. For example, `y=++x` is equivalent to `y=(x+=1)`. The value of `x++`, however, is the old value of `x`. For example, `y=x++` is equivalent to `y=(t=x,x+=1,t)`, where `t` is a variable of the same type as `x`.

Like addition and subtraction of pointers, `++` and `--` on pointers operate in terms of elements of the array into which the pointer points; `p++` makes `p` point to the next element (§5.3.1).

The increment operators are particularly useful for incrementing and decrementing variables in loops. For example, one can copy a zero-terminated string like this:

```
void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

Like C, C++ is both loved and hated for enabling such terse, expression-oriented coding. Because

```
while (*p++ = *q++) ;
```

is more than a little obscure to non-C programmers and because the style of coding is not uncommon in C and C++, it is worth examining more closely.

Consider first a more traditional way of copying an array of characters:

```
int length = strlen(q);
for (int i = 0; i<=length; i++) p[i] = q[i];
```

This is wasteful. The length of a zero-terminated string is found by reading the string looking for the terminating zero. Thus, we read the string twice: once to find its length and once to copy it. So we try this instead:

```
int i;
for (i = 0; q[i]!=0 ; i++) p[i] = q[i];
p[i] = 0;  // terminating zero
```

The variable ***i*** used for indexing can be eliminated because ***p*** and ***q*** are pointers:

```
while (*q != 0) {
        *p = *q;
        p++;        // point to next character
        q++;        // point to next character
}
*p = 0;             // terminating zero
```

Because the post-increment operation allows us first to use the value and then to increment it, we can rewrite the loop like this:

```
while (*q != 0) {
        *p++ = *q++;
}
*p = 0; // terminating zero
```

The value of ***p++ = *q++*** is ***q***. We can therefore rewrite the example like this:

```
while ((*p++ = *q++) != 0) { }
```

In this case, we don't notice that ***q*** is zero until we already have copied it into ***p*** and incremented ***p***. Consequently, we can eliminate the final assignment of the terminating zero. Finally, we can reduce the example further by observing that we don't need the empty block and that the ''*!= 0*'' is redundant because the result of a pointer or integral condition is always compared to zero anyway. Thus, we get the version we set out to discover:

```
while (*p++ = *q++) ;
```

Is this version less readable than the previous versions? Not to an experienced C or C++ programmer. Is this version more efficient in time or space than the previous versions? Except for the first version that called ***strlen*()**, not really. Which version is the most efficient will vary among machine architectures and among compilers.

The most efficient way of copying a zero-terminated character string for your particular machine ought to be the standard string copy function:

```
char* strcpy(char*, const char*);    // from <string.h>
```

For more general copying, the standard ***copy*** algorithm (§2.7.2, §18.6.1) can be used. Whenever possible, use standard library facilities in preference to fiddling with pointers and bytes. Standard library functions may be inlined (§7.1.1) or even implemented using specialized machine

instructions. Therefore, you should measure carefully before believing that some piece of hand-crafted code outperforms library functions.

6.2.6 Free Store [expr.free]

A named object has its lifetime determined by its scope (§4.9.4). However, it is often useful to create an object that exists independently of the scope in which it was created. In particular, it is common to create objects that can be used after returning from the function in which they were created. The operator ***new*** creates such objects, and the operator ***delete*** can be used to destroy them. Objects allocated by ***new*** are said to be ''on the free store'' (also, to be ''heap objects,'' or ''allocated in dynamic memory'').

Consider how we might write a compiler in the style used for the desk calculator (§6.1). The syntax analysis functions might build a tree of the expressions for use by the code generator:

```
struct Enode {
        Token_value oper;
        Enode* left;
        Enode* right;
        // ...
};

Enode* expr(bool get)
{
        Enode* left = term(get);

        for ( ; ; )
                switch(curr_tok) {
                case PLUS:
                case MINUS:
                {       Enode* n = new Enode;     // create an Enode on free store
                        n->oper = curr_tok;
                        n->left = left;
                        n->right = term(true);
                        left = n;
                        break;
                }
                default:
                        return left;              // return node
                }
}
```

A code generator would then use the resulting nodes and delete them:

```
void generate(Enode* n)
{
        switch (n->oper) {
        case PLUS:
                // ...
                delete n;  // delete an Enode from the free store
        }
}
```

An object created by ***new*** exists until it is explicitly destroyed by ***delete***. Then, the space it occupied can be reused by ***new***. A C++ implementation does not guarantee the presence of a ''garbage collector'' that looks out for unreferenced objects and makes them available to ***new*** for reuse. Consequently, I will assume that objects created by ***new*** are manually freed using ***delete***. If a garbage collector is present, the ***delete***s can be omitted in most cases (§C.9.1).

The ***delete*** operator may be applied only to a pointer returned by ***new*** or to zero. Applying ***delete*** to zero has no effect.

More specialized versions of operator ***new*** can also be defined (§15.6).

6.2.6.1 Arrays [expr.array]

Arrays of objects can also be created using ***new***. For example:

```
char* save_string(const char* p)
{
    char* s = new char[strlen(p)+1];
    strcpy(s,p);            // copy from p to s
    return s;
}

int main(int argc, char* argv[])
{
    if (argc < 2) exit(1);
    char* p = save_string(argv[1]);
    // ...
    delete[] p;
}
```

The ''plain'' operator ***delete*** is used to delete individual objects; ***delete***[] is used to delete arrays.

To deallocate space allocated by ***new***, ***delete*** and ***delete***[] must be able to determine the size of the object allocated. This implies that an object allocated using the standard implementation of ***new*** will occupy slightly more space than a static object. Typically, one word is used to hold the object's size.

Note that a ***vector*** (§3.7.1, §16.3) is a proper object and can therefore be allocated and deallocated using plain ***new*** and ***delete***. For example:

```
void f(int n)
{
    vector<int>* p = new vector<int>(n);      // individual object
    int* q = new int[n];                      // array
    // ...
    delete p;
    delete[] q;
}
```

6.2.6.2 Memory Exhaustion [expr.exhaust]

The free store operators ***new***, ***delete***, ***new*[ ]**, and ***delete*[ ]** are implemented using functions:

```
void* operator new(size_t);      // space for individual object
void operator delete(void*);

void* operator new[](size_t);  // space for array
void operator delete[](void*);
```

When operator ***new*** needs to allocate space for an object, it calls ***operator new*( )** to allocate a suitable number of bytes. Similarly, when operator ***new*** needs to allocate space for an array, it calls ***operator new*[ ]( )**.

The standard implementations of ***operator new*( )** and ***operator new*[ ]( )** do not initialize the memory returned.

What happens when ***new*** can find no store to allocate? By default, the allocator throws a ***bad_alloc*** exception. For example:

```
void f()
{
    try {
        for(;;) new char[10000];
    }
    catch(bad_alloc) {
        cerr << "Memory exhausted!\n";
    }
}
```

However much memory we have available, this will eventually invoke the ***bad_alloc*** handler.

We can specify what ***new*** should do upon memory exhaustion. When ***new*** fails, it first calls a function specified by a call to ***set_new_handler*( )** declared in ***<new>***, if any. For example:

```
void out_of_store()
{
    cerr << "operator new failed: out of store\n";
    throw bad_alloc();
}

int main()
{
    set_new_handler(out_of_store); // make out_of_store the new_handler
    for (;;) new char[10000];
    cout << "done\n";
}
```

This will never get to write ***done***. Instead, it will write

```
operator new failed: out of store
```

See §14.4.5 for a plausible implementation of an ***operator new*( )** that checks to see if there is a new handler to call and that throws ***bad_alloc*** if not. A ***new_handler*** might do something more clever than simply terminating the program. If you know how ***new*** and ***delete*** work – for example,

because you provided your own ***operator new*( )** and ***operator delete*( )** – the handler might attempt to find some memory for ***new*** to return. In other words, a user might provide a garbage collector, thus rendering the use of ***delete*** optional. Doing this is most definitely not a task for a beginner, though. For almost everybody who needs an automatic garbage collector, the right thing to do is to acquire one that has already been written and tested (§C.9.1).

By providing a ***new_handler***, we take care of the check for memory exhaustion for every ordinary use of ***new*** in the program. Two alternative ways of controlling memory allocation exist. We can either provide nonstandard allocation and deallocation functions (§15.6) for the standard uses of ***new*** or rely on additional allocation information provided by the user (§10.4.11, §19.4.5).

6.2.7 Explicit Type Conversion [expr.cast]

Sometimes, we have to deal with‘‘raw memory;’’ that is, memory that holds or will hold objects of a type not known to the compiler. For example, a memory allocator may return a ***void**** pointing to newly allocated memory or we might want to state that a given integer value is to be treated as the address of an I/O device:

```
void* malloc(size_t);

void f()
{
        int* p = static_cast<int*>(malloc(100));                    // new allocation used as ints
        IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00);       // device at 0Xff00
        // ...
}
```

A compiler does not know the type of the object pointed to by the ***void****. Nor can it know whether the integer ***0Xff00*** is a valid address. Consequently, the correctness of the conversions are completely in the hands of the programmer. Explicit type conversion, often called *casting*, is occasionally essential. However, traditionally it is seriously overused and a major source of errors.

The ***static_cast*** operator converts between related types such as one pointer type to another, an enumeration to an integral type, or a floating-point type to an integral type. The ***reinterpret_cast*** handles conversions between unrelated types such as an integer to a pointer. This distinction allows the compiler to apply some minimal type checking for ***static_cast*** and makes it easier for a programmer to find the more dangerous conversions represented as ***reinterpret_cast***s. Some ***static_cast***s are portable, but few ***reinterpret_cast***s are. Hardly any guarantees are made for ***reinterpret_cast***, but generally it produces a value of a new type that has the same bit pattern as its argument. If the target has at least as many bits as the original value, we can ***reinterpret_cast*** the result back to its original type and use it. The result of a ***reinterpret_cast*** is guaranteed to be usable only if its result type is the exact type used to define the value involved. Note that ***reinterpret_cast*** is the kind of conversion that must be used for pointers to functions (§7.7).

If you feel tempted to use an explicit type conversion, take the time to consider if it is *really* necessary. In C++, explicit type conversion is unnecessary in most cases when C needs it (§1.6) and also in many cases in which earlier versions of C++ needed it (§1.6.2, §B.2.3). In many programs, explicit type conversion can be completely avoided; in others, its use can be localized to a few routines. In this book, explicit type conversion is used in realistic situations in §6.2.7, §7.7, §13.5, §15.4, and §25.4.1, only.

A form of run-time checked conversion, ***dynamic_cast*** (§15.4.1), and a cast for removing ***const*** qualifiers, ***const_cast*** (§15.4.2.1), are also provided.

From C, C++ inherited the notation ***(T)e***, which performs any conversion that can be expressed as a combination of ***static_cast***s, ***reinterpret_cast***s, and ***const_cast***s to make a value of type ***T*** from the expression ***e*** (§B.2.3). This C-style cast is far more dangerous than the named conversion operators because the notation is harder to spot in a large program and the kind of conversion intended by the programmer is not explicit. That is, ***(T)e*** might be doing a portable conversion between related types, a nonportable conversion between unrelated types, or removing the ***const*** modifier from a pointer type. Without knowing the exact types of ***T*** and ***e***, you cannot tell.

6.2.8 Constructors [expr.ctor]

The construction of a value of type ***T*** from a value ***e*** can be expressed by the functional notation ***T(e)***. For example:

```
void f(double d)
{
    int i = int(d);              // truncate d
    complex z = complex(d);      // make a complex from d
    // ...
}
```

The ***T(e)*** construct is sometimes referred to as a *function-style cast*. For a built-in type ***T***, ***T(e)*** is equivalent to ***static_cast<T>(e)***. Unfortunately, this implies that the use of ***T(e)*** is not always safe. For arithmetic types, values can be truncated and even explicit conversion of a longer integer type to a shorter (such as ***long*** to ***char***) can result in undefined behavior. I try to use the notation exclusively where the construction of a value is well-defined; that is, for narrowing arithmetic conversions (§C.6), for conversion from integers to enumerations (§4.8), and the construction of objects of user-defined types (§2.5.2, §10.2.3).

Pointer conversions cannot be expressed directly using the ***T(e)*** notation. For example, ***char*(2)*** is a syntax error. Unfortunately, the protection that the constructor notation provides against such dangerous conversions can be circumvented by using ***typedef*** names (§4.9.7) for pointer types.

The constructor notation ***T()*** is used to express the default value of type ***T***. For example:

```
void f(double d)
{
    int j = int();               // default int value
    complex z = complex();       // default complex value
    // ...
}
```

The value of an explicit use of the constructor for a built-in type is ***0*** converted to that type (§4.9.5). Thus, ***int()*** is another way of writing ***0***. For a user-defined type ***T***, ***T()*** is defined by the default constructor (§10.4.2), if any.

The use of the constructor notation for built-in types is particularly important when writing templates. Then, the programmer does not know whether a template parameter will refer to a built-in type or a user-defined type (§16.3.4, §17.4.1.2).

6.3 Statement Summary [expr.stmts]

Here are a summary and some examples of C++ statements:

Statement Syntax
statement:
declaration
{ *statement-list*$_{opt}$ }
try { *statement-list*$_{opt}$ } *handler-list*
expression$_{opt}$;
if (*condition*) *statement*
if (*condition*) *statement* **else** *statement*
switch (*condition*) *statement*
while (*condition*) *statement*
do *statement* **while** (*expression*) ;
for (*for-init-statement condition*$_{opt}$; *expression*$_{opt}$) *statement*
case *constant-expression* : *statement*
default : *statement*
break ;
continue ;
return *expression*$_{opt}$;
goto *identifier* ;
identifier : *statement*
statement-list:
statement statement-list$_{opt}$
condition:
expression
type-specifier declarator = *expression*
handler-list:
catch (*exception-declaration*) { *statement-list*$_{opt}$ }
handler-list handler-list$_{opt}$

Note that a declaration is a statement and that there is no assignment statement or procedure call statement; assignments and function calls are expressions. The statements for handling exceptions, *try-block*s, are described in §8.3.1.

6.3.1 Declarations as Statements [expr.dcl]

A declaration is a statement. Unless a variable is declared ***static***, its initializer is executed whenever the thread of control passes through the declaration (see also §10.4.8). The reason for allowing declarations wherever a statement can be used (and a few other places; §6.3.2.1, §6.3.3.1) is to enable the programmer to minimize the errors caused by uninitialized variables and to allow better locality in code. There is rarely a reason to introduce a variable before there is a value for it to hold. For example:

```
void f(vector<string>& v, int i, const char* p)
{
    if (p==0) return;
    if (i<0 || v.size()<=i) error("bad index");
    string s = v[i];
    if (s == p) {
        // ...
    }
    // ...
}
```

The ability to place declarations after executable code is essential for many constants and for single-assignment styles of programming where a value of an object is not changed after initialization. For user-defined types, postponing the definition of a variable until a suitable initializer is available can also lead to better performance. For example,

```
string s; /* ... */ s = "The best is the enemy of the good.";
```

can easily be much slower than

```
string s = "Voltaire";
```

The most common reason to declare a variable without an initializer is that it requires a statement to initialize it. Examples are input variables and arrays.

6.3.2 Selection Statements [expr.select]

A value can be tested by either an ***if*** statement or a ***switch*** statement:

```
if ( condition ) statement
if ( condition ) statement else statement
switch ( condition ) statement
```

The comparison operators

```
==   !=   <   <=   >   >=
```

return the ***bool true*** if the comparison is true and ***false*** otherwise.

In an ***if*** statement, the first (or only) statement is executed if the expression is nonzero and the second statement (if it is specified) is executed otherwise. This implies that any arithmetic or pointer expression can be used as a condition. For example, if ***x*** is an integer, then

```
if (x) // ...
```

means

```
if (x != 0) // ...
```

For a pointer *p*,

```
if (p) // ...
```

is a direct statement of the test ''does *p* point to a valid object,'' whereas

```
if (p != 0) // ...
```

states the same question indirectly by comparing to a value known not to point to an object. Note that the representation of the pointer *0* is not all-zeros on all machines (§5.1.1). Every compiler I have checked generated the same code for both forms of the test.

The logical operators

```
&&   ||   !
```

are most commonly used in conditions. The operators && and || will not evaluate their second argument unless doing so is necessary. For example,

```
if (p && 1<p->count) // ...
```

first tests that *p* is nonzero. It tests *1<p->count* only if *p* is nonzero.

Some *if-statement*s can conveniently be replaced by *conditional-expression*s. For example,

```
if (a <= b)
        max = b;
else
        max = a;
```

is better expressed like this:

```
max = (a<=b) ? b : a;
```

The parentheses around the condition are not necessary, but I find the code easier to read when they are used.

A *switch-statement* can alternatively be written as a set of ***if-statement***s. For example,

```
switch (val) {
case 1:
        f();
        break;
case 2:
        g();
        break;
default:
        h();
        break;
}
```

could alternatively be expressed as

```
if (val == 1)
        f();
else  if (val == 2)
        g();
else
        h();
```

The meaning is the same, but the first (***switch***) version is preferred because the nature of the operation (testing a value against a set of constants) is explicit. This makes the ***switch*** statement easier to read for nontrivial examples. It can also lead to the generation of better code.

Beware that a case of a switch must be terminated somehow unless you want to carry on executing the next case. Consider:

```
switch (val) {                    // beware
case  1:
        cout << "case  1\n";
case  2:
        cout << "case  2\n";
default:
        cout << "default: case  not  found\n";
}
```

Invoked with ***val==1***, this prints

```
case  1
case  2
default: case  not  found
```

to the great surprise of the uninitiated. It is a good idea to comment the (rare) cases in which a fall-through is intentional so that an uncommented fall-through can be assumed to be an error. A ***break*** is the most common way of terminating a case, but a ***return*** is often useful (§6.1.1).

6.3.2.1 Declarations in Conditions [expr.cond]

To avoid accidental misuse of a variable, it is usually a good idea to introduce the variable into the smallest scope possible. In particular, it is usually best to delay the definition of a local variable until one can give it an initial value. That way, one cannot get into trouble by using the variable before its initial value is assigned.

One of the most elegant applications of these two principles is to declare a variable in a condition. Consider:

```
if (double  d = prim(true)) {
        left /= d;
        break;
}
```

Here, ***d*** is declared and initialized and the value of ***d*** after initialization is tested as the value of the condition. The scope of ***d*** extends from its point of declaration to the end of the statement that the condition controls. For example, had there been an ***else***-branch to the *if-statement*, ***d*** would be in scope on both branches.

The obvious and traditional alternative is to declare ***d*** before the condition. However, this opens the scope (literally) for the use of ***d*** before its initialization or after its intended useful life:

```
double d;
// ...

d2 = d;     // oops!
// ...

if (d = prim(true)) {
        left /= d;
        break;
}
// ...

d = 2.0; // two unrelated uses of d
```

In addition to the logical benefits of declaring variables in conditions, doing so also yields the most compact source code.

A declaration in a condition must declare and initialize a single variable or ***const***.

6.3.3 Iteration Statements [expr.loop]

A loop can be expressed as a ***for***, ***while***, or ***do*** statement:

```
while ( condition ) statement
do statement while ( expression ) ;
for ( for-init-statement condition_opt ; expression_opt ) statement
```

Each of these statements executes a statement (called the *controlled* statement or the *body of the loop*) repeatedly until the condition becomes false or the programmer breaks out of the loop some other way.

The *for-statement* is intended for expressing fairly regular loops. The loop variable, the termination condition, and the expression that updates the loop variable can be presented ''up front'' on a single line. This can greatly increase readability and thereby decrease the frequency of errors. If no initialization is needed, the initializing statement can be empty. If the *condition* is omitted, the *for-statement* will loop forever unless the user explicitly exits it by a ***break***, ***return***, ***goto***, ***throw***, or some less obvious way such as a call of ***exit***( ) (§9.4.1.1). If the *expression* is omitted, we must update some form of loop variable in the body of the loop. If the loop isn't of the simple ''introduce a loop variable, test the condition, update the loop variable'' variety, it is often better expressed as a *while-statement*. A *for-statement* is also useful for expressing a loop without an explicit termination condition:

```
for(;;) { // ''forever''
        // ...
}
```

A *while-statement* simply executes its controlled statement until its condition becomes ***false***. I tend to prefer *while-statement*s over *for-statement*s when there isn't an obvious loop variable or where the update of a loop variable naturally comes in the middle of the loop body. An input loop is an example of a loop where there is no obvious loop variable:

```
while(cin>>ch) // ...
```

In my experience, the *do-statement* is a source of errors and confusion. The reason is that its body is always executed once before the condition is evaluated. However, for the body to work correctly, something very much like the condition must hold even the first time through. More often than I would have guessed, I have found that condition not to hold as expected either when the program was first written and tested or later after the code preceding it has been modified. I also prefer the condition ''up front where I can see it.'' Consequently, I tend to avoid *do-statement*s.

6.3.3.1 Declarations in For-Statements [expr.for]

A variable can be declared in the initializer part of a *for-statement*. If that initializer is a declaration, the variable (or variables) it introduces is in scope until the end of the *for-statement*. For example:

```
void f(int v[], int max)
{
    for (int i = 0; i<max; i++) v[i] = i*i;
}
```

If the final value of an index needs to be known after exit from a ***for***-loop, the index variable must be declared outside the ***for***-loop (e.g., §6.3.4).

6.3.4 Goto [expr.goto]

C++ possesses the infamous ***goto***:

```
goto identifier ;
identifier : statement
```

The ***goto*** has few uses in general high-level programming, but it can be very useful when C++ code is generated by a program rather than written directly by a person; for example, ***goto***s can be used in a parser generated from a grammar by a parser generator. The ***goto*** can also be important in the rare cases in which optimal efficiency is essential, for example, in the inner loop of some real-time application.

One of the few sensible uses of ***goto*** in ordinary code is to break out from a nested loop or *switch-statement* (a ***break*** breaks out of only the innermost enclosing loop or *switch-statement*). For example:

```
void f()
{
    int i;
    int j;
```

```
        for (i = 0; i<n; i++)
            for (j = 0; j<m; j++) if (nm[i][j] == a) goto found;
        // not found
        // ...
found:
        // nm[i][j] == a
}
```

There is also a ***continue*** statement that, in effect, goes to the end of a loop statement, as explained in §6.1.5.

6.4 Comments and Indentation [expr.comment]

Judicious use of comments and consistent use of indentation can make the task of reading and understanding a program much more pleasant. Several different consistent styles of indentation are in use. I see no fundamental reason to prefer one over another (although, like most programmers, I have my preferences, and this book reflects them). The same applies to styles of comments.

Comments can be misused in ways that seriously affect the readability of a program. The compiler does not understand the contents of a comment, so it has no way of ensuring that a comment

[1] is meaningful,
[2] describes the program, and
[3] is up to date.

Most programs contain comments that are incomprehensible, ambiguous, and just plain wrong. Bad comments can be worse than no comments.

If something can be stated *in the language itself*, it should be, and not just mentioned in a comment. This remark is aimed at comments such as these:

```
// variable "v" must be initialized

// variable "v" must be used only by function "f()"

// call function "init()" before calling any other function in this file

// call function "cleanup()" at the end of your program

// don't use function "weird()"

// function "f()" takes two arguments
```

Such comments can often be rendered unnecessary by proper use of C++. For example, one might utilize the linkage rules (§9.2) and the visibility, initialization, and cleanup rules for classes (see §10.4.1) to make the preceding examples redundant.

Once something has been stated clearly in the language, it should not be mentioned a second time in a comment. For example:

```
a = b+c;    // a becomes b+c
count++;    // increment the counter
```

Such comments are worse than simply redundant. They increase the amount of text the reader has to look at, they often obscure the structure of the program, and they may be wrong. Note, however,

that such comments are used extensively for teaching purposes in programming language textbooks such as this. This is one of the many ways a program in a textbook differs from a real program.

My preference is for:

[1] A comment for each source file stating what the declarations in it have in common, references to manuals, general hints for maintenance, etc.
[2] A comment for each class, template, and namespace
[3] A comment for each nontrivial function stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
[4] A comment for each global and namespace variable and constant
[5] A few comments where the code is nonobvious and/or nonportable
[6] Very little else

For example:

```
//      tbl.c: Implementation of the symbol table.

/*
        Gaussian elimination with partial pivoting.
        See Ralston: "A first course ..." pg 411.
*/

//      swap() assumes the stack layout of an SGI R6000.

/*************************************

        Copyright (c) 1997 AT&T, Inc.
        All rights reserved

*************************************/
```

A well-chosen and well-written set of comments is an essential part of a good program. Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.

Note also that if `//` comments are used exclusively in a function, then any part of that function can be commented out using `/* */` style comments, and vice versa.

6.5 Advice [expr.advice]

[1] Prefer the standard library to other libraries and to ‘‘handcrafted code;’’ §6.1.8.
[2] Avoid complicated expressions; §6.2.3.
[3] If in doubt about operator precedence, parenthesize; §6.2.3.
[4] Avoid explicit type conversion (casts); §6.2.7.
[5] When explicit type conversion is necessary, prefer the more specific cast operators to the C-style cast; §6.2.7.
[6] Use the $T(e)$ notation exclusively for well-defined construction; §6.2.8.
[7] Avoid expressions with undefined order of evaluation; §6.2.2.
[8] Avoid ***goto***; §6.3.4.
[9] Avoid *do-statement*s; §6.3.3.
[10] Don’t declare a variable until you have a value to initialize it with; §6.3.1, §6.3.2.1, §6.3.3.1.

[11] Keep comments crisp; §6.4.
[12] Maintain a consistent indentation style; §6.4.
[13] Prefer defining a member *operator new*( ) (§15.6) to replacing the global *operator new*( ); §6.2.6.2.
[14] When reading input, always consider ill-formed input; §6.1.3.

6.6 Exercises [expr.exercises]

1. (∗1) Rewrite the following *for* statement as an equivalent *while* statement:

```
for (i=0; i<max_length; i++) if (input_line[i] == ´?´) quest_count++;
```

   Rewrite it to use a pointer as the controlled variable, that is, so that the test is of the form `*p=´?´`.
2. (∗1) Fully parenthesize the following expressions:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1,2)+3
a = - 1 + + b - - - 5
a = b == c ++
a = b = c = 0
a[4][2] *= * b ? c : * d * 2
a-b,c=d
```

3. (∗2) Read a sequence of possibly whitespace-separated (name,value) pairs, where the name is a single whitespace-separated word and the value is an integer or a floating-point value. Compute and print the sum and mean for each name and the sum and mean for all names. Hint: §6.1.8.
4. (∗1) Write a table of values for the bitwise logical operations (§6.2.4) for all possible combinations of *0* and *1* operands.
5. (∗1.5) Find 5 different C++ constructs for which the meaning is undefined (§C.2). (∗1.5) Find 5 different C++ constructs for which the meaning is implementation-defined (§C.2).
6. (∗1) Find 10 different examples of nonportable C++ code.
7. (∗2) Write 5 expressions for which the order of evaluation is undefined. Execute them to see what one or – preferably – more implementations do with them.
8. (∗1.5) What happens if you divide by zero on your system? What happens in case of overflow and underflow?
9. (∗1) Fully parenthesize the following expressions:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

10. (*2) Write these functions: ***strlen***( ), which returns the length of a C-style string; ***strcpy***( ), which copies a string into another; and ***strcmp***( ), which compares two strings. Consider what the argument types and return types ought to be. Then compare your functions with the standard library versions as declared in <*cstring*> (<*string.h*>) and as specified in §20.4.1.
11. (∗1) See how your compiler reacts to these errors:

```
void f(int a, int b)
{
    if (a = 3) // ...
    if (a&077 == 0) // ...
    a := b+1;
}
```

Devise more simple errors and see how the compiler reacts.

12. (∗2) Modify the program from §6.6[3] to also compute the median.
13. (∗2) Write a function ***cat***( ) that takes two C-style string arguments and returns a string that is the concatenation of the arguments. Use ***new*** to find store for the result.
14. (∗2) Write a function ***rev***( ) that takes a string argument and reverses the characters in it. That is, after ***rev***(***p***) the last character of ***p*** will be the first, etc.
15. (∗1.5) What does the following example do?

```
void send(int* to, int* from, int count)
//    Duff's device. Helpful comment deliberately deleted.
{
    int n = (count+7)/8;
    switch (count%8) {
    case 0:    do { *to++ = *from++;
    case 7:         *to++ = *from++;
    case 6:         *to++ = *from++;
    case 5:         *to++ = *from++;
    case 4:         *to++ = *from++;
    case 3:         *to++ = *from++;
    case 2:         *to++ = *from++;
    case 1:         *to++ = *from++;
            } while (--n>0);
    }
}
```

Why would anyone write something like that?

16. (∗2) Write a function ***atoi***(***const char****) that takes a string containing digits and returns the corresponding ***int***. For example, ***atoi***("*123*") is *123*. Modify ***atoi***( ) to handle C++ octal and hexadecimal notation in addition to plain decimal numbers. Modify ***atoi***( ) to handle the C++

character constant notation.

17. (∗2) Write a function ***itoa***`(`***int i, char b***`[])` that creates a string representation of ***i*** in ***b*** and returns ***b***.
18. (*2) Type in the calculator example and get it to work. Do not ‘‘save time’’ by using an already entered text. You’ll learn most from finding and correcting ‘‘little silly errors.’’
19. (∗2) Modify the calculator to report line numbers for errors.
20. (∗3) Allow a user to define functions in the calculator. Hint: Define a function as a sequence of operations just as a user would have typed them. Such a sequence can be stored either as a character string or as a list of tokens. Then read and execute those operations when the function is called. If you want a user-defined function to take arguments, you will have to invent a notation for that.
21. (∗1.5) Convert the desk calculator to use a ***symbol*** structure instead of using the static variables ***number_value*** and ***string_value***.
22. (∗2.5) Write a program that strips comments out of a C++ program. That is, read from ***cin***, remove both `//` comments and `/* */` comments, and write the result to ***cout***. Do not worry about making the layout of the output look nice (that would be another, and much harder, exercise). Do not worry about incorrect programs. Beware of `//`, `/*`, and `*/` in comments, strings, and character constants.
23. (∗2) Look at some programs to get an idea of the variety of indentation, naming, and commenting styles actually used.

7

Functions

To iterate is human,
to recurse divine.
– L. Peter Deutsch

Function declarations and definitions — argument passing — return values — function overloading — ambiguity resolution — default arguments — ***stdargs*** — pointers to functions — macros — advice — exercises.

7.1 Function Declarations [fct.dcl]

The typical way of getting something done in a C++ program is to call a function to do it. Defining a function is the way you specify how an operation is to be done. A function cannot be called unless it has been previously declared.

A function declaration gives the name of the function, the type of the value returned (if any) by the function, and the number and types of the arguments that must be supplied in a call of the function. For example:

```
Elem* next_elem();
char* strcpy(char* to, const char* from);
void exit(int);
```

The semantics of argument passing are identical to the semantics of initialization. Argument types are checked and implicit argument type conversion takes place when necessary. For example:

```
double sqrt(double);

double sr2 = sqrt(2);          // call sqrt() with the argument double(2)
double sq3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

The value of such checking and type conversion should not be underestimated.

A function declaration may contain argument names. This can be a help to the reader of a program, but the compiler simply ignores such names. As mentioned in §4.7, ***void*** as a return type means that the function does not return a value.

7.1.1 Function Definitions [fct.def]

Every function that is called in a program must be defined somewhere (once only). A function definition is a function declaration in which the body of the function is presented. For example:

```
extern void swap(int*, int*);  // a declaration

void swap(int* p, int* q)      // a definition
{
    int t = *p;
    *p = *q;
    *q = t;
}
```

The type of the definition and all declarations for a function must specify the same type. The argument names, however, are not part of the type and need not be identical.

It is not uncommon to have function definitions with unused arguments:

```
void search(table* t, const char* key, const char*)
{
    // no use of the third argument
}
```

As shown, the fact that an argument is unused can be indicated by not naming it. Typically, unnamed arguments arise from the simplification of code or from planning ahead for extensions. In both cases, leaving the argument in place, although unused, ensures that callers are not affected by the change.

A function can be defined to be ***inline***. For example:

```
inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

The ***inline*** specifier is a hint to the compiler that it should attempt to generate code for a call of ***fac***( ) inline rather than laying down the code for the function once and then calling through the usual function call mechanism. A clever compiler can generate the constant *720* for a call ***fac***(*6*). The possibility of mutually recursive inline functions, inline functions that recurse or not depending on input, etc., makes it impossible to guarantee that every call of an ***inline*** function is actually inlined. The degree of cleverness of a compiler cannot be legislated, so one compiler might generate *720*, another *6*****fac***(*5*), and yet another an un-inlined call ***fac***(*6*).

To make inlining possible in the absence of unusually clever compilation and linking facilities, the definition – and not just the declaration – of an inline function must be in scope (§9.2). An ***inline*** specifier does not affect the semantics of a function. In particular, an inline function still has a unique address and so has ***static*** variables (§7.1.2) of an inline function.

7.1.2 Static Variables [fct.static]

A local variable is initialized when the thread of execution reaches its definition. By default, this happens in every call of the function and each invocation of the function has its own copy of the variable. If a local variable is declared ***static***, a single, statically allocated object will be used to represent that variable in all calls of the function. It will be initialized only the first time the thread of execution reaches its definition. For example:

```
void f(int a)
{
    while (a--) {
        static int n = 0;          // initialized once
        int x = 0;                 // initialized n times

        cout << "n == " << n++ << ", x == " << x++ << '\n';
    }
}

int main()
{
    f(3);
}
```

This prints:

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

A static variable provides a function with ‘‘a memory’’ without introducing a global variable that might be accessed and corrupted by other functions (see also §10.2.4).

7.2 Argument Passing [fct.arg]

When a function is called, store is set aside for its formal arguments and each formal argument is initialized by its corresponding actual argument. The semantics of argument passing are identical to the semantics of initialization. In particular, the type of an actual argument is checked against the type of the corresponding formal argument, and all standard and user-defined type conversions are performed. There are special rules for passing arrays (§7.2.1), a facility for passing unchecked arguments (§7.6), and a facility for specifying default arguments (§7.5). Consider:

```
void f(int val, int& ref)
{
    val++;
    ref++;
}
```

When ***f*( )** is called, ***val*++** increments a local copy of the first actual argument, whereas ***ref*++** increments the second actual argument. For example,

```
void g()
{
    int i = 1;
    int j = 1;
    f(i,j);
}
```

will increment ***j*** but not ***i***. The first argument, ***i***, is passed *by value*, the second argument, ***j***, is passed *by reference*. As mentioned in §5.5, functions that modify call-by-reference arguments can make programs hard to read and should most often be avoided (but see §21.2.1). It can, however, be noticeably more efficient to pass a large object by reference than to pass it by value. In that case, the argument might be declared ***const*** to indicate that the reference is used for efficiency reasons only and not to enable the called function to change the value of the object:

```
void f(const Large& arg)
{
    // the value of "arg" cannot be changed without explicit use of type conversion
}
```

The absence of ***const*** in the declaration of a reference argument is taken as a statement of intent to modify the variable:

```
void g(Large& arg); // assume that g() modifies arg
```

Similarly, declaring a pointer argument ***const*** tells readers that the value of an object pointed to by that argument is not changed by the function. For example:

```
int strlen(const char*);                    // number of characters in a C-style string
char* strcpy(char* to, const char* from);   // copy a C-style string
int strcmp(const char*, const char*);       // compare C-style strings
```

The importance of using ***const*** arguments increases with the size of a program.

Note that the semantics of argument passing are different from the semantics of assignment. This is important for ***const*** arguments, reference arguments, and arguments of some user-defined types (§10.4.4.1).

A literal, a constant, and an argument that requires conversion can be passed as a ***const***& argument, but not as a non-***const*** argument. Allowing conversions for a ***const T***& argument ensures that such an argument can be given exactly the same set of values as a ***T*** argument by passing the value in a temporary, if necessary. For example:

```
float fsqrt(const float&); // Fortran-style sqrt taking a reference argument

void g(double d)
{
    float r = fsqrt(2.0f);      // pass ref to temp holding 2.0f
    r = fsqrt(r);               // pass ref to r
    r = fsqrt(d);               // pass ref to temp holding float(d)
}
```

Disallowing conversions for non-***const*** reference arguments (§5.5) avoids the possibility of silly mistakes arising from the introduction of temporaries. For example:

```
void update(float& i);

void g(double d, float r)
{
    update(2.0f);   // error: const argument
    update(r);      // pass ref to r
    update(d);      // error: type conversion required
}
```

Had these calls been allowed, *update*( ) would quietly have updated temporaries that immediately were deleted. Usually, that would come as an unpleasant surprise to the programmer.

7.2.1 Array Arguments [fct.array]

If an array is used as a function argument, a pointer to its initial element is passed. For example:

```
int strlen(const char*);

void f()
{
    char v[] = "an array";
    int i = strlen(v);
    int j = strlen("Nicholas");
}
```

That is, an argument of type *T*[ ] will be converted to a *T** when passed as an argument. This implies that an assignment to an element of an array argument changes the value of an element of the argument array. In other words, arrays differ from other types in that an array is not (and cannot be) passed by value.

The size of an array is not available to the called function. This can be a nuisance, but there are several ways of circumventing this problem. C-style strings are zero-terminated, so their size can be computed easily. For other arrays, a second argument specifying the size can be passed. For example:

```
void compute1(int* vec_ptr, int vec_size);      // one way

struct Vec {
    int* ptr;
    int size;
};

void compute2(const Vec& v);        // another way
```

Alternatively, a type such as *vector* (§3.7.1, §16.3) can be used instead of an array.

Multidimensional arrays are trickier (see §C.7), but often arrays of pointers can be used instead, and they need no special treatment. For example:

```
char* day[] = {
    "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

Again, *vector* and similar types are alternatives to the built-in, low-level arrays and pointers.

7.3 Value Return [fct.return]

A value must be returned from a function that is not declared *void* (however, *main*( ) is special; see §3.2). Conversely, a value cannot be returned from a *void* function. For example:

```
int f1() { }                // error: no value returned
void f2() { }               // ok

int f3() { return 1; }      // ok
void f4() { return 1; }     // error: return value in void function

int f5() { return; }        // error: return value missing
void f6() { return; }       // ok
```

A return value is specified by a return statement. For example:

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

A function that calls itself is said to be *recursive*.

There can be more than one return statement in a function:

```
int fac2(int n)
{
    if (n > 1) return n*fac2(n-1);
    return 1;
}
```

Like the semantics of argument passing, the semantics of function value return are identical to the semantics of initialization. A return statement is considered to initialize an unnamed variable of the returned type. The type of a return expression is checked against the type of the returned type, and all standard and user-defined type conversions are performed. For example:

```
double f() { return 1; }   // 1 is implicitly converted to double(1)
```

Each time a function is called, a new copy of its arguments and local (automatic) variables is created. The store is reused after the function returns, so a pointer to a local variable should never be returned. The contents of the location pointed to will change unpredictably:

```
int* fp() { int local = 1; /* ... */ return &local; }    // bad
```

This error is less common than the equivalent error using references:

```
int& fr() { int local = 1; /* ... */ return local; }    // bad
```

Fortunately, a compiler can easily warn about returning references to local variables.

A *void* function cannot return a value. However, a call of a *void* function doesn't yield a value, so a *void* function can use a call of a *void* function as the expression in a *return* statement. For example:

```
void g(int* p);

void h(int* p) { /* ... */ return g(p); }   // ok: return of ''no value''
```

This form of return is important when writing template functions where the return type is a template parameter (see §18.4.4.2).

7.4 Overloaded Function Names [fct.over]

Most often, it is a good idea to give different functions different names, but when some functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name. Using the same name for operations on different types is called *overloading*. The technique is already used for the basic operations in C++. That is, there is only one name for addition, +, yet it can be used to add values of integer, floating-point, and pointer types. This idea is easily extended to functions defined by the programmer. For example:

```
void print(int);              // print an int
void print(const char*);      // print a C-style character string
```

As far as the compiler is concerned, the only thing functions of the same name have in common is that name. Presumably, the functions are in some sense similar, but the language does not constrain or aid the programmer. Thus overloaded function names are primarily a notational convenience. This convenience is significant for functions with conventional names such as ***sqrt***, ***print***, and ***open***. When a name is semantically significant, this convenience becomes essential. This happens, for example, with operators such as +, *, and <<, in the case of constructors (§11.7), and in generic programming (§2.7.2, Chapter 18). When a function ***f*** is called, the compiler must figure out which of the functions with the name ***f*** is to be invoked. This is done by comparing the types of the actual arguments with the types of the formal arguments of all functions called ***f***. The idea is to invoke the function that is the best match on the arguments and give a compile-time error if no function is the best match. For example:

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print(long)
    print(1.0);     // print(double)
    print(1);       // error, ambiguous: print(long(1)) or print(double(1))?
}
```

Finding the right version to call from a set of overloaded functions is done by looking for a best match between the type of the argument expression and the parameters (formal arguments) of the functions. To approximate our notions of what is reasonable, a series of criteria are tried in order:

[1] Exact match; that is, match using no or only trivial conversions (for example, array name to pointer, function name to pointer to function, and ***T*** to ***const T***)

[2] Match using promotions; that is, integral promotions (***bool*** to ***int***, ***char*** to ***int***, ***short*** to ***int***, and their ***unsigned*** counterparts; §C.6.1), ***float*** to ***double***, and ***double*** to ***long double***

[3] Match using standard conversions (for example, ***int*** to ***double***, ***double*** to ***int***, ***Derived**** to ***Base**** (§12.2), T* to ***void**** (§5.6), ***int*** to ***unsigned int***; §C.6)

[4] Match using user-defined conversions (§11.4)

[5] Match using the ellipsis . . . in a function declaration (§7.6)

If two matches are found at the highest level where a match is found, the call is rejected as ambiguous. The resolution rules are this elaborate primarily to take into account the elaborate C and C++ rules for built-in numeric types (§C.6). For example:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
    print(c);        // exact match: invoke print(char)
    print(i);        // exact match: invoke print(int)
    print(s);        // integral promotion: invoke print(int)
    print(f);        // float to double promotion: print(double)

    print('a');      // exact match: invoke print(char)
    print(49);       // exact match: invoke print(int)
    print(0);        // exact match: invoke print(int)
    print("a");      // exact match: invoke print(const char*)
}
```

The call ***print(0)*** invokes ***print(int)*** because ***0*** is an ***int***. The call ***print('a')*** invokes ***print(char)*** because ***'a'*** is a ***char*** (§4.3.1). The reason to distinguish between conversions and promotions is that we want to prefer safe promotions, such as ***char*** to ***int***, over unsafe conversions, such as ***int*** to ***char***.

The overloading resolution is independent of the order of declaration of the functions considered.

Overloading relies on a relatively complicated set of rules, and occasionally a programmer will be surprised which function is called. So, why bother? Consider the alternative to overloading. Often, we need similar operations performed on objects of several types. Without overloading, we must define several functions with different names:

```
void print_int(int);
void print_char(char);
void print_string(const char*); // C-style string

void g(int i, char c, const char* p, double d)
{
    print_int(i);        // ok
    print_char(c);       // ok
    print_string(p);     // ok

    print_int(c);        // ok? calls print_int(int(c))
    print_char(i);       // ok? calls print_char(char(i))
    print_string(i);     // error
    print_int(d);        // ok? calls print_int(int(d))
}
```

Compared to the overloaded ***print()***, we have to remember several names and remember to use those correctly. This can be tedious, defeats attempts to do generic programming (§2.7.2), and generally encourages the programmer to focus on relatively low-level type issues. Because there is no overloading, all standard conversions apply to arguments to these functions. It can also lead to

errors. In the previous example, this implies that only one of the four calls with a ''wrong'' argument is caught by the compiler. Thus, overloading can increase the chances that an unsuitable argument will be rejected by the compiler.

7.4.1 Overloading and Return Type [fct.return]

Return types are not considered in overload resolution. The reason is to keep resolution for an individual operator (§11.2.1, §11.2.4) or function call context-independent. Consider:

```
float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
    float fl = sqrt(da);    // call sqrt(double)
    double d = sqrt(da);    // call sqrt(double)
    fl = sqrt(fla);         // call sqrt(float)
    d = sqrt(fla);          // call sqrt(float)
}
```

If the return type were taken into account, it would no longer be possible to look at a call of ***sqrt*( )** in isolation and determine which function was called.

7.4.2 Overloading and Scopes [fct.scope]

Functions declared in different non-namespace scopes do not overload. For example:

```
void f(int);

void g()
{
    void f(double);
    f(1);           // call f(double)
}
```

Clearly, ***f*(*int*)** would have been the best match for ***f*(*1*)**, but only ***f*(*double*)** is in scope. In such cases, local declarations can be added or subtracted to get the desired behavior. As always, intentional hiding can be a useful technique, but unintentional hiding is a source of surprises. When overloading across class scopes (§15.2.2) or namespace scopes (§8.2.9.2) is wanted, *using-declarations* or ***using-directives*** can be used (§8.2.2). See also §8.2.6 and §8.2.9.2.

7.4.3 Manual Ambiguity Resolution [fct.man.ambig]

Declaring too few (or too many) overloaded versions of a function can lead to ambiguities. For example:

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);
```

```
void k(int i)
{
    f1(i);      // ambiguous: f1(char) or f1(long)
    f2(0);      // ambiguous: f2(char*) or f2(int*)
}
```

Where possible, the thing to do in such cases is to consider the set of overloaded versions of a function as a whole and see if it makes sense according to the semantics of the function. Often the problem can be solved by adding a version that resolves ambiguities. For example, adding

```
inline void f1(int n) { f1(long(n)); }
```

would resolve all ambiguities similar to *f1(i)* in favor of the larger type *long int*.

One can also add an explicit type conversion to resolve a specific call. For example:

```
f2(static_cast<int*>(0));
```

However, this is most often simply an ugly stopgap. Soon another similar call will be made and have to be dealt with.

Some C++ novices get irritated by the ambiguity errors reported by the compiler. More experienced programmers appreciate these error messages as useful indicators of design errors.

7.4.4 Resolution for Multiple Arguments [fct.fct.res]

Given the overload resolution rules, one can ensure that the simplest algorithm (function) will be used when the efficiency or precision of computations differs significantly for the types involved. For example:

```
int pow(int, int);
double pow(double, double);

complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);

void k(complex z)
{
    int i = pow(2,2);              // invoke pow(int,int)
    double d = pow(2.0,2.0);       // invoke pow(double,double)
    complex z2 = pow(2,z);         // invoke pow(double,complex)
    complex z3 = pow(z,2);         // invoke pow(complex,int)
    complex z4 = pow(z,z);         // invoke pow(complex,complex)
}
```

In the process of choosing among overloaded functions with two or more arguments, a best match is found for each argument using the rules from §7.4. A function that is the best match for one argument and a better than or equal match for all other arguments is called. If no such function exists, the call is rejected as ambiguous. For example:

```
void g()
{
    double d = pow(2.0,2);   // error: pow(int(2.0),2) or pow(2.0,double(2))?
}
```

The call is ambiguous because `2.0` is the best match for the first argument of `pow(double,double)` and `2` is the best match for the second argument of `pow(int,int)`.

7.5 Default Arguments [fct.defarg]

A general function often needs more arguments than are necessary to handle simple cases. In particular, functions that construct objects (§10.2.3) often provide several options for flexibility. Consider a function for printing an integer. Giving the user an option of what base to print it in seems reasonable, but in most programs integers will be printed as decimal integer values. For example:

```
void print(int value, int base =10); // default base is 10

void f()
{
    print(31);
    print(31,10);
    print(31,16);
    print(31,2);
}
```

might produce this output:

```
31 31 1f 11111
```

The effect of a default argument can alternatively be achieved by overloading:

```
void print(int value, int base);
inline void print(int value) { print(value,10); }
```

However, overloading makes it less obvious to the reader that the intent is to have a single print function plus a shorthand.

A default argument is type checked at the time of the function declaration and evaluated at the time of the call. Default arguments may be provided for trailing arguments only. For example:

```
int f(int, int =0, char* =0);    // ok
int g(int =0, int =0, char*);    // error
int h(int =0, int, char* =0);    // error
```

Note that the space between the `*` and the `=` is significant (`*=` is an assignment operator; §6.2):

```
int nasty(char*=0);              // syntax error
```

A default argument can be repeated in a subsequent declaration in the same scope but not changed. For example:

```
void f(int x = 7);
void f(int = 7);          // ok
void f(int = 8);          // error: different default arguments

void g()
{
    void f(int x = 9);    // ok: this declaration hides the outer one
    // ...
}
```

Declaring a name in a nested scope so that the name hides a declaration of the same name in an outer scope is error prone.

7.6 Unspecified Number of Arguments [fct.stdarg]

For some functions, it is not possible to specify the number and type of all arguments expected in a call. Such a function is declared by terminating the list of argument declarations with the ellipsis (`...`), which means ''and maybe some more arguments.'' For example:

```
int printf(const char* ...);
```

This specifies that a call of the C standard library function ***printf*( )** (§21.8) must have at least one argument, a ***char****, but may or may not have others. For example:

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n",2,3,5);
```

Such a function must rely on information not available to the compiler when interpreting its argument list. In the case of ***printf*( )**, the first argument is a format string containing special character sequences that allow ***printf*( )** to handle other arguments correctly; ***%s*** means ''expect a ***char**** argument'' and ***%d*** means ''expect an ***int*** argument.'' However, the compiler cannot in general know that, so it cannot ensure that the expected arguments are really there or that an argument is of the proper type. For example,

```
#include <stdio.h>

int main()
{
    printf("My name is %s %s\n",2);
}
```

will compile and (at best) cause some strange-looking output (try it!).

Clearly, if an argument has not been declared, the compiler does not have the information needed to perform the standard type checking and type conversion for it. In that case, a ***char*** or a ***short*** is passed as an ***int*** and a ***float*** is passed as a ***double***. This is not necessarily what the programmer expects.

A well-designed program needs at most a few functions for which the argument types are not completely specified. Overloaded functions and functions using default arguments can be used to

take care of type checking in most cases when one would otherwise consider leaving argument types unspecified. Only when both the number of arguments *and* the type of arguments vary is the ellipsis necessary. The most common use of the ellipsis is to specify an interface to C library functions that were defined before C++ provided alternatives:

```
int fprintf(FILE*, const char* ...);        // from <cstdio>
int execl(const char* ...);                 // from UNIX header
```

A standard set of macros for accessing the unspecified arguments in such functions can be found in *<cstdarg>*. Consider writing an error function that takes one integer argument indicating the severity of the error followed by an arbitrary number of strings. The idea is to compose the error message by passing each word as a separate string argument. The list of string arguments should be terminated by a null pointer to *char*:

```
extern void error(int ...);
extern char* itoa(int, char[]);        // see §6.6[17]

const char* Null_cp = 0;

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:
        error(0,argv[0],Null_cp);
        break;

    case 2:
        error(0,argv[0],argv[1],Null_cp);
        break;

    default:
        char buffer[8];
        error(1,argv[0], "with",itoa(argc-1,buffer),"arguments", Null_cp);
    }
    // ...
}
```

The function *itoa*( ) returns the character string representing its integer argument.

Note that using the integer *0* as the terminator would not have been portable: on some implementations, the integer zero and the null pointer do not have the same representation. This illustrates the subtleties and extra work that face the programmer once type checking has been suppressed using the ellipsis.

The error function could be defined like this:

```
void error(int severity ...) // "severity" followed by a zero-terminated list of char*s
{
    va_list ap;
    va_start(ap,severity);      // arg startup
```

```
        for ( ; ; ) {
                char* p = va_arg(ap, char*);
                if (p == 0) break;
                cerr << p << ´ ´;
        }

        va_end(ap);                     // arg cleanup

        cerr << ´\n´;
        if (severity) exit(severity);
}
```

First, a *va_list* is defined and initialized by a call of *va_start*( ). The macro *va_start* takes the name of the *va_list* and the name of the last formal argument as arguments. The macro *va_arg*( ) is used to pick the unnamed arguments in order. In each call, the programmer must supply a type; *va_arg*( ) assumes that an actual argument of that type has been passed, but it typically has no way of ensuring that. Before returning from a function in which *va_start*( ) has been used, *va_end*( ) must be called. The reason is that *va_start*( ) may modify the stack in such a way that a return cannot successfully be done; *va_end*( ) undoes any such modifications.

7.7 Pointer to Function [fct.pf]

There are only two things one can do to a function: call it and take its address. The pointer obtained by taking the address of a function can then be used to call the function. For example:

```
void error(string s) { /* ... */ }

void (*efct)(string);           // pointer to function

void f()
{
        efct = &error;          // efct points to error
        efct("error");          // call error through efct
}
```

The compiler will discover that *efct* is a pointer and call the function pointed to. That is, dereferencing of a pointer to function using * is optional. Similarly, using & to get the address of a function is optional:

```
void (*f1)(string) = &error;    // ok
void (*f2)(string) = error;     // also ok; same meaning as &error

void g()
{
        f1("Vasa");             // ok
        (*f1)("Mary Rose");     // also ok
}
```

Pointers to functions have argument types declared just like the functions themselves. In pointer assignments, the complete function type must match exactly. For example:

```
void (*pf)(string);   // pointer to void(string)
void f1(string);      // void(string)
int f2(string);       // int(string)
void f3(int*);        // void(int*)

void f()
{
    pf = &f1;                 // ok
    pf = &f2;                 // error: bad return type
    pf = &f3;                 // error: bad argument type

    pf("Hera");               // ok
    pf(1);                    // error: bad argument type

    int i = pf("Zeus");       // error: void assigned to int
}
```

The rules for argument passing are the same for calls directly to a function and for calls to a function through a pointer.

It is often convenient to define a name for a pointer-to-function type to avoid using the somewhat nonobvious declaration syntax all the time. Here is an example from a UNIX system header:

```
typedef void (*SIG_TYP)(int);        // from <signal.h>
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int, SIG_ARG_TYP);
```

An array of pointers to functions is often useful. For example, the menu system for my mouse-based editor is implemented using arrays of pointers to functions to represent operations. The system cannot be described in detail here, but this is the general idea:

```
typedef void (*PF)();

PF edit_ops[] = {          // edit operations
    &cut, &paste, &copy, &search
};

PF file_ops[] = {          // file management
    &open, &append, &close, &write
};
```

We can then define and initialize the pointers that control actions selected from a menu associated with the mouse buttons:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

In a complete implementation, more information is needed to define each menu item. For example, a string specifying the text to be displayed must be stored somewhere. As the system is used, the meaning of mouse buttons changes frequently with the context. Such changes are performed (partly) by changing the value of the button pointers. When a user selects a menu item, such as item 3 for button 2, the associated operation is executed:

```
button2[2]();  // call button2's 3rd function
```

One way to gain appreciation of the expressive power of pointers to functions is to try to write such code without them – and without using their better-behaved cousins, the virtual functions (§12.2.6). A menu can be modified at run-time by inserting new functions into the operator table. It is also easy to construct new menus at run-time.

Pointers to functions can be used to provide a simple form of polymorphic routines, that is, routines that can be applied to objects of many different types:

```
typedef int (*CFT)(const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
    Sort the "n" elements of vector "base" into increasing order
    using the comparison function pointed to by "cmp".
    The elements are of size "sz".

    Shell sort (Knuth, Vol3, pg84)
*/

{
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap) {
                char* b = static_cast<char*>(base); // necessary cast
                char* pj = b+j*sz;                  // &base[j]
                char* pjg = b+(j+gap)*sz;           // &base[j+gap]

                if (cmp(pj,pjg)<0) {                // swap base[j] and base[j+gap]:
                    for (int k=0; k<sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjg[k];
                        pjg[k] = temp;
                    }
                }
            }
}
```

The *ssort*( ) routine does not know the type of the objects it sorts, only the number of elements (the array size), the size of each element, and the function to call to perform a comparison. The type of *ssort*( ) was chosen to be the same as the type of the standard C library sort routine, *qsort*( ). Real programs use *qsort*( ), the C++ standard library algorithm *sort* (§18.7.1), or a specialized sort routine. This style of code is common in C, but it is not the most elegant way of expressing this algorithm in C++ (see §13.3, §13.5.2).

Such a sort function could be used to sort a table such as this:

```
struct User {
    char* name;
    char* id;
    int dept;
};
```

```
User heads[] = {
    "Ritchie D.M",      "dmr",    11271,
    "Sethi R.",         "ravi",   11272,
    "Szymanski T.G.",   "tgs",    11273,
    "Schryer N.L.",     "nls",    11274,
    "Schryer N.L.",     "nls",    11275,
    "Kernighan B.W.",   "bwk",    11276
};

void print_id(User* v, int n)
{
    for (int i=0; i<n; i++)
        cout << v[i].name << '\t' << v[i].id << '\t' << v[i].dept << '\n';
}
```

To be able to sort, we must first define appropriate comparison functions. A comparison function must return a negative value if its first argument is less than the second, zero if the arguments are equal, and a positive number otherwise:

```
int cmp1(const void* p, const void* q)    // Compare name strings
{
    return strcmp(static_cast<const User*>(p)->name, static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q)    // Compare dept numbers
{
    return static_cast<const User*>(p)->dept - static_cast<const User*>(q)->dept;
}
```

This program sorts and prints:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    ssort(heads, 6, sizeof(User), cmp1);
    print_id(heads, 6);
    cout << "\n";

    cout << "Heads in order of department number:\n";
    ssort(heads, 6, sizeof(User), cmp2);
    print_id(heads, 6);
}
```

You can take the address of an overloaded function by assigning to or initializing a pointer to function. In that case, the type of the target is used to select from the set of overloaded functions. For example:

```
void f(int);
int f(char);

void (*pf1)(int) = &f;      // void f(int)
int (*pf2)(char) = &f;      // int f(char)
void (*pf3)(char) = &f;     // error: no void f(char)
```

A function must be called through a pointer to function with exactly the right argument and return types. There is no implicit conversion of argument or return types when pointers to functions are assigned or initialized. This means that

```
int cmp3(const mytype*,const mytype*);
```

is not a suitable argument for *ssort*( ). The reason is that accepting *cmp3* as an argument to *ssort*( ) would violate the guarantee that *cmp3* will be called with arguments of type *mytype** (see also §9.2.5).

7.8 Macros [fct.macro]

Macros are very important in C but have far fewer uses in C++. The first rule about macros is: Don't use them unless you have to. Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer. Because they rearrange the program text before the compiler proper sees it, macros are also a major problem for many programming tools. So when you use macros, you should expect inferior service from tools such as debuggers, cross-reference tools, and profilers. If you must use macros, please read the reference manual for your own implementation of the C++ preprocessor carefully and try not to be too clever. Also to warn readers, follow the convention to name macros using lots of capital letters. The syntax of macros is presented in §A.11.

A simple macro is defined like this:

```
#define NAME rest of line
```

Where *NAME* is encountered as a token, it is replaced by *rest of line*. For example,

```
named = NAME
```

will expand into

```
named = rest of line
```

A macro can also be defined to take arguments. For example:

```
#define MAC(x,y) argument1: x argument2: y
```

When *MAC* is used, two argument strings must be presented. They will replace *x* and *y* when *MAC*( ) is expanded. For example,

```
expanded = MAC(foo bar, yuk yuk)
```

will be expanded into

```
expanded = argument1: foo bar argument2: yuk yuk
```

Macro names cannot be overloaded, and the macro preprocessor cannot handle recursive calls:

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c) /* trouble?: redefines, does not overload */

#define FAC(n) (n>1)?n*FAC(n-1):1          /* trouble: recursive macro */
```

Macros manipulate character strings and know little about C++ syntax and nothing about C++ types or scope rules. Only the expanded form of a macro is seen by the compiler, so an error in a macro will be reported when the macro is expanded, not when it is defined. This leads to very obscure error messages.

Here are some plausible macros:

```
#define CASE break;case
#define FOREVER for(;;)
```

Here are some completely unnecessary macros:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

Here are some dangerous macros:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

To see why they are dangerous, try expanding this:

```
int xx = 0;        // global counter

void f()
{
        int xx = 0;                  // local variable
        int y = SQUARE(xx+2);        // y=xx+2*xx+2; that is y=xx+(2*xx)+2
        INCR_xx;                     // increments local xx
}
```

If you must use a macro, use the scope resolution operator `::` when referring to global names (§4.9.4) and enclose occurrences of a macro argument name in parentheses whenever possible. For example:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

If you must write macros complicated enough to require comments, it is wise to use `/* */` comments because C preprocessors that do not know about `//` comments are sometimes used as part of C++ tools. For example:

```
#define M2(a) something(a)    /* thoughtful comment */
```

Using macros, you can design your own private language. Even if you prefer this ‘‘enhanced language’’ to plain C++, it will be incomprehensible to most C++ programmers. Furthermore, the C preprocessor is a very simple macro processor. When you try to do something nontrivial, you are likely to find it either impossible or unnecessarily hard to do. The ***const***, ***inline***, ***template***, and ***namespace*** mechanisms are intended as alternatives to many traditional uses of preprocessor constructs. For example:

```
const int answer = 42;
template<class T> inline T min(T a, T b) { return (a<b)?a:b; }
```

When writing a macro, it is not unusual to need a new name for something. A string can be created by concatenating two strings using the ## macro operator. For example,

```
#define NAME2(a,b) a##b

int NAME2(hack,cah)();
```

will produce

```
int hackcah();
```

for the compiler to read.

The directive

```
#undef X
```

ensures that no macro called ***X*** is defined – whether or not one was before the directive. This affords some protection against undesired macros. However, it is not always easy to know what the effects of ***X*** on a piece of code were supposed to be.

7.8.1 Conditional Compilation [fct.cond]

One use of macros is almost impossible to avoid. The directive ***#ifdef identifier*** conditionally causes all input to be ignored until a ***#endif*** directive is seen. For example,

```
int f(int a
#ifdef arg_two
,int b
#endif
);
```

produces

```
int f(int a
);
```

for the compiler to see unless a macro called ***arg_two*** has been ***#define***d. This example confuses tools that assume sane behavior from the programmer.

Most uses of ***#ifdef*** are less bizarre, and when used with restraint, ***#ifdef*** does little harm. See also §9.3.3.

Names of the macros used to control ***#ifdef*** should be chosen carefully so that they don't clash with ordinary identifiers. For example:

```
struct Call_info {
    Node* arg_one;
    Node* arg_two;
    // ...
};
```

This innocent-looking source text will cause some confusion should someone write:

```
#define arg_two x
```

Unfortunately, common and unavoidable headers contain many dangerous and unnecessary macros.

7.9 Advice [dcl.advice]

[1] Be suspicious of non-*const* reference arguments; if you want the function to modify its arguments, use pointers and value return instead; §5.5.
[2] Use *const* reference arguments when you need to minimize copying of arguments; §5.5.
[3] Use *const* extensively and consistently; §7.2.
[4] Avoid macros; §7.8.
[5] Avoid unspecified numbers of arguments; §7.6.
[6] Don't return pointers or references to local variables; §7.3.
[7] Use overloading when functions perform conceptually the same task on different types; §7.4.
[8] When overloading on integers, provide functions to eliminate common ambiguities; §7.4.3.
[9] When considering the use of a pointer to function, consider whether a virtual function (§2.5.5) or a template (§2.7.2) would be a better alternative; §7.7.
[10] If you must use macros, use ugly names with lots of capital letters; §7.8.

7.10 Exercises [fct.exercises]

1. (∗1) Write declarations for the following: a function taking arguments of type pointer to character and reference to integer and returning no value; a pointer to such a function; a function taking such a pointer as an argument; and a function returning such a pointer. Write the definition of a function that takes such a pointer as an argument and returns its argument as the return value. Hint: Use *typedef*.
2. (∗1) What does the following mean? What would it be good for?

```
typedef int (&rifii) (int, int);
```

3. (∗1.5) Write a program like ''Hello, world!'' that takes a name as a command-line argument and writes ''Hello, *name* !''. Modify this program to take any number of names as arguments and to say hello to each.
4. (∗1.5) Write a program that reads an arbitrary number of files whose names are given as command-line arguments and writes them one after another on *cout*. Because this program concatenates its arguments to produce its output, you might call it *cat*.
5. (∗2) Convert a small C program to C++. Modify the header files to declare all functions called and to declare the type of every argument. Where possible, replace *#define*s with *enum*, *const*, or *inline*. Remove *extern* declarations from *.c* files and if necessary convert all function definitions to C++ function definition syntax. Replace calls of *malloc*( ) and *free*( ) with *new* and *delete*. Remove unnecessary casts.
6. (∗2) Implement *ssort*( ) (§7.7) using a more efficient sorting algorithm. Hint: *qsort*( ).
7. (∗2.5) Consider:

```
struct Tnode {
    string word;
    int count;
    Tnode* left;
    Tnode* right;
};
```

Write a function for entering new words into a tree of ***Tnode***s. Write a function to write out a tree of ***Tnode***s. Write a function to write out a tree of ***Tnode***s with the words in alphabetical order. Modify ***Tnode*** so that it stores (only) a pointer to an arbitrarily long word stored as an array of characters on free store using ***new***. Modify the functions to use the new definition of ***Tnode***.

8. (∗2.5) Write a function to invert a two-dimensional array. Hint: §C.7.
9. (∗2) Write an encryption program that reads from ***cin*** and writes the encoded characters to ***cout***. You might use this simple encryption scheme: the encrypted form of a character ***c*** is ***c^key[i]***, where ***key*** is a string passed as a command-line argument. The program uses the characters in ***key*** in a cyclic manner until all the input has been read. Re-encrypting encoded text with the same key produces the original text. If no key (or a null string) is passed, then no encryption is done.
10. (∗3.5) Write a program to help decipher messages encrypted with the method described in §7.10[9] without knowing the key. Hint: See David Kahn: *The Codebreakers*, Macmillan, 1967, New York, pp. 207-213.
11. (∗3) Write an ***error*** function that takes a ***printf***-style format string containing ***%s***, ***%c***, and ***%d*** directives and an arbitrary number of arguments. Don't use ***printf()***. Look at §21.8 if you don't know the meaning of ***%s***, ***%c***, and ***%d***. Use ***<cstdarg>***.
12. (∗1) How would you choose names for pointer to function types defined using ***typedef***?
13. (∗2) Look at some programs to get an idea of the diversity of styles of names actually used. How are uppercase letters used? How is the underscore used? When are short names such as ***i*** and ***x*** used?
14. (∗1) What is wrong with these macro definitions?

```
#define PI = 3.141593;
#define MAX(a,b) a>b?a:b
#define fac(a) (a)*fac((a)-1)
```

15. (∗3) Write a macro processor that defines and expands simple macros (like the C preprocessor does). Read from ***cin*** and write to ***cout***. At first, don't try to handle macros with arguments. Hint: The desk calculator (§6.1) contains a symbol table and a lexical analyzer that you could modify.
16. (∗2) Implement ***print()*** from §7.5.
17. (∗2) Add functions such as ***sqrt()***, ***log()***, and ***sin()*** to the desk calculator from §6.1. Hint: Predefine the names and call the functions through an array of pointers to functions. Don't forget to check the arguments in a function call.
18. (∗1) Write a factorial function that does not use recursion. See also §11.14[6].
19. (∗2) Write functions to add one day, one month, and one year to a ***Date*** as defined in §5.9[13]. Write a function that gives the day of the week for a given ***Date***. Write a function that gives the ***Date*** of the first Monday following a given ***Date***.

8

Namespaces and Exceptions

The year is 787!
A.D.?
– Monty Python

No rule is so general,
which admits not some exception.
– Robert Burton

Modularity, interfaces, and exceptions — namespaces — ***using*** — ***using namespace*** — avoiding name clashes — name lookup — namespace composition — namespace aliases — namespaces and C code — exceptions — ***throw*** and ***catch*** — exceptions and program structure — advice — exercises.

8.1 Modularization and Interfaces [name.module]

Any realistic program consists of a number of separate parts. For example, even the simple ''Hello, world!'' program involves at least two parts: the user code requests ***Hello, world!*** to be printed, and the I/O system does the printing.

Consider the desk calculator example from §6.1. It can be viewed as being composed of five parts:

[1] The parser, doing syntax analysis
[2] The lexer, composing tokens out of characters
[3] The symbol table, holding (string,value) pairs
[4] The driver, ***main*( )**
[5] The error handler

This can be represented graphically:

where an arrow means ‘‘using.’’ To simplify the picture, I have not represented the fact that every part relies on error handling. In fact, the calculator was conceived as three parts, with the driver and error handler added for completeness.

When one module uses another, it doesn’t need to know everything about the module used. Ideally, most of the details of a module are unknown to its users. Consequently, we make a distinction between a module and its interface. For example, the parser directly relies on the lexer’s interface (only), rather than on the complete lexer. The lexer simply implements the services advertised in its interface. This can be presented graphically like this:

Dashed lines means ‘‘implements.’’ I consider this to be the real structure of the program, and our job as programmers is to represent this faithfully in code. That done, the code will be simple, efficient, comprehensible, maintainable, etc., because it will directly reflect our fundamental design.

The following sections show how the logical structure of the desk calculator program can be made clear, and §9.3 shows how the program source text can be physically organized to take advantage of it. The calculator is a tiny program, so in ‘‘real life’’ I wouldn’t bother using namespaces and separate compilation (§2.4.1, §9.1) to the extent I do here. It is simply used to present techniques useful for larger programs without our drowning in code. In real programs, each ‘‘module’’ represented by a separate namespace will often have hundreds of functions, classes, templates, etc.

To demonstrate a variety of techniques and language features, I develop the modularization of

the calculator in stages. In ''real life,'' a program is unlikely to grow through all of these stages. An experienced programmer might pick a design that is ''about right'' from the start. However, as a program evolves over the years, dramatic structural changes are not uncommon.

Error handling permeates the structure of a program. When breaking up a program into modules or (conversely) when composing a program out of modules, we must take care to minimize dependencies between modules caused by error handling. C++ provides exceptions to decouple the detection and reporting of errors from the handling of errors. Therefore, the discussion of how to represent modules as namespaces (§8.2) is followed by a demonstration of how we can use exceptions to further improve modularity (§8.3).

There are many more notions of modularity than the ones discussed in this chapter and the next. For example, we might use concurrently executing and communicating processes to represent important aspects of modularity. Similarly, the use of separate address spaces and the communication of information between address spaces are important topics not discussed here. I consider these notions of modularity largely independent and orthogonal. Interestingly, in each case, separating a system into modules is easy. The hard problem is to provide safe, convenient, and efficient communication across module boundaries.

8.2 Namespaces [name.namespace]

A namespace is a mechanism for expressing logical grouping. That is, if some declarations logically belong together according to some criteria, they can be put in a common namespace to express that fact. For example, the declarations of the parser from the desk calculator (§6.1.1) may be placed in a namespace *Parser*:

```
namespace Parser {
    double expr(bool);
    double prim(bool get) { /* ... */ }
    double term(bool get) { /* ... */ }
    double expr(bool get) { /* ... */ }
}
```

The function *expr*( ) must be declared first and then later defined to break the dependency loop described in §6.1.1.

The input part of the desk calculator could be also placed in its own namespace:

```
namespace Lexer {
    enum Token_value {
        NAME,       NUMBER,       END,
        PLUS='+',   MINUS='-',    MUL='*',    DIV='/',
        PRINT=';',  ASSIGN='=',   LP='(',     RP=')'
    };

    Token_value curr_tok;
    double number_value;
    string string_value;

    Token_value get_token() { /* ... */ }
}
```

This use of namespaces makes it reasonably obvious what the lexer and the parser provide to a user. However, had I included the source code for the functions, this structure would have been obscured. If function bodies are included in the declaration of a realistically-sized namespace, you typically have to wade through pages or screenfuls of information to find what services are offered, that is, to find the interface.

An alternative to relying on separately specified interfaces is to provide a tool that extracts an interface from a module that includes implementation details. I don't consider that a good solution. Specifying interfaces is a fundamental design activity (see §23.4.3.4), a module can provide different interfaces to different users, and often an interface is designed long before the implementation details are made concrete.

Here is a version of the *Parser* with the interface separated from the implementation:

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
}

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

Note that as a result of separating the implementation of the interface, each function now has exactly one declaration and one definition. Users will see only the interface containing declarations. The implementation – in this case, the function bodies – will be placed ''somewhere else'' where a user need not look.

As shown, a member can be declared within a namespace definition and defined later using the *namespace-name*::*member-name* notation.

Members of a namespace must be introduced using this notation:

```
namespace namespace-name {
    // declaration and definitions
}
```

We cannot declare a new member of a namespace outside a namespace definition using the qualifier syntax. For example:

```
void Parser::logical(bool);    // error: no logical() in Parser
```

The idea is to make it reasonably easy to find all names in a namespace declaration and also to catch errors such as misspellings and type mismatches. For example:

```
double Parser::trem(bool);    // error: no trem() in Parser
double Parser::prim(int);     // error: Parser::prim() takes a bool argument
```

A namespace is a scope. Thus, ''namespace'' is a very fundamental and relatively simple concept. The larger a program is, the more useful namespaces are to express logical separations of its parts. Ordinary local scopes, global scopes, and classes are namespaces (§C.10.3).

Ideally, every entity in a program belongs to some recognizable logical unit (''module''). Therefore, every declaration in a nontrivial program should ideally be in some namespace named to

indicate its logical role in the program. The exception is ***main*( )**, which must be global in order for the run-time environment to recognize it as special (§8.3.3).

8.2.1 Qualified Names [name.qualified]

A namespace is a scope. The usual scope rules hold for namespaces, so if a name is previously declared in the namespace or in an enclosing scope, it can be used without further fuss. A name from another namespace can be used when qualified by the name of its namespace. For example:

```
double Parser::term(bool get)              // note Parser:: qualification
{
    double left = prim(get);               // no qualification needed

    for ( ; ; )
        switch (Lexer::curr_tok) {         // note Lexer:: qualification
        case Lexer::MUL:                   // note Lexer:: qualification
            left *= prim(true);            // no qualification needed
        // ...
        }
    // ...
}
```

The ***Parser*** qualifier is necessary to state that this ***term*( )** is the one declared in ***Parser*** and not some unrelated global function. Because ***term*( )** is a member of ***Parser***, it need not use a qualifier for ***prim*( )**. However, had the ***Lexer*** qualifier not been present, ***curr_tok*** would have been considered undeclared because the members of namespace ***Lexer*** are not in scope from within the ***Parser*** namespace.

8.2.2 Using Declarations [name.using.dcl]

When a name is frequently used outside its namespace, it can be a bother to repeatedly qualify it with its namespace name. Consider:

```
double Parser::prim(bool get)          // handle primaries
{
    if (get) Lexer::get_token();

    switch (Lexer::curr_tok) {
    case Lexer::NUMBER:                // floating-point constant
        Lexer::get_token();
        return Lexer::number_value;

    case Lexer::NAME:
    {   double& v = table[Lexer::string_value];
        if (Lexer::get_token() == Lexer::ASSIGN) v = expr(true);
        return v;
    }

    case Lexer::MINUS:                 // unary minus
        return -prim(true);
```

```
        case Lexer::LP:
        {       double e = expr(true);
                if (Lexer::curr_tok != Lexer::RP) return Error::error(") expected");
                Lexer::get_token();            // eat ')'
                return e;
        }
        case Lexer::END:
                return 1;
        default:
                return Error::error("primary expected");
        }
}
```

The repeated qualification ***Lexer*** is tedious and distracting. This redundancy can be eliminated by a *using-declaration* to state in one place that the ***get_token*** used in this scope is ***Lexer***'s ***get_token***. For example:

```
double Parser::prim(bool get)          // handle primaries
{
        using Lexer::get_token;    // use Lexer's get_token
        using Lexer::curr_tok;     // use Lexer's curr_tok
        using Error::error;        // use Error's error

        if (get) get_token();

        switch (curr_tok) {
        case Lexer::NUMBER:            // floating-point constant
                get_token();
                return Lexer::number_value;
        case Lexer::NAME:
        {       double& v = table[Lexer::string_value];
                if (get_token() == Lexer::ASSIGN) v = expr(true);
                return v;
        }
        case Lexer::MINUS:             // unary minus
                return -prim(true);
        case Lexer::LP:
        {       double e = expr(true);
                if (curr_tok != Lexer::RP) return error(") expected");
                get_token();                   // eat ')'
                return e;
        }
        case Lexer::END:
                return 1;
        default:
                return error("primary expected");
        }
}
```

A *using-declaration* introduces a local synonym.

It is often a good idea to keep local synonyms as local as possible to avoid confusion.

However, all parser functions use similar sets of names from other modules. We can therefore place the *using-declaration*s in the ***Parser***'s namespace definition:

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);

    using Lexer::get_token;    // use Lexer's get_token
    using Lexer::curr_tok;     // use Lexer's curr_tok
    using Error::error;        // use Error's error
}
```

This allows us to simplify the ***Parser*** functions almost to our original version (§6.1.1):

```
double Parser::term(bool get)        // multiply and divide
{
    double left = prim(get);

    for (;;)
        switch (curr_tok) {
        case Lexer::MUL:
            left *= prim(true);
            break;

        case Lexer::DIV:
            if (double d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");

        default:
            return left;
        }
}
```

I could have introduced the token names into the ***Parser***'s namespace. However, I left them explicitly qualified as a reminder of ***Parser***'s dependency on ***Lexer***.

8.2.3 Using Directives [name.using.dir]

What if our aim were to simplify the ***Parser*** functions to be *exactly* our original versions? This would be a reasonable aim for a large program that was being converted to using namespaces from a previous version with less explicit modularity.

A *using-directive* makes names from a namespace available almost as if they had been declared outside their namespace (§8.2.8). For example:

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
```

```
        using namespace Lexer;    // make all names from Lexer available
        using namespace Error;    // make all names from Error available
}
```

This allows us to write *Parser*'s functions exactly as we originally did (§6.1.1):

```
double Parser::term(bool get)        // multiply and divide
{
        double left = prim(get);

        for (;;)
                switch (curr_tok) {                    // Lexer's curr_tok
                case MUL:                              // Lexer's MUL
                        left *= prim(true);
                        break;
                case DIV:                              // Lexer's DIV
                        if (double d = prim(true)) {
                                left /= d;
                                break;
                        }
                        return error("divide by 0");   // Error's error
                default:
                        return left;
                }
}
```

Global *using-directive*s are a tool for transition (§8.2.9) and are otherwise best avoided. In a namespace, a ***using-directive*** is a tool for namespace composition (§8.2.8). In a function (only), a ***using-directive*** can be safely used as a notational convenience (§8.3.3.1).

8.2.4 Multiple Interfaces [name.multi]

It should be clear that the namespace definition we evolved for ***Parser*** is not the interface that the ***Parser*** presents to its users. Instead, it is the set of declarations that is needed to write the individual parser functions conveniently. The ***Parser***'s interface to its users should be far simpler:

```
namespace Parser {
        double expr(bool);
}
```

Fortunately, the two *namespace-definition*s for ***Parser*** can coexist so that each can be used where it is most appropriate. We see the namespace ***Parser*** used to provide two things:

[1] The common environment for the functions implementing the parser

[2] The external interface offered by the parser to its users

Thus, the driver code, ***main*( )**, should see only:

```
namespace Parser {              // interface for users
        double expr(bool);
}
```

The functions implementing the parser should see whichever interface we decided on as the best for expressing those functions' shared environment. That is:

```
namespace Parser {                  // interface for implementers
    double prim(bool);
    double term(bool);
    double expr(bool);

    using Lexer::get_token;         // use Lexer's get_token
    using Lexer::curr_tok;          // use Lexer's curr_tok
    using Error::error;             // use Error's error
}
```

or graphically:

The arrows represent ''relies on the interface provided by'' relations.

Parser´ is the small interface offered to users. The name ***Parser´*** (Parser prime) is not a C++ identifier. It was chosen deliberately to indicate that this interface doesn't have a separate name in the program. The lack of a separate name need not lead to confusion because programmers naturally invent different and obvious names for the different interfaces and because the physical layout of the program (see §9.3.2) naturally provides separate (file) names.

The interface offered to implementers is larger than the interface offered to users. Had this interface been for a realistically-sized module in a real system, it would change more often than the interface seen by users. It is important that the users of a module (in this case, ***main***( ) using ***Parser***) are insulated from such changes.

We don't need to use two separate namespaces to express the two different interfaces, but if we wanted to, we could. Designing interfaces is one of the most fundamental design activities and one in which major benefits can be gained and lost. Consequently, it is worthwhile to consider what we are really trying to achieve and to discuss a number of alternatives.

Please keep in mind that the solution presented is the simplest of those we consider, and often the best. Its main weaknesses are that the two interfaces don't have separate names and that the compiler doesn't necessarily have sufficient information to check the consistency of the two definitions of the namespace. However, even though the compiler doesn't always get the opportunity to check the consistency, it usually does. Furthermore, the linker catches most errors missed by the compiler.

The solution presented here is the one I use for the discussion of physical modularity (§9.3) and the one I recommend in the absence of further logical constraints (see also §8.2.7).

8.2.4.1 Interface Design Alternatives [name.alternatives]

The purpose of interfaces is to minimize dependencies between different parts of a program. Minimal interfaces lead to systems that are easier to understand, have better data hiding properties, are easier to modify, and compile faster.

When dependencies are considered, it is important to remember that compilers and programmers tend to take a somewhat simple-minded approach to them: ''If a definition is in scope at point X, then anything written at point X depends on anything stated in that definition.'' Typically, things are not really that bad because most definitions are irrelevant to most code. Given the definitions we have used, consider:

```
namespace Parser {          // interface for implementers
    // ...
    double expr(bool);
    // ...
}

int main()
{
    // ...
    Parser::expr(false);
    // ...
}
```

The function *main*( ) depends on *Parser*::*expr*( ) only, but it takes time, brain power, computation, etc., to figure that out. Consequently, for realistically-sized programs people and compilation systems often play it safe and assume that where there might be a dependency, there is one. This is typically a perfectly reasonable approach.

Thus, our aim is to express our program so that the set of potential dependencies is reduced to the set of actual dependencies.

First, we try the obvious: define a user interface to the parser in terms of the implementer interface we already have:

```
namespace Parser {                  // interface for implementers
    // ...
    double expr(bool);
    // ...
}

namespace Parser_interface {            // interface for users
    using Parser::expr;
}
```

Clearly, users of *Parser_interface* depend only, and indirectly, on *Parser*::*expr*( ). However, a crude look at the dependency graph gives us this:

Now the ***driver*** appears vulnerable to any change in the ***Parser*** interface from which it was supposed to be insulated. Even this appearance of a dependency is undesirable, so we explicitly restrict ***Parser_interface***'s dependency on ***Parser*** by having only the relevant part of the implementer interface to parser (that was called ***Parser´*** earlier) in scope where we define ***Parser_interface***:

```
namespace Parser {          // interface for users
    double expr(bool);
}
namespace Parser_interface {    // separately named interface for users
    using Parser::expr;
}
```

or graphically:

To ensure the consistency of ***Parser*** and ***Parser´***, we again rely on the compilation system as a whole, rather than on just the compiler working on a single compilation unit. This solution differs from the one in §8.2.4 only by the extra namespace ***Parser_interface***. If we wanted to, we could give ***Parser_interface*** a concrete representation by giving it its own ***expr( )*** function:

```
namespace Parser_interface {
    double expr(bool);
}
```

Now ***Parser*** need not be in scope in order to define ***Parser_interface***. It needs to be in scope only where ***Parser_interface::expr( )*** is defined:

```
double Parser_interface::expr(bool get)
{
    return Parser::expr(get);
}
```

This last variant can be represented graphically like this:

Now all dependencies are minimized. Everything is concrete and properly named. However, for most problems I face, this solution is also massive overkill.

8.2.5 Avoiding Name Clashes [name.clash]

Namespaces are intended to express logical structure. The simplest such structure is the distinction between code written by one person vs. code written by someone else. This simple distinction can be of great practical importance.

When we use only a single global scope, it is unnecessarily difficult to compose a program out of separate parts. The problem is that the supposedly-separate parts each define the same names. When combined into the same program, these names clash. Consider:

```
// my.h:
    char f(char);
    int f(int);
    class String { /* ... */ };

// your.h:
    char f(char);
    double f(double);
    class String { /* ... */ };
```

Given these definitions, a third party cannot easily use both ***my.h*** and ***your.h***. The obvious solution is to wrap each set of declarations in its own namespace:

```
namespace My {
    char f(char);
    int f(int);
    class String { /* ... */ };
}

namespace Your {
    char f(char);
    double f(double);
    class String { /* ... */ };
}
```

Now we can use declarations from ***My*** and ***Your*** through explicit qualification (§8.2.1), *using-declarations* (§8.2.2), or *using-directives* (§8.2.3).

8.2.5.1 Unnamed Namespaces [name.unnamed]

It is often useful to wrap a set of declarations in a namespace simply to protect against the possibility of name clashes. That is, the aim is to preserve locality of code rather than to present an interface to users. For example:

```
#include "header.h"
namespace Mine {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

Since we don't want the name ***Mine*** to be known outside a local context, it simply becomes a bother to invent a redundant global name that might accidentally clash with someone else's names. In that case, we can simply leave the namespace without a name:

```
#include "header.h"
namespace {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

Clearly, there has to be some way of accessing members of an unnamed namespace from the outside. Consequently, an unnamed namespace has an implied *using-directive*. The previous declaration is equivalent to

```
namespace $$$ {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
using namespace $$$;
```

where $$$ is some name unique to the scope in which the namespace is defined. In particular, unnamed namespaces in different translation units are different. As desired, there is no way of naming a member of an unnamed namespace from another translation unit.

8.2.6 Name Lookup [name.koenig]

A function taking an argument of type ***T*** is more often than not defined in the same namespace as ***T***. Consequently, if a function isn't found in the context of its use, we look in the namespaces of its arguments. For example:

```
namespace Chrono {
    class Date { /* ... */ };

    bool operator==(const Date&, const std::string&);
```

```
        std::string format(const Date&);      // make string representation
        // ...
}

void f(Chrono::Date d, int i)
{
        std::string s = format(d);          // Chrono::format()
        std::string t = format(i);          // error: no format() in scope
}
```

This lookup rule saves the programmer a lot of typing compared to using explicit qualification, yet it doesn't pollute the namespace the way a *using-directive* (§8.2.3) can. It is especially useful for operator operands (§11.2.4) and template arguments (§C.13.8.4), where explicit qualification can be quite cumbersome.

Note that the namespace itself needs to be in scope and the function must be declared before it can be found and used.

Naturally, a function can take arguments from more than one namespace. For example:

```
void f(Chrono::Date d, std::string s)
{
        if (d == s) {
                // ...
        }
        else if (d == "August 4, 1914") {
                // ...
        }
}
```

In such cases, we look for the function in the scope of the call (as ever) and in the namespaces of every argument (including each argument's class and base classes) and do the usual overload resolution (§7.4) of all functions we find. In particular, for the call *d==s*, we look for *operator==* in the scope surrounding *f*( ), in the *std* namespace (where == is defined for *string*), and in the *Chrono* namespace. There is a *std*::*operator*==( ), but it doesn't take a *Date* argument, so we use *Chrono*::*operator*==( ), which does. See also §11.2.4.

When a class member invokes a function, other members of the same class and its base classes are preferred over functions potentially found based on the argument types (§11.2.4).

8.2.7 Namespace Aliases [name.alias]

If users give their namespaces short names, the names of different namespaces will clash:

```
namespace A { // short name, will clash (eventually)
        // ...
}

A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

However, long namespace names can be impractical in real code:

```
namespace American_Telephone_and_Telegraph {      // too long
    // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

This dilemma can be resolved by providing a short alias for a longer namespace name:

```
// use namespace alias to shorten names:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

Namespace aliases also allow a user to refer to ‘‘the library’’ and have a single declaration defining what library that really is. For example:

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Sibelius";
```

This can immensely simplify the task of replacing one version of a library with another. By using ***Lib*** rather than ***Foundation_library_v2r11*** directly, you can update to version ‘‘v3r02’’ by changing the initialization of the alias ***Lib*** and recompiling. The recompile will catch source level incompatibilities. On the other hand, overuse of aliases (of any kind) can lead to confusion.

8.2.8 Namespace Composition [name.compose]

Often, we want to compose an interface out of existing interfaces. For example:

```
namespace His_string {
    class String { /* ... */ };
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
    void fill(char);
    // ...
}

namespace Her_vector {
    template<class T> class Vector { /* ... */ };
    // ...
}

namespace My_lib {
    using namespace His_string;
    using namespace Her_vector;
    void my_fct(String&);
}
```

Given this, we can now write the program in terms of ***My_lib***:

```
void f()
{
    My_lib::String s = "Byron";    // finds My_lib::His_string::String
    // ...
}

using namespace My_lib;

void g(Vector<String>& vs)
{
    // ...
    my_fct(vs[5]);
    // ...
}
```

If an explicitly qualified name (such as ***My_lib::String***) isn't declared in the namespace mentioned, the compiler looks in namespaces mentioned in *using-directive*s (such as ***His_string***).

Only if we need to define something, do we need to know the real namespace of an entity:

```
void My_lib::fill()          // error: no fill() declared in My_lib
{
    // ...
}

void His_string::fill()      // ok: fill() declared in His_string
{
    // ...
}

void My_lib::my_fct(My_lib::Vector<My_lib::String>& v) // ok
{
    // ...
}
```

Ideally, a namespace should

[1] express a logically coherent set of features,

[2] not give users access to unrelated features, and

[3] not impose a significant notational burden on users.

The composition techniques presented here and in the following subsections – together with the ***#include*** mechanism (§9.2.1) – provide strong support for this.

8.2.8.1 Selection [name.select]

Occasionally, we want access to only a few names from a namespace. We could do that by writing a namespace declaration containing only those names we want. For example, we could declare a version of ***His_string*** that provided the ***String*** itself and the concatenation operator only:

```
namespace His_string {                    // part of His_string only
    class String { /* ... */ };
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
}
```

However, unless I am the designer or maintainer of ***His_string***, this can easily get messy. A change to the ‘‘real’’ definition of ***His_string*** will not be reflected in this declaration. Selection of features from a namespace is more explicitly made with *using-declaration*s:

```
namespace My_string {
    using His_string::String;
    using His_string::operator+;    // use any + from His_string
}
```

A *using-declaration* brings every declaration with a given name into scope. In particular, a single *using-declaration* can bring in every variant of an overloaded function.

In this way, if the maintainer of ***His_string*** adds a member function to ***String*** or an overloaded version of the concatenation operator, that change will automatically become available to users of ***My_string***. Conversely, if a feature is removed from ***His_string*** or has its interface changed, affected uses of ***My_string*** will be detected by the compiler (see also §15.2.2).

8.2.8.2 Composition and Selection [name.comp]

Combining composition (by *using-directive*s) with selection (by *using-declaration*s) yields the flexibility needed for most real-world examples. With these mechanisms, we can provide access to a variety of facilities in such a way that we resolve name clashes and ambiguities arising from their composition. For example:

```
namespace His_lib {
    class String { /* ... */ };
    template<class T> class Vector { /* ... */ };
    // ...
}

namespace Her_lib {
    template<class T> class Vector { /* ... */ };
    class String { /* ... */ };
    // ...
}

namespace My_lib {
    using namespace His_lib; // everything from His_lib
    using namespace Her_lib; // everything from Her_lib

    using His_lib::String;      // resolve potential clash in favor of His_lib
    using Her_lib::Vector;      // resolve potential clash in favor of Her_lib

    template<class T> class List { /* ... */ }; // additional stuff
    // ...
}
```

When looking into a namespace, names explicitly declared there (including names declared by *using-declarations*) take priority over names made accessible in another scope by a *using-directive* (see also §C.10.1). Consequently, a user of ***My_lib*** will see the name clashes for ***String*** and ***Vector*** resolved in favor of ***His_lib*::*String*** and ***Her_lib*::*Vector***. Also, ***My_lib*::*List*** will be used by default independently of whether ***His_lib*** or ***Her_lib*** are providing a ***List***.

Usually, I prefer to leave a name unchanged when including it into a new namespace. In that way, I don't have to remember two different names for the same entity. However, sometimes a new name is needed or simply nice to have. For example:

```
namespace Lib2 {
    using namespace His_lib; // everything from His_lib
    using namespace Her_lib; // everything from Her_lib

    using His_lib::String;     // resolve potential clash in favor of His_lib
    using Her_lib::Vector;     // resolve potential clash in favor of Her_lib

    typedef Her_lib::String Her_string;        // rename

    template<class T> class His_vec            // ''rename''
        : public His_lib::Vector<T> { /* ... */ };

    template<class T> class List { /* ... */ }; // additional stuff
    // ...
}
```

There is no specific language mechanism for renaming. Instead, the general mechanisms for defining new entities are used.

8.2.9 Namespaces and Old Code [name.get]

Millions of lines of C and C++ code rely on global names and existing libraries. How can we use namespaces to alleviate problems in such code? Redesigning existing code isn't always a viable option. Fortunately, it is possible to use C libraries as if they were defined in a namespace. However, this cannot be done for libraries written in C++ (§9.2.4). On the other hand, namespaces are designed so that they can be introduced with minimal disruption into an older C++ program.

8.2.9.1 Namespaces and C [name.c]

Consider the canonical first C program:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
```

Breaking this program wouldn't be a good idea. Making standard libraries special cases isn't a good idea either. Consequently, the language rules for namespaces are designed to make it relatively easy to take a program written without namespaces and turn it into a more explicitly structured one using namespaces. In fact, the calculator program (§6.1) is an example of this.

The *using-directive* is the key to achieving this. For example, the declarations of the standard C I/O facilities from the C header ***stdio.h*** are wrapped in a namespace like this:

```
// stdio.h:

    namespace std {
        // ...
        int printf(const char* ... );
        // ...
    }
    using namespace std;
```

This achieves backwards compatibility. Also, a new header file ***cstdio*** is defined for people who don't want the names implicitly available:

```
// cstdio:

    namespace std {
        // ...
        int printf(const char* ... );
        // ...
    }
```

C++ standard library implementers who worry about replication of declarations will, of course, define ***stdio.h*** by including ***cstdio***:

```
// stdio.h:

    #include<cstdio>
    using namespace std;
```

I consider nonlocal *using-directive*s primarily a transition tool. Most code referring to names from other namespaces can be expressed more clearly with explicit qualification and *using-declaration*s.

The relationship between namespaces and linkage is described in §9.2.4.

8.2.9.2 Namespaces and Overloading [name.over]

Overloading (§7.4) works across namespaces. This is essential to allow us to migrate existing libraries to use namespaces with minimal source code changes. For example:

```
// old A.h:

    void f(int);
    // ...

// old B.h:

    void f(char);
    // ...

// old user.c:

    #include "A.h"
    #include "B.h"
```

```
void g()
{
    f('a');    // calls the f() from B.h
}
```

This program can be upgraded to a version using namespaces without changing the actual code:

```
// new A.h:

    namespace A {
        void f(int);
        // ...
    }

// new B.h:

    namespace B {
        void f(char);
        // ...
    }

// new user.c:

    #include "A.h"
    #include "B.h"

    using namespace A;
    using namespace B;

    void g()
    {
        f('a');    // calls the f() from B.h
    }
```

Had we wanted to keep ***user.c*** completely unchanged, we would have placed the *using-directive*s in the header files.

8.2.9.3 Namespaces Are Open [name.open]

A namespace is open; that is, you can add names to it from several namespace declarations. For example:

```
namespace A {
    int f(); // now A has member f()
}

namespace A {
    int g(); // now A has two members, f() and g()
}
```

In this way, we can support large program fragments within a single namespace the way an older library or application lives within the single global namespace. To do this, we must distribute the namespace definition over several header and source code files. As shown by the calculator example (§8.2.4), the openness of namespaces allows us to present different interfaces to different kinds

of users by presenting different parts of a namespace. This openness is also an aid to transition. For example,

```
// my header:
    void f(); // my function
    // ...
    #include<stdio.h>
    int g(); // my function
    // ...
```

can be rewritten without reordering of the declarations:

```
// my header:

    namespace Mine {
        void f(); // my function
        // ...
    }

    #include<stdio.h>

    namespace Mine {
        int g(); // my function
        // ...
    }
```

When writing new code, I prefer to use many smaller namespaces (see §8.2.8) rather than putting really major pieces of code into a single namespace. However, that is often impractical when converting major pieces of software to use namespaces.

When defining a previously declared member of a namespace, it is safer to use the ***Mine*::** syntax than to re-open ***Mine***. For example:

```
void Mine::ff()     // error: no ff() declared in Mine
{
    // ...
}
```

A compiler catches this error. However, because new functions can be defined within a namespace, a compiler cannot catch the equivalent error in a re-opened namespace:

```
namespace Mine { // re-opening Mine to define functions

    void ff()  // oops! no ff() declared in Mine; ff() is added to Mine by this definition
    {
        // ...
    }

    // ...
}
```

The compiler has no way of knowing that you didn't want that new ***ff*( )**.

8.3 Exceptions [name.except]

When a program is composed of separate modules, and especially when those modules come from separately developed libraries, error handling needs to be separated into two distinct parts:

[1] The reporting of error conditions that cannot be resolved locally

[2] The handling of errors detected elsewhere

The author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors but cannot detect them – or else they would be handled in the user's code and not left for the library to find.

In the calculator example, we bypassed this problem by designing the program as a whole. By doing that, we could fit error handling into our overall framework. However, when we separate the logical parts of the calculator into separate namespaces, we see that every namespace depends on namespace ***Error*** (§8.2.2) and that the error handling in ***Error*** relies on every module behaving appropriately after an error. Let's assume that we don't have the freedom to design the calculator as a whole and don't want the tight coupling between ***Error*** and all other modules. Instead, assume that the parser, etc., are written without knowledge of how a driver might like to handle errors.

Even though ***error*( )** was very simple, it embodied a strategy for error handling:

```
namespace Error {
    int no_of_errors;

    double error(const char* s)
    {
        std::cerr << "error: " << s << '\n';
        no_of_errors++;
        return 1;
    }
}
```

The ***error*( )** function writes out an error message, supplies a default value that allows its caller to continue a computation, and keeps track of a simple error state. Importantly, every part of the program knows that ***error*( )** exists, how to call it, and what to expect from it. For a program composed of separately-developed libraries, that would be too much to assume.

Exceptions are C++'s means of separating error reporting from error handling. In this section, exceptions are briefly described in the context of their use in the calculator example. Chapter 14 provides a more extensive discussion of exceptions and their uses.

8.3.1 Throw and Catch [name.throw]

The notion of an *exception* is provided to help deal with error reporting. For example:

```
struct Range_error {
    int i;
    Range_error(int ii) { i = ii; }   // constructor (§2.5.2, §10.2.3)
};
```

```
char to_char(int i)
{
    if (i<numeric_limits<char>::min() || numeric_limits<char>::max()<i) // see §22.2
        throw Range_Error();
    return c;
}
```

The ***to_char()*** function either returns the ***char*** with the numeric value ***i*** or throws a ***Range_error***. The fundamental idea is that a function that finds a problem it cannot cope with *throws* an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle a problem can indicate that it is willing to *catch* exceptions of the type used to report the problem. For example, to call ***to_char()*** and catch the exception it might throw, we could write:

```
void g(int i)
{
    try {
        char c = to_char(i);
        // ...
    }
    catch (Range_error) {
        cerr << "oops\n";
    }
}
```

The construct

```
    catch ( /* ... */ ) {
        // ...
    }
```

is called an *exception handler*. It can be used only immediately after a block prefixed with the keyword ***try*** or immediately after another exception handler; ***catch*** is also a keyword. The parentheses contain a declaration that is used in a way similar to how a function argument declaration is used. That is, it specifies the type of the objects that can be caught by this handler and optionally names the object caught. For example, if we wanted to know the value of the ***Range_error*** thrown, we would provide a name for the argument to ***catch*** exactly the way we name function arguments. For example:

```
void h(int i)
{
    try {
        char c = to_char(i);
        // ...
    }
    catch (Range_error x) {
        cerr << "oops: to_char(" << x.i << ")\n";
    }
}
```

If any code in a *try-block* – or called from it – throws an exception, the try-block's handlers will be

examined. If the exception thrown is of a type specified for a handler, that handler is executed. If not, the exception handlers are ignored and the *try-block* acts just like an ordinary block.

Basically, C++ exception handling is a way to transfer control to designated code in a calling function. Where needed, some information about the error can be passed along to the caller. C programmers can think of exception handling as a well-behaved mechanism replacing ***setjmp***/***longjmp*** (§16.1.2). The important interaction between exception handling and classes is described in Chapter 14.

8.3.2 Discrimination of Exceptions [name.discrimination]

Typically, a program will have several different possible run-time errors. Such errors can be mapped into exceptions with distinct names. I prefer to define types with no other purpose than exception handling. This minimizes confusion about their purpose. In particular, I never use a built-in type, such as ***int***, as an exception. In a large program, I would have no effective way to find unrelated uses of ***int*** exceptions. Thus, I could never be sure that such other uses didn't interfere with my use.

Our calculator (§6.1) must handle two kinds of run-time errors: syntax errors and attempts to divide by zero. No values need to be passed to a handler from the code that detects an attempt to divide by zero, so zero divide can be represented by a simple empty type:

```
struct Zero_divide { };
```

On the other hand, a handler would most likely prefer to get an indication of what kind of syntax error occurred. Here, we pass a string along:

```
struct Syntax_error {
    const char* p;
    Syntax_error(const char* q) { p = q; }
};
```

For notational convenience, I added a constructor (§2.5.2, §10.2.3) to the ***struct***.

A user of the parser can discriminate between the two exceptions by adding handlers for both to a ***try*** block. Where needed, the appropriate handler will be entered. If we ''fall through the bottom'' of a handler, the execution continues at the end of the list of handlers:

```
try {
    // ...
    expr(false);
    // we get here if and only if expr() didn't cause an exception
    // ...
}

catch (Syntax_error) {
    // handle syntax error
}
```

```
catch (Zero_divide) {
        // handle divide by zero
}
// we get here if expr didn't cause an exception or if a Syntax_error
// or Zero_divide exception was caught (and its handler didn't return,
// throw an exception, or in some other way alter the flow of control).
```

A list of handlers looks a bit like a ***switch*** statement, but there is no need for ***break*** statements. The syntax of a list of handlers differs from the syntax of a list of cases partly for that reason and partly to indicate that each handler is a scope (§4.9.4).

A function need not catch all possible exceptions. For example, the previous *try-block* didn't try to catch exceptions potentially generated by the parser's input operations. Those exceptions simply ''pass through,'' searching for a caller with an appropriate handler.

From the language's point of view, an exception is considered handled immediately upon entry into its handler so that any exceptions thrown while executing a handler must be dealt with by the callers of the *try-block*. For example, this does not cause an infinite loop:

```
class input_overflow { /* ... */ };

void f()
{
        try {
                // ...
        }
        catch (input_overflow) {
                // ...
                throw input_overflow();
        }
}
```

Exception handlers can be nested. For example:

```
class XXII { /* ... */ };

void f()
{
        // ...
        try {
                // ...
        }
        catch (XXII) {
                try {
                        // something complicated
                }
                catch (XXII) {
                        // complicated handler code failed
                }
        }
        // ...
}
```

However, such nesting is rare in human-written code and is more often than not an indication of poor style.

8.3.3 Exceptions in the Calculator [name.calc]

Given the basic exception-handling mechanism, we can rework the calculator example from §6.1 to separate the handling of errors found at run-time from the main logic of the calculator. This will result in an organization of the program that more realistically matches what is found in programs built from separate, loosely connected parts.

First, *error*( ) can be eliminated. Instead, the parser functions know only the types used to signal errors:

```
namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}
```

The parser detects three syntax errors:

```
Token_value Lexer::get_token()
{
    using namespace std;        // to use cin, isalpha(), etc.

        // ...

        default:                // NAME, NAME =, or error
            if (isalpha(ch)) {
                cin.putback(ch);
                cin >> string_value;
                return curr_tok=NAME;
            }
            throw Error::Syntax_error("bad token");
    }
}

double Parser::prim(bool get)       // handle primaries
{
    // ...

    case Lexer::LP:
    {   double e = expr(true);
        if (curr_tok != Lexer::RP) throw Error::Syntax_error("')' expected");
        get_token();            // eat ')'
        return e;
    }
```

```
        case Lexer::END:
            return 1;
        default:
            throw Error::Syntax_error("primary expected");
        }
    }
```

When a syntax error is detected, ***throw*** is used to transfer control to a handler defined in some (direct or indirect) caller. The ***throw*** operator also passes a value to the handler. For example,

```
throw Syntax_error("primary expected");
```

passes a ***Syntax_error*** object containing a pointer to the string ***primary expected*** to the handler.

Reporting a divide-by-zero error doesn't require any data to be passed along:

```
double Parser::term(bool get)        // multiply and divide
{
    // ...
    case Lexer::DIV:
        if (double d = prim(true)) {
            left /= d;
            break;
        }
        throw Error::Zero_divide();

    // ...
}
```

The driver can now be defined to handle ***Zero_divide*** and ***Syntax_error*** exceptions. For example:

```
int main(int argc, char* argv[])
{
    // ...
    while (*input) {
        try {
            Lexer::get_token();
            if (Lexer::curr_tok == Lexer::END) break;
            if (Lexer::curr_tok == Lexer::PRINT) continue;
            cout << Parser::expr(false) << '\n';
        }
        catch(Error::Zero_divide) {
            cerr << "attempt to divide by zero\n";
            skip();
        }
        catch(Error::Syntax_error e) {
            cerr << "syntax error:" << e.p << "\n";
            skip();
        }
    }

    if (input != &cin) delete input;
    return no_of_errors;
}
```

The function ***skip*( )** tries to bring the parser into a well-defined state after an error by skipping tokens until it finds an end-of-line or a semicolon. It, ***no_of_errors***, and ***input*** are obvious candidates for a ***Driver*** namespace:

```
namespace Driver {
    int no_of_errors;
    std::istream* input;
    void skip();
}

void Driver::skip()
{
    no_of_errors++;

    while (*input) {
        char ch;
        input->get(ch);

        switch (ch) {
        case '\n':
        case ';':
            input->get(ch);
            return;
        }
    }
}
```

The code for ***skip*( )** is deliberately written at a lower level of abstraction than the parser code so as to avoid being caught by exceptions from the parser while handling parser exceptions.

I retained the idea of counting the number of errors and reporting that number as the program's return value. It is often useful to know if a program encountered an error even if it was able to recover from it.

I did not put ***main*( )** in the ***Driver*** namespace. The global ***main*( )** is the initial function of a program (§3.2); a ***main*( )** in another namespace has no special meaning.

8.3.3.1 Alternative Error-Handling Strategies [name.strategy]

The original error-handling code was shorter and more elegant than the version using exceptions. However, it achieved that elegance by tightly coupling all parts of the program. That approach doesn't scale well to programs composed of separately developed libraries.

We could consider eliminating the separate error-handling function ***skip*( )** by introducing a state variable in ***main*( )**. For example:

```
int main(int argc, char* argv[])    // example of poor style
{
    // ...

    bool in_error = false;
```

```
        while (*Driver::input) {
            try {
                Lexer::get_token();
                if (Lexer::curr_tok == Lexer::END) break;
                if (Lexer::curr_tok == Lexer::PRINT) {
                    in_error = false;
                    continue;
                }
                if (in_error == false) cout << Parser::expr(false) << '\n';
            }

            catch(Error::Zero_divide) {
                cerr << "attempt to divide by zero\n";
                in_error = true;
            }
            catch(Error::Syntax_error e) {
                cerr << "syntax error:" << e.p << "\n";
                in_error = true;
            }
        }

        if (Driver::input != std::cin) delete Driver::input;
        return Driver::no_of_errors;
    }
```

I consider this a bad idea for several reasons:

[1] State variables are a common source of confusion and errors, especially if they are allowed to proliferate and affect larger sections of a program. In particular, I consider the version of *main*( ) using *in_error* less readable than the version using *skip*( ).

[2] It is generally a good strategy to keep error handling and ''normal'' code separate.

[3] Doing error handling using the same level of abstraction as the code that caused the error is hazardous; the error-handling code might repeat the same error that triggered the error handling in the first place. I leave it as an exercise to find how that can happen for the version of *main*( ) using *in_error* (§8.5[7]).

[4] It is more work to modify the ''normal'' code to add error-handling code than to add separate error-handling routines.

Exception handling is intended for dealing with nonlocal problems. If an error can be handled locally, it almost always should be. For example, there is no reason to use an exception to handle the too-many-arguments error:

```
    int main(int argc, char* argv[])
    {
        using namespace std;
        using namespace Driver;

        switch (argc) {
        case 1:                                  // read from standard input
            input = &cin;
            break;
```

```
        case 2:                                    // read argument string
            input = new istringstream(argv[1]);
            break;
        default:
            cerr << "too many arguments\n";
            return 1;
        }
        // as before
    }
```

Exceptions are discussed further in Chapter 14.

8.4 Advice [name.advice]

[1] Use namespaces to express logical structure; §8.2.
[2] Place every nonlocal name, except ***main***( ), in some namespace; §8.2.
[3] Design a namespace so that you can conveniently use it without accidentally gaining access to unrelated namespaces; §8.2.4.
[4] Avoid very short names for namespaces; §8.2.7.
[5] If necessary, use namespace aliases to abbreviate long namespaces names; §8.2.7.
[6] Avoid placing heavy notational burdens on users of your namespaces; §8.2.2, §8.2.3.
[7] Use the ***Namespace*** : : ***member*** notation when defining namespace members; §8.2.8.
[8] Use ***using namespace*** only for transition or within a local scope; §8.2.9.
[9] Use exceptions to decouple the treatment of ‘‘errors’’ from the code dealing with the ordinary processing; §8.3.3.
[10] Use user-defined rather than built-in types as exceptions; §8.3.2.
[11] Don’t use exceptions when local control structures are sufficient; §8.3.3.1.

8.5 Exercises [name.exercises]

1. (∗2.5) Write a doubly-linked list of ***string*** module in the style of the ***Stack*** module from §2.4. Exercise it by creating a list of names of programming languages. Provide a ***sort***( ) function for that list, and provide a function that reverses the order of the strings in it.
2. (∗2) Take some not-too-large program that uses at least one library that does not use namespaces and modify it to use a namespace for that library. Hint: §8.2.9.
3. (∗2) Modify the desk calculator program into a module in the style of §2.4 using namespaces. Don’t use any global *using-directive*s. Keep a record of the mistakes you made. Suggest ways of avoiding such mistakes in the future.
4. (∗1) Write a program that throws an exception in one function and catches it in another.
5. (∗2) Write a program consisting of functions calling each other to a calling depth of 10. Give each function an argument that determines at which level an exception is thrown. Have ***main***( ) catch these exceptions and print out which exception is caught. Don’t forget the case in which an exception is caught in the function that throws it.

6. (∗2) Modify the program from §8.5[5] to measure if there is a difference in the cost of catching exceptions depending on where in a class stack the exception is thrown. Add a string object to each function and measure again.
7. (∗1) Find the error in the first version of ***main***( ) in §8.3.3.1.
8. (∗2) Write a function that either returns a value or that throws that value based on an argument. Measure the difference in run-time between the two ways.
9. (∗2) Modify the calculator version from §8.5[3] to use exceptions. Keep a record of the mistakes you make. Suggest ways of avoiding such mistakes in the future.
10. (∗2.5) Write ***plus***( ), ***minus***( ), ***multiply***( ), and ***divide***( ) functions that check for possible overflow and underflow and that throw exceptions if such errors happen.
11. (∗2) Modify the calculator to use the functions from §8.5[10].

9

Source Files and Programs

Form must follow function.
– Le Corbusier

Separate compilation — linking — header files — standard library headers — the one-definition rule — linkage to non-C++ code — linkage and pointers to functions — using headers to express modularity — single-header organization — multiple-header organization — include guards — programs — advice — exercises.

9.1 Separate Compilation [file.separate]

A file is the traditional unit of storage (in a file system) and the traditional unit of compilation. There are systems that do not store, compile, and present C++ programs to the programmer as sets of files. However, the discussion here will concentrate on systems that employ the traditional use of files.

Having a complete program in one file is usually impossible. In particular, the code for the standard libraries and the operating system is typically not supplied in source form as part of a user's program. For realistically-sized applications, even having all of the user's own code in a single file is both impractical and inconvenient. The way a program is organized into files can help emphasize its logical structure, help a human reader understand the program, and help the compiler to enforce that logical structure. Where the unit of compilation is a file, all of a file must be recompiled whenever a change (however small) has been made to it or to something on which it depends. For even a moderately sized program, the amount of time spent recompiling can be significantly reduced by partitioning the program into files of suitable size.

A user presents a *source file* to the compiler. The file is then preprocessed; that is, macro processing (§7.8) is done and ***#include*** directives bring in headers (§2.4.1, §9.2.1). The result of preprocessing is called a *translation unit*. This unit is what the compiler proper works on and what the C++ language rules describe. In this book, I differentiate between source file and translation unit

only where necessary to distinguish what the programmer sees from what the compiler considers.

To enable separate compilation, the programmer must supply declarations providing the type information needed to analyze a translation unit in isolation from the rest of the program. The declarations in a program consisting of many separately compiled parts must be consistent in exactly the same way the declarations in a program consisting of a single source file must be. Your system will have tools to help ensure this. In particular, the linker can detect many kinds of inconsistencies. The *linker* is the program that binds together the separately compiled parts. A linker is sometimes (confusingly) called a *loader*. Linking can be done completely before a program starts to run. Alternatively, new code can be added to the program (''dynamically linked'') later.

The organization of a program into source files is commonly called the *physical structure* of a program. The physical separation of a program into separate files should be guided by the logical structure of the program. The same dependency concerns that guide the composition of programs out of namespaces guide its composition into source files. However, the logical and physical structure of a program need not be identical. For example, it can be useful to use several source files to store the functions from a single namespace, to store a collection of namespace definitions in a single file, and to scatter the definition of a namespace over several files (§8.2.4).

Here, we will first consider some technicalities relating to linking and then discuss two ways of breaking the desk calculator (§6.1, §8.2) into files.

9.2 Linkage [file.link]

Names of functions, classes, templates, variables, namespaces, enumerations, and enumerators must be used consistently across all translation units unless they are explicitly specified to be local.

It is the programmer's task to ensure that every namespace, class, function, etc. is properly declared in every translation unit in which it appears and that all declarations referring to the same entity are consistent. For example, consider two files:

```
// file1.c:
        int x = 1;
        int f() { /* do something */ }

// file2.c:
        extern int x;
        int f();
        void g() { x = f(); }
```

The *x* and *f*( ) used by *g*( ) in *file2.c* are the ones defined in *file1.c*. The keyword *extern* indicates that the declaration of *x* in *file2.c* is (just) a declaration and not a definition (§4.9). Had *x* been initialized, *extern* would simply be ignored because a declaration with an initializer is always a definition. An object must be defined exactly once in a program. It may be declared many times, but the types must agree exactly. For example:

```
// file1.c:
        int x = 1;
        int b = 1;
        extern int c;
```

```
// file2.c:
    int x;                  // meaning int x = 0;
    extern double b;
    extern int c;
```

There are three errors here: ***x*** is defined twice, ***b*** is declared twice with different types, and ***c*** is declared twice but not defined. These kinds of errors (linkage errors) cannot be detected by a compiler that looks at only one file at a time. Most, however, are detectable by the linker. Note that a variable defined without an initializer in the global or a namespace scope is initialized by default. This is *not* the case for local variables (§4.9.5, §10.4.2) or objects created on the free store (§6.2.6). For example, the following program fragment contains two errors:

```
// file1.c:
    int x;
    int f() { return x; }

// file2.c:
    int x;
    int g() { return f(); }
```

The call of ***f*( )** in ***file2.c*** is an error because ***f*( )** has not been declared in ***file2.c***. Also, the program will not link because ***x*** is defined twice. Note that these are not errors in C (§B.2.2).

A name that can be used in translation units different from the one in which it was defined is said to have *external linkage*. All the names in the previous examples have external linkage. A name that can be referred to only in the translation unit in which it is defined is said to have *internal linkage*.

An ***inline*** function (§7.1.1, §10.2.9) must be defined – by identical definitions (§9.2.3) – in every translation unit in which it is used. Consequently, the following example isn't just bad taste; it is illegal:

```
// file1.c:
    inline int f(int i) { return i; }

// file2.c:
    inline int f(int i) { return i+1; }
```

Unfortunately, this error is hard for an implementation to catch, and the following – otherwise perfectly logical – combination of external linkage and inlining is banned to make life simpler for compiler writers:

```
// file1.c:
    extern inline int g(int i);
    int h(int i) { return g(i); }    // error: g() undefined in this translation unit

// file2.c:
    extern inline int g(int i) { return i+1; }
```

By default, ***const***s (§5.4) and ***typedef***s (§4.9.7) have internal linkage. Consequently, this example is legal (although potentially confusing):

```
// file1.c:
        typedef int T;
        const int x = 7;

// file2.c:
        typedef void T;
        const int x = 8;
```

Global variables that are local to a single compilation unit are a common source of confusion and are best avoided. To ensure consistency, you should usually place global ***const***s and ***inline***s in header files only (§9.2.1).

A ***const*** can be given external linkage by an explicit declaration:

```
// file1.c:
        extern const int a = 77;

// file2.c:
        extern const int a;

        void g()
        {
                cout << a << '\n';
        }
```

Here, ***g()*** will print *77*.

An unnamed namespace (§8.2.5) can be used to make names local to a compilation unit. The effect of an unnamed namespace is very similar to that of internal linkage. For example:

```
// file 1.c:
        namespace {
                class X { /* ... */ };
                void f();
                int i;
                // ...
        }

// file2.c:
        class X { /* ... */ };
        void f();
        int i;
        // ...
```

The function ***f()*** in ***file1.c*** is not the same function as the ***f()*** in ***file2.c***. Having a name local to a translation unit and also using that same name elsewhere for an entity with external linkage is asking for trouble.

In C and older C++ programs, the keyword ***static*** is (confusingly) used to mean ''use internal linkage'' (§B.2.3). Don't use ***static*** except inside functions (§7.1.2) and classes (§10.2.4).

9.2.1 Header Files [file.header]

The types in all declarations of the same object, function, class, etc., must be consistent. Consequently, the source code submitted to the compiler and later linked together must be consistent. One imperfect but simple method of achieving consistency for declarations in different translation units is to ***#include*** *header files* containing interface information in source files containing executable code and/or data definitions.

The ***#include*** mechanism is a text manipulation facility for gathering source program fragments together into a single unit (file) for compilation. The directive

```
#include "to_be_included"
```

replaces the line in which the ***#include*** appears with the contents of the file ***to_be_included***. The content should be C++ source text because the compiler will proceed to read it.

To include standard library headers, use the angle brackets < and > around the name instead of quotes. For example:

```
#include <iostream>        // from standard include directory
#include "myheader.h"      // from current directory
```

Unfortunately, spaces are significant within the < > or " " of an include directive:

```
#include <  iostream  >     // will not find <iostream>
```

It may seem extravagant to recompile a file each time it is included somewhere, but the included files typically contain only declarations and not code needing extensive analysis by the compiler. Furthermore, most modern C++ implementations provide some form of precompiling of header files to minimize the work needed to handle repeated compilation of the same header.

As a rule of thumb, a header may contain:

Named namespaces	`namespace N { /* ... */ }`
Type definitions	`struct Point { int x, y; };`
Template declarations	`template<class T> class Z;`
Template definitions	`template<class T> class V { /* ... */ };`
Function declarations	`extern int strlen(const char*);`
Inline function definitions	`inline char get(char* p) { return *p++; }`
Data declarations	`extern int a;`
Constant definitions	`const float pi = 3.141593;`
Enumerations	`enum Light { red, yellow, green };`
Name declarations	`class Matrix;`
Include directives	`#include <algorithm>`
Macro definitions	`#define VERSION 12`
Conditional compilation directives	`#ifdef __cplusplus`
Comments	`/* check for end of file */`

This rule of thumb for what may be placed in a header is not a language requirement. It is simply a reasonable way of using the ***#include*** mechanism to express the physical structure of a program. Conversely, a header should never contain:

Ordinary function definitions	`char get(char* p) { return *p++; }`
Data definitions	`int a;`
Aggregate definitions	`short tbl[] = { 1, 2, 3 };`
Unnamed namespaces	`namespace { /* ... */ }`
Exported template definitions	`export template<class T> f(T t) { /* ... */ }`

Header files are conventionally suffixed by `.h`, and files containing function or data definitions are suffixed by `.c`. They are therefore often referred to as ''.h files'' and ''.c files,'' respectively. Other conventions, such as `.C`, `.cxx`, `.cpp`, and `.cc`, are also found. The manual for your compiler will be quite specific about this issue.

The reason for recommending that the definition of simple constants, but not the definition of aggregates, be placed in header files is that it is hard for implementations to avoid replication of aggregates presented in several translation units. Furthermore, the simple cases are far more common and therefore more important for generating good code.

It is wise not to be too clever about the use of *#include*. My recommendation is to *#include* only complete declarations and definitions and to do so only in the global scope, in linkage specification blocks, and in namespace definitions when converting old code (§9.2.2). As usual, it is wise to avoid macro magic. One of my least favorite activities is tracking down an error caused by a name being macro-substituted into something completely different by a macro defined in an indirectly *#include*d header that I have never even heard of.

9.2.2 Standard Library Headers [file.std.header]

The facilities of the standard library are presented through a set of standard headers (§16.1.2). No suffix is needed for standard library headers; they are known to be headers because they are included using the *#include*`<...>` syntax rather than *#include*`"..."`. The absence of a `.h` suffix does not imply anything about how the header is stored. A header such as `<map>` may be stored as a text file called `map.h` in a standard directory. On the other hand, standard headers are not required to be stored in a conventional manner. An implementation is allowed to take advantage of knowledge of the standard library definition to optimize the standard library implementation and the way standard headers are handled. For example, an implementation might have knowledge of the standard math library (§22.3) built in and treat *#include*`<cmath>` as a switch that makes the standard math functions available without reading any file.

For each C standard-library header `<X.h>`, there is a corresponding standard C++ header `<cX>`. For example, *#include*`<cstdio>` provides what *#include*`<stdio.h>` does. A typical `stdio.h` will look something like this:

```
#ifdef __cplusplus          // for C++ compliers only (§9.2.4)
namespace std {             // the standard library is defined in namespace std (§8.2.9)

extern "C" {                // stdio functions have C linkage (§9.2.4)
#endif

    // ...
    int printf(const char* ...);
    // ...
```

```
#ifdef __cplusplus
}
}
using namespace std;        // make stdio available in global namespace
#endif
```

That is, the actual declarations are (most likely) shared, but linkage and namespace issues must be addressed to allow C and C++ to share a header.

9.2.3 The One-Definition Rule [file.odr]

A given class, enumeration, and template, etc., must be defined exactly once in a program.

From a practical point of view, this means that there must be exactly one definition of, say, a class residing in a single file somewhere. Unfortunately, the language rule cannot be that simple. For example, the definition of a class may be composed through macro expansion (ugh!), while a definition of a class may be textually included in two source files by ***#include*** directives (§9.2.1). Worse, a ''file'' isn't a concept that is part of the C and C++ language definitions; there exist implementations that do not store programs in source files.

Consequently, the rule in the standard that says that there must be a unique definition of a class, template, etc., is phrased in a somewhat more complicated and subtle manner. This rule is commonly referred to as ''the one-definition rule,'' the ODR. That is, two definitions of a class, template, or inline function are accepted as examples of the same unique definition if and only if

[1] they appear in different translation units, and

[2] they are token-for-token identical, and

[3] the meanings of those tokens are the same in both translation units.

For example:

```
// file1.c:
    struct S { int a; char b; };
    void f(S*);

// file2.c:
    struct S { int a; char b; };
    void f(S* p) { /* ... */ }
```

The ODR says that this example is valid and that ***S*** refers to the same class in both source files. However, it is unwise to write out a definition twice like that. Someone maintaining ***file2.c*** will naturally assume that the definition of ***S*** in ***file2.c*** is the only definition of ***S*** and so feel free to change it. This could introduce a hard-to-detect error.

The intent of the ODR is to allow inclusion of a class definition in different translation units from a common source file. For example:

```
// file s.h:
    struct S { int a; char b; };
    void f(S*);
```

```
// file1.c:
        #include "s.h"
        // use f() here

// file2.c:
        #include "s.h"
        void f(S* p) { /* ... */ }
```

or graphically:

Here are examples of the three ways of violating the ODR:

```
// file1.c:
        struct S1 { int a; char b; };

        struct S1 { int a; char b; };    // error: double definition
```

This is an error because a ***struct*** may not be defined twice in a single translation unit.

```
// file1.c:
        struct S2 { int a; char b; };

// file2.c:
        struct S2 { int a; char bb; }; // error
```

This is an error because ***S2*** is used to name classes that differ in a member name.

```
// file1.c:
        typedef int X;
        struct S3 { X a; char b; };

// file2.c:
        typedef char X;
        struct S3 { X a; char b; };     // error
```

Here the two definitions of ***S3*** are token-for-token identical, but the example is an error because the meaning of the name ***X*** has sneakily been made to differ in the two files.

Checking against inconsistent class definitions in separate translation units is beyond the ability of most C++ implementations. Consequently, declarations that violate the ODR can be a source of subtle errors. Unfortunately, the technique of placing shared definitions in headers and ***#including*** them doesn't protect against this last form of ODR violation. Local typedefs and macros can change the meaning of ***#include***d declarations:

```
// file s.h:
    struct S { Point a; char b; };

// file1.c:
    #define Point int
    #include "s.h"
    // ...

// file2.c:
    class Point { /* ... */ };
    #include "s.h"
    // ...
```

The best defense against this kind of hackery is to make headers as self-contained as possible. For example, if class ***Point*** had been declared in the ***s.h*** header the error would have been detected.

A template definition can be ***#include***d in several translation units as long as the ODR is adhered to. In addition, an exported template can be used given only a declaration:

```
// file1.c:
    export template<class T> T twice(T t) { return t+t; }

// file2.c:
    template<class T> T twice(T t);          // declaration
    int g(int i) { return twice(i); }
```

The keyword ***export*** means ‘‘accessible from another translation unit’’ (§13.7).

9.2.4 Linkage to Non-C++ Code [file.c]

Typically, a C++ program contains parts written in other languages. Similarly, it is common for C++ code fragments to be used as parts of programs written mainly in some other language. Cooperation can be difficult between program fragments written in different languages and even between fragments written in the same language but compiled with different compilers. For example, different languages and different implementations of the same language may differ in their use of machine registers to hold arguments, the layout of arguments put on a stack, the layout of built-in types such as strings and integers, the form of names passed by the compiler to the linker, and the amount of type checking required from the linker. To help, one can specify a *linkage* convention to be used in an ***extern*** declaration. For example, this declares the C and C++ standard library function ***strcpy*( )** and specifies that it should be linked according to the C linkage conventions:

```
extern "C" char* strcpy(char*, const char*);
```

The effect of this declaration differs from the effect of the ‘‘plain’’ declaration

```
extern char* strcpy(char*, const char*);
```

only in the linkage convention used for calling ***strcpy*( )**.

The ***extern "C"*** directive is particularly useful because of the close relationship between C and C++. Note that the ***C*** in ***extern "C"*** names a linkage convention and not a language. Often, ***extern "C"*** is used to link to Fortran and assembler routines that happen to conform to the conventions of a C implementation.

An ***extern "C"*** directive specifies the linkage convention (only) and does not affect the semantics of calls to the function. In particular, a function declared ***extern "C"*** still obeys the C++ type checking and argument conversion rules and not the weaker C rules. For example:

```
extern "C" int f();

int g()
{
    return f(1);    // error: no argument expected
}
```

Adding ***extern "C"*** to a lot of declarations can be a nuisance. Consequently, there is a mechanism to specify linkage to a group of declarations. For example:

```
extern "C" {
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...
}
```

This construct, commonly called a *linkage block*, can be used to enclose a complete C header to make a header suitable for C++ use. For example:

```
extern "C" {
#include <string.h>
}
```

This technique is commonly used to produce a C++ header from a C header. Alternatively, conditional compilation (§7.8.1) can be used to create a common C and C++ header:

```
#ifdef __cplusplus
extern "C" {
#endif

    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...

#ifdef __cplusplus
}
#endif
```

The predefined macro name ***__cplusplus*** is used to ensure that the C++ constructs are edited out when the file is used as a C header.

Any declaration can appear within a linkage block:

```
extern "C" {            // any declaration here, for example:
    int g1;             // definition
    extern int g2;      // declaration, not definition
}
```

In particular, the scope and storage class of variables are not affected, so ***g1*** is still a global variable

– and is still defined rather than just declared. To declare but not define a variable, you must apply the keyword ***extern*** directly in the declaration. For example:

```
extern "C" int g3;              // declaration, not definition
```

This looks odd at first glance. However, it is a simple consequence of keeping the meaning unchanged when adding ***"C"*** to an extern declaration and the meaning of a file unchanged when enclosing it in a linkage block.

A name with C linkage can be declared in a namespace. The namespace will affect the way the name is accessed in the C++ program, but not the way a linker sees it. The ***printf*( )** from ***std*** is a typical example:

```
#include<cstdio>

void f()
{
    std::printf("Hello, ");     // ok
    printf("world!\n");         // error: no global printf()
}
```

Even when called ***std*::*printf***, it is still the same old C ***printf*( )** (§21.8).

Note that this allows us to include libraries with C linkage into a namespace of our choice rather than polluting the global namespace. Unfortunately, the same flexibility is not available to us for headers defining functions with C++ linkage in the global namespace. The reason is that linkage of C++ entities must take namespaces into account so that the object files generated will reflect the use or lack of use of namespaces.

9.2.5 Linkage and Pointers to Functions [file.ptof]

When mixing C and C++ code fragments in one program, we sometimes want to pass pointers to functions defined in one language to functions defined in the other. If the two implementations of the two languages share linkage conventions and function-call mechanisms, such passing of pointers to functions is trivial. However, such commonality cannot in general be assumed, so care must be taken to ensure that a function is called the way it expects to be called.

When linkage is specified for a declaration, the specified linkage applies to all function types, function names, and variable names introduced by the declaration(s). This makes all kinds of strange – and occasionally essential – combinations of linkage possible. For example:

```
typedef int (*FT)(const void*, const void*);                    // FT has C++ linkage

extern "C" {
    typedef int (*CFT)(const void*, const void*);               // CFT has C linkage
    void qsort(void* p, size_t n, size_t sz, CFT cmp);          // cmp has C linkage
}

void isort(void* p, size_t n, size_t sz, FT cmp);               // cmp has C++ linkage
void xsort(void* p, size_t n, size_t sz, CFT cmp);              // cmp has C linkage
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp);    // cmp has C++ linkage

int compare(const void*, const void*);                          // compare() has C++ linkage
extern "C" int ccmp(const void*, const void*);                  // ccmp() has C linkage
```

```
void f(char* v, int sz)
{
    qsort(v,sz,1,&compare); // error
    qsort(v,sz,1,&ccmp);    // ok

    isort(v,sz,1,&compare); // ok
    isort(v,sz,1,&ccmp);    // error
}
```

An implementation in which C and C++ use the same calling conventions might accept the cases marked *error* as a language extension.

9.3 Using Header Files [file.using]

To illustrate the use of headers, I present a few alternative ways of expressing the physical structure of the calculator program (§6.1, §8.2).

9.3.1 Single Header File [file.single]

The simplest solution to the problem of partitioning a program into several files is to put the definitions in a suitable number of `.c` files and to declare the types needed for them to communicate in a single `.h` file that each `.c` file `#include`s. For the calculator program, we might use five `.c` files – *lexer.c*, *parser.c*, *table.c*, *error.c*, and *main.c* – to hold function and data definitions, plus the header *dc.h* to hold the declarations of every name used in more than one `.c` file.

The header *dc.h* would look like this:

```
// dc.h:

namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}

#include <string>

namespace Lexer {

    enum Token_value {
        NAME,       NUMBER,     END,
        PLUS='+',   MINUS='-',  MUL='*',    DIV='/',
        PRINT=';',  ASSIGN='=', LP='(',     RP=')'
    };

    extern Token_value curr_tok;
    extern double number_value;
    extern std::string string_value;
```

```
        Token_value get_token();
}

namespace Parser {
        double prim(bool get);    // handle primaries
        double term(bool get);    // multiply and divide
        double expr(bool get);    // add and subtract

        using Lexer::get_token;
        using Lexer::curr_tok;
}

#include <map>

extern std::map<std::string, double> table;

namespace Driver {
        extern int no_of_errors;
        extern std::istream* input;
        void skip();
}
```

The keyword *extern* is used for every declaration of a variable to ensure that multiple definitions do not occur as we *#include dc.h* in the various *.c* files. The corresponding definitions are found in the appropriate *.c* files.

Leaving out the actual code, *lexer.c* will look something like this:

```
// lexer.c:

#include "dc.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

Using headers in this manner ensures that every declaration in a header will at some point be included in the file containing its definition. For example, when compiling *lexer.c* the compiler will be presented with:

```
namespace Lexer {    // from dc.h
        // ...
        Token_value get_token();
}

// ...

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

This ensures that the compiler will detect any inconsistencies in the types specified for a name. For example, had *get_token()* been declared to return a *Token_value*, but defined to return an *int*, the compilation of *lexer.c* would have failed with a type-mismatch error. If a definition is missing,

the linker will catch the problem. If a declaration is missing, some *.c* file will fail to compile.

File *parser.c* will look like this:

```
// parser.c:

#include "dc.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

File *table.c* will look like this:

```
// table.c:

#include "dc.h"

std::map<std::string,double> table;
```

The symbol table is simply a variable of the standard library *map* type. This defines *table* to be global. In a realistically-sized program, this kind of minor pollution of the global namespace builds up and eventually causes problems. I left this sloppiness here simply to get an opportunity to warn against it.

Finally, file *main.c* will look like this:

```
// main.c:

#include "dc.h"
#include <sstream>

int Driver::no_of_errors = 0;
std::istream* Driver::input = 0;

void Driver::skip() { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

To be recognized as *the* *main()* of the program, *main()* must be a global function, so no namespace is used here.

The physical structure of the system can be presented like this:

Note that the headers on the top are all headers for standard library facilities. For many forms of program analysis, these libraries can be ignored because they are well known and stable. For tiny

programs, the structure can be simplified by moving all *#include* directives to the common header.

This single-header style of physical partitioning is most useful when the program is small and its parts are not intended to be used separately. Note that when namespaces are used, the logical structure of the program is still represented within *dc.h*. If namespaces are not used, the structure is obscured, although comments can be a help.

For larger programs, the single header file approach is unworkable in a conventional file-based development environment. A change to the common header forces recompilation of the whole program, and updates of that single header by several programmers are error-prone. Unless strong emphasis is placed on programming styles relying heavily on namespaces and classes, the logical structure deteriorates as the program grows.

9.3.2 Multiple Header Files [file.multi]

An alternative physical organization lets each logical module have its own header defining the facilities it provides. Each *.c* file then has a corresponding *.h* file specifying what it provides (its interface). Each *.c* file includes its own *.h* file and usually also other *.h* files that specify what it needs from other modules in order to implement the services advertised in the interface. This physical organization corresponds to the logical organization of a module. The interface for users is put into its *.h* file, the interface for implementers is put into a file suffixed *_impl.h*, and the module's definitions of functions, variables, etc. are placed in *.c* files. In this way, the parser is represented by three files. The parser's user interface is provided by *parser.h*:

```
// parser.h:

namespace Parser {          // interface for users
    double expr(bool get);
}
```

The shared environment for the functions implementing the parser is presented by *parser_impl.h*:

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

namespace Parser {          // interface for implementers
    double prim(bool get);
    double term(bool get);
    double expr(bool get);

    using Lexer::get_token;
    using Lexer::curr_tok;
}
```

The user's header *parser.h* is *#include*d to give the compiler a chance to check consistency (§9.3.1).

The functions implementing the parser are stored in *parser.c* together with *#include* directives for the headers that the *Parser* functions need:

```
// parser.c:

#include "parser_impl.h"
#include "table.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

Graphically, the parser and the driver's use of it look like this:

As intended, this is a rather close match to the logical structure described in §8.3.3. To simplify this structure, we could have ***#include***d ***table.h*** in ***parser_impl.h*** rather than in ***parser.c***. However, ***table.h*** is an example of something that is not necessary to express the shared context of the parser functions; it is needed only by their implementation. In fact, it is used by just one function, ***expr()***, so if we were really keen on minimizing dependencies we could place ***expr()*** in its own ***.c*** file and ***#include table.h*** there only:

Such elaboration is not appropriate except for larger modules. For realistically-sized modules, it is common to ***#include*** extra files where needed for individual functions. Furthermore, it is not uncommon to have more than one ***_impl.h***, since different subsets of the module's functions need different shared contexts.

Please note that the ***_impl.h*** notation is not a standard or even a common convention; it is simply the way I like to name things.

Why bother with this more complicated scheme of multiple header files? It clearly requires far less thought simply to throw every declaration into a single header, as was done for ***dc.h***.

The multiple-header organization scales to modules several magnitudes larger than our toy parser and to programs several magnitudes larger than our calculator. The fundamental reason for using this type of organization is that it provides a better localization of concerns. When analyzing

and modifying a large program, it is essential for a programmer to focus on a relatively small chunk of code. The multiple-header organization makes it easy to determine exactly what the parser code depends on and to ignore the rest of the program. The single-header approach forces us to look at every declaration used by any module and decide if it is relevant. The simple fact is that maintenance of code is invariably done with incomplete information and from a local perspective. The multiple-header organization allows us to work successfully ''from the inside out'' with only a local perspective. The single-header approach – like every other organization centered around a global repository of information – requires a top-down approach and will forever leave us wondering exactly what depends on what.

The better localization leads to less information needed to compile a module, and thus to faster compiles. The effect can be dramatic. I have seen compile times drop by a factor of ten as the result of a simple dependency analysis leading to a better use of headers.

9.3.2.1 Other Calculator Modules [file.multi.etc]

The remaining calculator modules can be organized similarly to the parser. However, those modules are so small that they don't require their own *_impl.h* files. Such files are needed only where a logical module consists of many functions that need a shared context.

The error handler was reduced to the set of exception types so that no *error.c* was needed:

```
// error.h:

namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}
```

The lexer provides a rather large and messy interface:

```
// lexer.h:

#include <string>

namespace Lexer {

    enum Token_value {
        NAME,       NUMBER,     END,
        PLUS='+',   MINUS='-',  MUL='*',    DIV='/',
        PRINT=';',  ASSIGN='=', LP='(',     RP=')'
    };

    extern Token_value curr_tok;
    extern double number_value;
    extern std::string string_value;

    Token_value get_token();
}
```

In addition to ***lexer.h***, the implementation of the lexer depends on ***error.h***, ***<iostream>***, and the functions determining the kinds of characters declared in ***<cctype>***:

```
// lexer.c:

#include "lexer.h"
#include "error.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

We could have factored out the ***#include*** statements for ***error.h*** as the ***Lexer***'s ***_impl.h*** file. However, I considered that excessive for this tiny program.

As usual, we ***#include*** the interface offered by the module – in this case, ***lexer.h*** – in the module's implementation to give the compiler a chance to check consistency.

The symbol table is essentially self-contained, although the standard library header ***<map>*** could drag in all kinds of interesting stuff to implement an efficient ***map*** template class:

```
// table.h:

#include <map>
#include <string>

extern std::map<std::string,double> table;
```

Because we assume that every header may be ***#include***d in several ***.c*** files, we must separate the declaration of ***table*** from its definition, even though the difference between ***table.c*** and ***table.h*** is the single keyword ***extern***:

```
// table.c:

#include "table.h"

std::map<std::string,double> table;
```

Basically, the driver depends on everything:

```
// main.c:

#include "parser.h"
#include "lexer.h"
#include "error.h"
#include "table.h"

namespace Driver {
    int no_of_errors;
    std::istream* input;
    void skip();
}
```

```
#include <sstream>

int main(int argc, char* argv[]) { /* ... */ }
```

Because the ***Driver*** namespace is used exclusively by ***main***( ), I placed it in ***main.c***. Alternatively, I could have factored it out as ***driver.h*** and #***include***d it.

For a larger system, it is usually worthwhile organizing things so that the driver has fewer direct dependencies. Often, is it also worth minimizing what is done in ***main***( ) by having ***main***( ) call a driver function placed in a separate source file. This is particularly important for code intended to be used as a library. Then, we cannot rely on code in ***main***( ) and must be prepared to be called from a variety of functions (§9.6[8]).

9.3.2.2 Use of Headers [file.multi.use]

The number of headers to use for a program is a function of many factors. Many of these factors have more to do with the way files are handled on your system than with C++. For example, if your editor does not have facilities for looking at several files at the same time, then using many headers becomes less attractive. Similarly, if opening and reading 20 files of 50 lines each is noticeably more time-consuming than reading a single file of 1000 lines, you might think twice before using the multiple-header style for a small project.

A word of caution: a dozen headers plus the standard headers for the program's execution environment (which can often be counted in the hundreds) are usually manageable. However, if you partition the declarations of a large program into the logically minimal-sized headers (putting each structure declaration in its own file, etc.), you can easily get an unmanageable mess of hundreds of files even for minor projects. I find that excessive.

For large projects, multiple headers are unavoidable. In such projects, hundreds of files (not counting standard headers) are the norm. The real confusion starts when they start to be counted in the thousands. At that scale, the basic techniques discussed here still apply, but their management becomes a Herculean task. Remember that for realistically-sized programs, the single-header style is not an option. Such programs will have multiple headers. The choice between the two styles of organization occurs (repeatedly) for the parts that make up the program.

The single-header style and the multiple-header style are not really alternatives to each other. They are complementary techniques that must be considered whenever a significant module is designed and must be reconsidered as a system evolves. It's crucial to remember that one interface doesn't serve all equally well. It is usually worthwhile to distinguish between the implementers' interface and the users' interface. In addition, many larger systems are structured so that providing a simple interface for the majority of users and a more extensive interface for expert users is a good idea. The expert users' interfaces (''complete interfaces'') tend to #***include*** many more features than the average user would ever want to know about. In fact, the average users' interface can often be identified by eliminating features that require the inclusion of headers that define facilities that would be unknown to the average user. The term ''average user'' is not derogatory. In the fields in which I don't *have* to be an expert, I strongly prefer to be an average user. In that way, I minimize hassles.

9.3.3 Include Guards [file.guards]

The idea of the multiple-header approach is to represent each logical module as a consistent, self-contained unit. Viewed from the program as a whole, many of the declarations needed to make each logical module complete are redundant. For larger programs, such redundancy can lead to errors, as a header containing class definitions or inline functions gets ***#include***d twice in the same compilation unit (§9.2.3).

We have two choices. We can

[1] reorganize our program to remove the redundancy, or

[2] find a way to allow repeated inclusion of headers.

The first approach – which led to the final version of the calculator – is tedious and impractical for realistically-sized programs. We also need that redundancy to make the individual parts of the program comprehensible in isolation.

The benefits of an analysis of redundant ***#include***s and the resulting simplifications of the program can be significant both from a logical point of view and by reducing compile times. However, it can rarely be complete, so some method of allowing redundant ***#include***s must be applied. Preferably, it must be applied systematically, since there is no way of knowing how thorough an analysis a user will find worthwhile.

The traditional solution is to insert *include guards* in headers. For example:

```
// error.h:

#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
        // ...
}

#endif      // CALC_ERROR_H
```

The contents of the file between the ***#ifndef*** and ***#endif*** are ignored by the compiler if ***CALC_ERROR_H*** is defined. Thus, the first time ***error.h*** is seen during a compilation, its contents are read and ***CALC_ERROR_H*** is given a value. Should the compiler be presented with ***error.h*** again during the compilation, the contents are ignored. This is a piece of macro hackery, but it works and it is pervasive in the C and C++ worlds. The standard headers all have include guards.

Header files are included in essentially arbitrary contexts, and there is no namespace protection against macro name clashes. Consequently, I choose rather long and ugly names as my include guards.

Once people get used to headers and include guards, they tend to include *lots* of headers directly and indirectly. Even with C++ implementations that optimize the processing of headers, this can be undesirable. It can cause unnecessarily long compile time, and it can bring ***lots*** of declarations and macros into scope. The latter might affect the meaning of the program in unpredictable and adverse ways. Headers should be included only when necessary.

9.4 Programs [file.programs]

A program is a collection of separately compiled units combined by a linker. Every function, object, type, etc., used in this collection must have a unique definition (§4.9, §9.2.3). The program must contain exactly one function called *main*( ) (§3.2). The main computation performed by the program starts with the invocation of *main*( ) and ends with a return from *main*( ). The *int* returned by *main*( ) is passed to whatever system invoked *main*( ) as the result of the program.

This simple story must be elaborated on for programs that contain global variables (§10.4.9) or that throw an uncaught exception (§14.7).

9.4.1 Initialization of Nonlocal Variables [file.nonlocal]

In principle, a variable defined outside any function (that is, global, namespace, and class *static* variables) is initialized before *main*( ) is invoked. Such nonlocal variables in a translation unit are initialized in their declaration order (§10.4.9). If such a variable has no explicit initializer, it is by default initialized to the default for its type (§10.4.2). The default initializer value for built-in types and enumerations is *0*. For example:

```
double x = 2;          // nonlocal variables
double y;
double sqx = sqrt(x+y);
```

Here, *x* and *y* are initialized before *sqx*, so *sqrt*(*2*) is called.

There is no guaranteed order of initialization of global variables in different translation units. Consequently, it is unwise to create order dependencies between initializers of global variables in different compilation units. In addition, it is not possible to catch an exception thrown by the initializer of a global variable (§14.7). It is generally best to minimize the use of global variables and in particular to limit the use of global variables requiring complicated initialization.

Several techniques exist for enforcing an order of initialization of global variables in different translation units. However, none are both portable and efficient. In particular, dynamically linked libraries do not coexist happily with global variables that have complicated dependencies.

Often, a function returning a reference is a good alternative to a global variable. For example:

```
int& use_count( )
{
    static int uc = 0;
    return uc;
}
```

A call *use_count*( ) now acts as a global variable except that it is initialized at its first use (§5.5). For example:

```
void f( )
{
    cout << ++use_count( );    // read and increment
    // ...
}
```

The initialization of nonlocal static variables is controlled by whatever mechanism an

implementation uses to start up a C++ program. This mechanism is guaranteed to work properly only if *main*( ) is executed. Consequently, one should avoid nonlocal variables that require run-time initialization in C++ code intended for execution as a fragment of a non-C++ program.

Note that variables initialized by constant expressions (§C.5) cannot depend on the value of objects from other translation units and do not require run-time initialization. Such variables are therefore safe to use in all cases.

9.4.1.1 Program Termination [file.termination]

A program can terminate in several ways:

- By returning from *main*( )
- By calling *exit*( )
- By calling *abort*( )
- By throwing an uncaught exception

In addition, there are a variety of ill-behaved and implementation-dependent ways of making a program crash.

If a program is terminated using the standard library function *exit*( ), the destructors for constructed static objects are called (§10.4.9, §10.2.4). However, if the program is terminated using the standard library function *abort*( ), they are not. Note that this implies that *exit*( ) does not terminate a program immediately. Calling *exit*( ) in a destructor may cause an infinite recursion. The type of *exit*( ) is

```
void exit(int);
```

Like the return value of *main*( ) (§3.2), *exit*( )'s argument is returned to ''the system'' as the value of the program. Zero indicates successful completion.

Calling *exit*( ) means that the local variables of the calling function and its callers will not have their destructors invoked. Throwing an exception and catching it ensures that local objects are properly destroyed (§14.4.7). Also, a call of *exit*( ) terminates the program without giving the caller of the function that called *exit*( ) a chance to deal with the problem. It is therefore often best to leave a context by throwing an exception and letting a handler decide what to do next.

The C (and C++) standard library function *atexit*( ) offers the possibility to have code executed at program termination. For example:

```
void my_cleanup();

void somewhere()
{
        if (atexit(&my_cleanup)==0) {
                // my_cleanup will be called at normal termination
        }
        else {
                // oops: too many atexit functions
        }
}
```

This strongly resembles the automatic invocation of destructors for global variables at program termination (§10.4.9, §10.2.4). Note that an argument to *atexit*( ) cannot take arguments or return a

result. Also, there is an implementation-defined limit to the number of atexit functions; ***atexit*( )** indicates when that limit is reached by returning a nonzero value. These limitations make ***atexit*( )** less useful than it appears at first glance.

The destructor of an object created before a call of ***atexit*(*f*)** will be invoked after ***f*** is invoked. The destructor of an object created after a call of ***atexit*(*f*)** will be invoked before ***f*** is invoked.

The ***exit*( )**, ***abort*( )**, and ***atexit*( )** functions are declared in ***<cstdlib>***.

9.5 Advice [file.advice]

[1] Use header files to represent interfaces and to emphasize logical structure; §9.1, §9.3.2.
[2] ***#include*** a header in the source file that implements its functions; §9.3.1.
[3] Don't define global entities with the same name and similar-but-different meanings in different translation units; §9.2.
[4] Avoid non-inline function definitions in headers; §9.2.1.
[5] Use ***#include*** only at global scope and in namespaces; §9.2.1.
[6] ***#include*** only complete declarations; §9.2.1.
[7] Use include guards; §9.3.3.
[8] ***#include*** C headers in namespaces to avoid global names; §9.3.2.
[9] Make headers self-contained; §9.2.3.
[10] Distinguish between users' interfaces and implementers' interfaces; §9.3.2.
[11] Distinguish between average users' interfaces and expert users' interfaces; §9.3.2.
[12] Avoid nonlocal objects that require run-time initialization in code intended for use as part of non-C++ programs; §9.4.1.

9.6 Exercises [file.exercises]

1. (∗2) Find where the standard library headers are kept on your system. List their names. Are any nonstandard headers kept together with the standard ones? Can any nonstandard headers be ***#include***d using the <> notation?
2. (∗2) Where are the headers for nonstandard library ''foundation'' libraries kept?
3. (∗2.5) Write a program that reads a source file and writes out the names of files ***#include***d. Indent file names to show files ***#included*** by included files. Try this program on some real source files (to get an idea of the amount of information included).
4. (∗3) Modify the program from the previous exercise to print the number of comment lines, the number of non-comment lines, and the number of non-comment, whitespace-separated words for each file ***#include***d.
5. (∗2.5) An external include guard is a construct that tests outside the file it is guarding and ***include***s only once per compilation. Define such a construct, devise a way of testing it, and discuss its advantages and disadvantages compared to the include guards described in §9.3.3. Is there any significant run-time advantage to external include guards on your system.
6. (∗3) How is dynamic linking achieved on your system. What restrictions are placed on dynamically linked code? What requirements are placed on code for it to be dynamically linked?

7. (*3) Open and read 100 files containing 1500 characters each. Open and read one file containing 150,000 characters. Hint: See example in §21.5.1. Is there a performance difference? What is the highest number of files that can be simultaneously open on your system? Consider these questions in relation to the use of ***#include*** files.
8. (*2) Modify the desk calculator so that it can be invoked from ***main*( )** or from other functions as a simple function call.
9. (*2) Draw the ''module dependency diagrams'' (§9.3.2) for the version of the calculator that used ***error*( )** instead of exceptions (§8.2.2).

Part II

Abstraction Mechanisms

This part describes C++'s facilities for defining and using new types. Techniques commonly called object-oriented programming and generic programming are presented.

Chapters

10 Classes
11 Operator Overloading
12 Derived Classes
13 Templates
14 Exception Handling
15 Class Hierarchies

‘‘... there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer makes enemies of all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order...’’

— Nicollo Machiavelli (‘‘The Prince’’ §vi)

10

Classes

Those types are not "abstract";
they are as real as `int` *and* `float`.
– Doug McIlroy

Concepts and classes — class members — access control — constructors — ***static*** members — default copy — ***const*** member functions — ***this*** — ***struct***s — in-class function definition — concrete classes — member functions and helper functions — overloaded operators — use of concrete classes — destructors — default construction — local variables — user-defined copy — ***new*** and ***delete*** — member objects — arrays — static storage — temporary variables — unions — advice — exercises.

10.1 Introduction [class.intro]

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types. In addition, derived classes (Chapter 12) and templates (Chapter 13) provide ways of organizing related classes that allow the programmer to take advantage of their relationships.

A type is a concrete representation of a concept. For example, the C++ built-in type ***float*** with its operations `+`, `-`, `*`, etc., provides a concrete approximation of the mathematical concept of a real number. A class is a user-defined type. We design a new type to provide a definition of a concept that has no direct counterpart among the built-in types. For example, we might provide a type ***Trunk_line*** in a program dealing with telephony, a type ***Explosion*** for a videogame, or a type ***list<Paragraph>*** for a text-processing program. A program that provides types that closely match the concepts of the application tends to be easier to understand and easier to modify than a program that does not. A well-chosen set of user-defined types makes a program more concise. In addition, it makes many sorts of code analysis feasible. In particular, it enables the compiler to detect illegal uses of objects that would otherwise remain undetected until the program is thoroughly tested.

The fundamental idea in defining a new type is to separate the incidental details of the implementation (e.g., the layout of the data used to store an object of the type) from the properties essential to the correct use of it (e.g., the complete list of functions that can access the data). Such a separation is best expressed by channeling all uses of the data structure and internal housekeeping routines through a specific interface.

This chapter focuses on relatively simple ''concrete'' user-defined types that logically don't differ much from built-in types. Ideally, such types should not differ from built-in types in the way they are used, only in the way they are created.

10.2 Classes [class.class]

A *class* is a user-defined type. This section introduces the basic facilities for defining a class, creating objects of a class, and manipulating such objects.

10.2.1 Member Functions [class.member]

Consider implementing the concept of a date using a ***struct*** to define the representation of a ***Date*** and a set of functions for manipulating variables of this type:

```
struct Date {          // representation
    int d, m, y;
};

void init_date(Date& d, int, int, int);     // initialize d
void add_year(Date& d, int n);              // add n years to d
void add_month(Date& d, int n);             // add n months to d
void add_day(Date& d, int n);               // add n days to d
```

There is no explicit connection between the data type and these functions. Such a connection can be established by declaring the functions as members:

```
struct Date {
    int d, m, y;

    void init(int dd, int mm, int yy);   // initialize
    void add_year(int n);                // add n years
    void add_month(int n);               // add n months
    void add_day(int n);                 // add n days
};
```

Functions declared within a class definition (a ***struct*** is a kind of class; §10.2.8) are called member functions and can be invoked only for a specific variable of the appropriate type using the standard syntax for structure member access. For example:

```
Date my_birthday;

void f()
{
    Date today;
```

```
        today.init(16,10,1996);
        my_birthday.init(30,12,1950);

        Date tomorrow = today;
        tomorrow.add_day(1);
        // ...
}
```

Because different structures can have member functions with the same name, we must specify the structure name when defining a member function:

```
void Date::init(int dd, int mm, int yy)
{
        d = dd;
        m = mm;
        y = yy;
}
```

In a member function, member names can be used without explicit reference to an object. In that case, the name refers to that member of the object for which the function was invoked. For example, when ***Date*::*init*()** is invoked for ***today***, ***m*=*mm*** assigns to ***today*.*m***. On the other hand, when ***Date*::*init*()** is invoked for ***my_birthday***, ***m*=*mm*** assigns to ***my_birthday*.*m***. A class member function always ''knows'' for which object it was invoked.

The construct

```
class X { ... };
```

is called a *class definition* because it defines a new type. For historical reasons, a class definition is often referred to as a *class declaration*. Also, like declarations that are not definitions, a class definition can be replicated in different source files using ***#include*** without violating the one-definition rule (§9.2.3).

10.2.2 Access Control [class.access]

The declaration of ***Date*** in the previous subsection provides a set of functions for manipulating a ***Date***. However, it does not specify that those functions should be the only ones to depend directly on ***Date***'s representation and the only ones to directly access objects of class ***Date***. This restriction can be expressed by using a ***class*** instead of a ***struct***:

```
class Date {
        int d, m, y;
public:
        void init(int dd, int mm, int yy);    // initialize

        void add_year(int n);                 // add n years
        void add_month(int n);                // add n months
        void add_day(int n);                  // add n days
};
```

The ***public*** label separates the class body into two parts. The names in the first, *private*, part can be used only by member functions. The second, *public*, part constitutes the public interface to objects

of the class. A ***struct*** is simply a ***class*** whose members are public by default (§10.2.8); member functions can be defined and used exactly as before. For example:

```
inline void Date::add_year(int n)
{
    y += n;
}
```

However, nonmember functions are barred from using private members. For example:

```
void timewarp(Date& d)
{
    d.y -= 200;     // error: Date::y is private
}
```

There are several benefits to be obtained from restricting access to a data structure to an explicitly declared list of functions. For example, any error causing a ***Date*** to take on an illegal value (for example, December 36, 1985) must be caused by code in a member function. This implies that the first stage of debugging – localization – is completed before the program is even run. This is a special case of the general observation that any change to the behavior of the type ***Date*** can and must be effected by changes to its members. In particular, if we change the representation of a class, we need only change the member functions to take advantage of the new representation. User code directly depends only on the public interface and need not be rewritten (although it may need to be recompiled). Another advantage is that a potential user need examine only the definition of the member functions in order to learn to use a class.

The protection of private data relies on restriction of the use of the class member names. It can therefore be circumvented by address manipulation and explicit type conversion. But this, of course, is cheating. C++ protects against accident rather than deliberate circumvention (fraud). Only hardware can protect against malicious use of a general-purpose language, and even that is hard to do in realistic systems.

The ***init***( ) function was added partially because it is generally useful to have a function that sets the value of an object and partly because making the data private forces us to provide it.

10.2.3 Constructors [class.ctor]

The use of functions such as ***init***( ) to provide initialization for class objects is inelegant and error-prone. Because it is nowhere stated that an object must be initialized, a programmer can forget to do so – or do so twice (often with equally disastrous results). A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type, it is called a *constructor*. A constructor is recognized by having the same name as the class itself. For example:

```
class Date {
    // ...
    Date(int, int, int);        // constructor
};
```

When a class has a constructor, all objects of that class will be initialized. If the constructor requires arguments, these arguments must be supplied:

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);          // abbreviated form
Date my_birthday;               // error: initializer missing
Date release1_0(10,12);         // error: 3rd argument missing
```

It is often nice to provide several ways of initializing a class object. This can be done by providing several constructors. For example:

```
class Date {
    int d, m, y;
public:
    // ...
    Date(int, int, int);        // day, month, year
    Date(int, int);             // day, month, today's year
    Date(int);                  // day, today's month and year
    Date();                     // default Date: today
    Date(const char*);          // date in string representation
};
```

Constructors obey the same overloading rules as do other functions (§7.4). As long as the constructors differ sufficiently in their argument types, the compiler can select the correct one for each use:

```
Date today(4);
Date july4("July 4, 1983");
Date guy("5 Nov");
Date now;                       // default initialized as today
```

The proliferation of constructors in the ***Date*** example is typical. When designing a class, a programmer is always tempted to add features just because somebody might want them. It takes more thought to carefully decide what features are really needed and to include only those. However, that extra thought typically leads to smaller and more comprehensible programs. One way of reducing the number of related functions is to use default arguments (§7.5). In the ***Date***, each argument can be given a default value interpreted as ''pick the default: ***today***.''

```
class Date {
    int d, m, y;
public:
    Date(int dd=0, int mm=0, int yy=0);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;

    // check that the Date is valid
}
```

When an argument value is used to indicate ''pick the default,'' the value chosen must be outside the set of possible values for the argument. For ***day*** and ***month***, this is clearly so, but for ***year***, zero

may not be an obvious choice. Fortunately, there is no year zero on the European calendar; 1AD (*year==1*) comes immediately after 1BC (*year==-1*).

10.2.4 Static Members [class.static]

The convenience of a default value for ***Date***s was bought at the cost of a significant hidden problem. Our ***Date*** class became dependent on the global variable ***today***. This ***Date*** class can be used only in a context in which ***today*** is defined and correctly used by every piece of code. This is the kind of constraint that causes a class to be useless outside the context in which it was first written. Users get too many unpleasant surprises trying to use such context-dependent classes, and maintenance becomes messy. Maybe ''just one little global variable'' isn't too unmanageable, but that style leads to code that is useless except to its original programmer. It should be avoided.

Fortunately, we can get the convenience without the encumbrance of a publicly accessible global variable. A variable that is part of a class, yet is not part of an object of that class, is called a ***static*** member. There is exactly one copy of a ***static*** member instead of one copy per object, as for ordinary non-***static*** members. Similarly, a function that needs access to members of a class, yet doesn't need to be invoked for a particular object, is called a ***static*** member function.

Here is a redesign that preserves the semantics of default constructor values for ***Date*** without the problems stemming from reliance on a global:

```
class Date {
    int d, m, y;
    static Date default_date;
public:
    Date(int dd=0, int mm=0, int yy=0);
    // ...
    static void set_default(int, int, int);
};
```

We can now define the ***Date*** constructor like this:

```
Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : default_date.d;
    m = mm ? mm : default_date.m;
    y = yy ? yy : default_date.y;

    // check that the Date is valid
}
```

We can change the default date when appropriate. A static member can be referred to like any other member. In addition, a static member can be referred to without mentioning an object. Instead, its name is qualified by the name of its class. For example:

```
void f()
{
    Date::set_default(4,5,1945);
}
```

Static members – both function and data members – must be defined somewhere. For example:

```
Date Date::default_date(16,12,1770);

void Date::set_default(int d, int m, int y)
{
    Date::default_date = Date(d,m,y);
}
```

Now the default value is Beethoven's birth date – until someone decides otherwise.

Note that *Date*( ) serves as a notation for the value of *Date*::*default_date*. For example:

```
Date copy_of_default_date = Date();
```

Consequently, we don't need a separate function for reading the default date.

10.2.5 Copying Class Objects [class.default.copy]

By default, class objects can be copied. In particular, a class object can be initialized with a copy of another object of the same class. This can be done even where constructors have been declared. For example:

```
Date d = today;        // initialization by copy
```

By default, the copy of a class object is a copy of each member. If that default is not the behavior wanted for a class *X*, a more appropriate behavior can be provided by defining a copy constructor, *X*::*X*(*const X*&). This is discussed further in §10.4.4.1.

Similarly, class objects can by default be copied by assignment. For example:

```
void f(Date& d)
{
    d = today;
}
```

Again, the default semantics is memberwise copy. If that is not the right choice for a class *X*, the user can define an appropriate assignment operator (§10.4.4.1).

10.2.6 Constant Member Functions [class.constmem]

The *Date* defined so far provides member functions for giving a *Date* a value and changing it. Unfortunately, we didn't provide a way of examining the value of a *Date*. This problem can easily be remedied by adding functions for reading the day, month, and year:

```
class Date {
    int d, m, y;
public:
    int day() const { return d; }
    int month() const { return m; }
    int year() const;
    // ...
};
```

Note the *const* after the (empty) argument list in the function declarations. It indicates that these functions do not modify the state of a *Date*.

Naturally, the compiler will catch accidental attempts to violate this promise. For example:

```
inline int Date::year() const
{
    return y++;    // error: attempt to change member value in const function
}
```

When a *const* member function is defined outside its class, the *const* suffix is required:

```
inline int Date::year() const        // correct
{
    return y;
}

inline int Date::year()    // error: const missing in member function type
{
    return y;
}
```

In other words, the *const* is part of the type of *Date::day()* and *Date::year()*.

A *const* member function can be invoked for both *const* and non-*const* objects, whereas a non-*const* member function can be invoked only for non-*const* objects. For example:

```
void f(Date& d, const Date& cd)
{
    int i = d.year();      // ok
    d.add_year(1);         // ok

    int j = cd.year();     // ok
    cd.add_year(1);        // error: cannot change value of const cd
}
```

10.2.7 Self-Reference [class.this]

The state update functions *add_year()*, *add_month()*, and *add_day()* were defined not to return values. For such a set of related update functions, it is often useful to return a reference to the updated object so that the operations can be chained. For example, we would like to write

```
void f(Date& d)
{
    // ...
    d.add_day(1).add_month(1).add_year(1);
    // ...
}
```

to add a day, a month, and a year to *d*. To do this, each function must be declared to return a reference to a *Date*:

```
class Date {
    // ...
```

```
        Date& add_year(int n);     // add n years
        Date& add_month(int n);    // add n months
        Date& add_day(int n);      // add n days
};
```

Each (nonstatic) member function knows what object it was invoked for and can explictly refer to it. For example:

```
Date& Date::add_year(int n)
{
        if (d==29 && m==2 && !leapyear(y+n)) { // beware of February 29
                d = 1;
                m = 3;
        }
        y += n;
        return *this;
}
```

The expression ***this*** refers to the object for which a member function is invoked. It is equivalent to Simula's ***THIS*** and Smalltalk's ***self***.

In a nonstatic member function, the keyword ***this*** is a pointer to the object for which the function was invoked. In a non-***const*** member function of class ***X***, the type of ***this*** is ***X *const***. The ***const*** makes it clear that the user is not supposed to change the value of ***this***. In a ***const*** member function of class ***X***, the type of ***this*** is ***const X *const*** to prevent modification of the object itself (see also §5.4.1).

Most uses of ***this*** are implicit. In particular, every reference to a nonstatic member from within a class relies on an implicit use of ***this*** to get the member of the appropriate object. For example, the ***add_year*** function could equivalently, but tediously, have been defined like this:

```
Date& Date::add_year(int n)
{
        if (this->d==29 && this->m==2 && !leapyear(this->y+n)) {
                this->d = 1;
                this->m = 3;
        }
        this->y += n;
        return *this;
}
```

One common explicit use of ***this*** is in linked-list manipulation (e.g., §24.3.7.4).

10.2.7.1 Physical and Logical Constness [class.const]

Occasionally, a member function is logically ***const***, but it still needs to change the value of a member. To a user, the function appears not to change the state of its object. However, some detail that the user cannot directly observe is updated. This is often called *logical constness*. For example, the ***Date*** class might have a function returning a string representation that a user could use for output. Constructing this representation could be a relatively expensive operation. Therefore, it would make sense to keep a copy so that repeated requests would simply return the copy, unless the

Date's value had been changed. Caching values like that is more common for more complicated data structures, but let's see how it can be achieved for a *Date*:

```
class Date {
    bool cache_valid;
    string cache;
    void compute_cache_value();    // fill cache
    // ...
public:
    // ...
    string string_rep() const;     // string representation
};
```

From a user's point of view, *string_rep* doesn't change the state of its *Date*, so it clearly should be a *const* member function. On the other hand, the cache needs to be filled before it can be used. This can be achieved through brute force:

```
string Date::string_rep() const
{
    if (cache_valid == false) {
        Date* th = const_cast<Date*>(this);  // cast away const
        th->compute_cache_value();
        th->cache_valid = true;
    }
    return cache;
}
```

That is, the *const_cast* operator (§15.4.2.1) is used to obtain a pointer of type *Date** to *this*. This is hardly elegant, and it is not guaranteed to work when applied to an object that was originally declared as a *const*. For example:

```
Date d1;
const Date d2;
string s1 = d1.string_rep();
string s2 = d2.string_rep();    // undefined behavior
```

In the case of *d1*, *string_rep*( ) simply casts back to *d1*'s original type so that the call will work. However, *d2* was defined as a *const* and the implementation could have applied some form of memory protection to ensure that its value wasn't corrupted. Consequently, *d2.string_rep*( ) is not guaranteed to give a single predictable result on all implementations.

10.2.7.2 Mutable [class.mutable]

The explicit type conversion ''casting away *const*'' and its consequent implementation-dependent behavior can be avoided by declaring the data involved in the cache management to be *mutable*:

```
class Date {
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value() const;   // fill (mutable) cache
    // ...
public:
    // ...
    string string_rep() const;          // string representation
};
```

The storage specifier ***mutable*** specifies that a member should be stored in a way that allows updating – even when it is a member of a ***const*** object. In other words, ***mutable*** means ''can never be ***const***.'' This can be used to simplify the definition of ***string_rep()***:

```
string Date::string_rep() const
{
    if (!cache_valid) {
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```

and makes reasonable uses of ***string_rep()*** valid. For example:

```
Date d3;
const Date d4;
string s3 = d3.string_rep();
string s4 = d4.string_rep();    // ok!
```

Declaring members ***mutable*** is most appropriate when (only) part of a representation is allowed to change. If most of an object changes while the object remains logically ***const***, it is often better to place the changing data in a separate object and access it indirectly. If that technique is used, the string-with-cache example becomes:

```
struct cache {
    bool valid;
    string rep;
};

class Date {
    cache* c;                           // initialize in constructor (§10.4.6)
    void compute_cache_value() const;   // fill what cache refers to
    // ...
public:
    // ...
    string string_rep() const;          // string representation
};
```

```
string Date::string_rep() const
{
    if (!c->valid) {
        compute_cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

The programming techniques that support a cache generalize to various forms of lazy evaluation.

10.2.8 Structures and Classes [class.struct]

By definition, a ***struct*** is a class in which members are by default public; that is,

```
struct s { ...
```

is simply shorthand for

```
class s { public: ...
```

The access specifier ***private*:** can be used to say that the members following are private, just as ***public*:** says that the members following are public. Except for the different names, the following declarations are equivalent:

```
class Date1 {
    int d, m, y;
public:
    Date1(int dd, int mm, int yy);

    void add_year(int n);      // add n years
};

struct Date2 {
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);

    void add_year(int n);      // add n years
};
```

Which style you use depends on circumstances and taste. I usually prefer to use ***struct*** for classes that have all data public. I think of such classes as ‘‘not quite proper types, just data structures.’’ Constructors and access functions can be quite useful even for such structures, but as a shorthand rather than guarantors of properties of the type (invariants, see §24.3.7.1).

It is not a requirement to declare data first in a class. In fact, it often makes sense to place data members last to emphasize the functions providing the public user interface. For example:

```
class Date3 {
public:
    Date3(int dd, int mm, int yy);
```

```
        void add_year(int n);      // add n years
    private:
        int d, m, y;
    };
```

In real code, where both the public interface and the implementation details typically are more extensive than in tutorial examples, I usually prefer the style used for *Date3*.

Access specifiers can be used many times in a single class declaration. For example:

```
    class Date4 {
    public:
        Date4(int dd, int mm, int yy);
    private:
        int d, m, y;
    public:
        void add_year(int n);      // add n years
    };
```

Having more than one public section, as in *Date4*, tends to be messy. So does having more than one private section. However, allowing many access specifiers in a class is useful for machine-generated code.

10.2.9 In-Class Function Definitions [class.inline]

A member function defined within the class definition – rather than simply declared there – is taken to be an inline member function. That is, in-class definition of member functions is for small, frequently-used functions. Like the class definition it is part of, a member function defined in-class can be replicated in several translation units using *#include*. Like the class itself, its meaning must be the same wherever it is used (§9.2.3).

The style of placing the definition of data members last in a class can lead to a minor problem with public inline functions that refer to the representation. Consider:

```
    class Date {     // potentially confusing
    public:
        int day() const { return d; }    // return Date::d
        // ...
    private:
        int d, m, y;
    };
```

This is perfectly good C++ code because a member function declared within a class can refer to every member of the class as if the class were completely defined before the member function bodies were considered. However, this can confuse human readers.

Consequently, I usually either place the data first or define the inline member functions after the class itself. For example:

```
class Date {
public:
    int day() const;
    // ...
private:
    int d, m, y;
};

inline int Date::day() const { return d; }
```

10.3 Efficient User-Defined Types [class.concrete]

The previous section discussed bits and pieces of the design of a *Date* class in the context of introducing the basic language features for defining classes. Here, I reverse the emphasis and discuss the design of a simple and efficient *Date* class and show how the language features support this design.

Small, heavily-used abstractions are common in many applications. Examples are Latin characters, Chinese characters, integers, floating-point numbers, complex numbers, points, pointers, coordinates, transforms, (*pointer,offset*) pairs, dates, times, ranges, links, associations, nodes, (*value,unit*) pairs, disk locations, source code locations, *BCD* characters, currencies, lines, rectangles, scaled fixed-point numbers, numbers with fractions, character strings, vectors, and arrays. Every application uses several of these. Often, a few of these simple concrete types are used heavily. A typical application uses a few directly and many more indirectly from libraries.

C++ and other programming languages directly support a few of these abstractions. However, most are not, and cannot be, supported directly because there are too many of them. Furthermore, the designer of a general-purpose programming language cannot foresee the detailed needs of every application. Consequently, mechanisms must be provided for the user to define small concrete types. Such types are called concrete types or concrete classes to distinguish them from abstract classes (§12.3) and classes in class hierarchies (§12.2.4, §12.4).

It was an explicit aim of C++ to support the definition and efficient use of such user-defined data types very well. They are a foundation of elegant programming. As usual, the simple and mundane is statistically far more significant than the complicated and sophisticated.

In this light, let us build a better date class:

```
class Date {
public:         // public interface:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Bad_date { };  // exception class

    Date(int dd =0, Month mm =Month(0), int yy =0);  // 0 means ''pick a default''

// functions for examining the Date:
    int day() const;
    Month month() const;
    int year() const;
```

```
        string string_rep() const;          // string representation
        void char_rep(char s[]) const;      // C-style string representation

        static void set_default(int, Month, int);

    // functions for changing the Date:
        Date& add_year(int n);              // add n years
        Date& add_month(int n);             // add n months
        Date& add_day(int n);               // add n days
    private:
        int d, m, y;                        // representation
        static Date default_date;
    };
```

This set of operations is fairly typical for a user-defined type:

[1] A constructor specifying how objects/variables of the type are to be initialized.

[2] A set of functions allowing a user to examine a ***Date***. These functions are marked ***const*** to indicate that they don't modify the state of the object/variable for which they are called.

[3] A set of functions allowing the user to manipulate ***Date***s without actually having to know the details of the representation or fiddle with the intricacies of the semantics.

[4] A set of implicitly defined operations to allow ***Date***s to be freely copied.

[5] A class, ***Bad_date***, to be used for reporting errors as exceptions.

I defined a ***Month*** type to cope with the problem of remembering, for example, whether the 7th of June is written ***Date(6,7)*** (American style) or ***Date(7,6)*** (European style). I also added a mechanism for dealing with default arguments.

I considered introducing separate types ***Day*** and ***Year*** to cope with possible confusion of ***Date(1995,jul,27)*** and ***Date(27,jul,1995)***. However, these types would not be as useful as the ***Month*** type. Almost all such errors are caught at run-time anyway – the 26th of July year 27 is not a common date in my work. How to deal with historical dates before year 1800 or so is a tricky issue best left to expert historians. Furthermore, the day of the month can't be properly checked in isolation from its month and year. See §11.7.1 for a way of defining a convenient ***Year*** type.

The default date must be defined as a valid ***Date*** somewhere. For example:

```
Date Date::default_date(22,jan,1901);
```

I omitted the cache technique from §10.2.7.1 as unnecessary for a type this simple. If needed, it can be added as an implementation detail without affecting the user interface.

Here is a small – and contrived – example of how ***Date***s can be used:

```
void f(Date& d)
{
    Date lvb_day = Date(16,Date::dec,d.year());

    if (d.day()==29 && d.month()==Date::feb) {
        // ...
    }

    if (midnight()) d.add_day(1);

    cout << "day after:" << d+1 << '\n';
}
```

This assumes that the output operator << and the addition operator + have been declared for *Date*s. I do that in §10.3.3.

Note the *Date*::*feb* notation. The function *f*( ) is not a member of *Date*, so it must specify that it is referring to *Date*'s *feb* and not to some other entity.

Why is it worthwhile to define a specific type for something as simple as a date? After all, we could define a structure:

```
struct Date {
    int day, month, year;
};
```

and let programmers decide what to do with it. If we did that, though, every user would either have to manipulate the components of *Date*s directly or provide separate functions for doing so. In effect, the notion of a date would be scattered throughout the system, which would make it hard to understand, document, or change. Inevitably, providing a concept as only a simple structure causes extra work for every user of the structure.

Also, even though the *Date* type seems simple, it takes some thought to get right. For example, incrementing a *Date* must deal with leap years, with the fact that months are of different lengths, and so on (note: §10.6[1]). Also, the day-month-and-year representation is rather poor for many applications. If we decided to change it, we would need to modify only a designated set of functions. For example, to represent a *Date* as the number of days before or after January 1, 1970, we would need to change only *Date*'s member functions (§10.6[2]).

10.3.1 Member Functions [class.memfct]

Naturally, an implementation for each member function must be provided somewhere. For example, here is the definition of *Date*'s constructor:

```
Date::Date(int dd, Month mm, int yy)
{
    if (yy == 0) yy = default_date.year();
    if (mm == 0) mm = default_date.month();
    if (dd == 0) dd = default_date.day();

    int max;

    switch (mm) {
    case feb:
        max = 28+leapyear(yy);
        break;
    case apr: case jun: case sep: case nov:
        max = 30;
        break;
    case jan: case mar: case may: case jul: case aug: case oct: case dec:
        max = 31;
        break;
    default:
        throw Bad_date();   // someone cheated
    }
```

```
        if (dd<1 || max<dd) throw Bad_date();

        y = yy;
        m = mm;
        d = dd;
    }
```

The constructor checks that the data supplied denotes a valid *Date*. If not, say for *Date*(*30*,*Date*::*feb*,*1994*), it throws an exception (§8.3, Chapter 14), which indicates that something went wrong in a way that cannot be ignored. If the data supplied is acceptable, the obvious initialization is done. Initialization is a relatively complicated operation because it involves data validation. This is fairly typical. On the other hand, once a *Date* has been created, it can be used and copied without further checking. In other words, the constructor establishes the invariant for the class (in this case, that it denotes a valid date). Other member functions can rely on that invariant and must maintain it. This design technique can simplify code immensely (see §24.3.7.1).

I'm using the value *Month*(*0*) – which doesn't represent a month – to represent ''pick the default month.'' I could have defined an enumerator in *Month* specifically to represent that. But I decided that it was better to use an obviously anomalous value to represent ''pick the default month'' rather than give the appearance that there were 13 months in a year. Note that *0* can be used because it is within the range guaranteed for the enumeration *Month* (§4.8).

I considered factoring out the data validation in a separate function *is_date*(). However, I found the resulting user code more complicated and less robust than code relying on catching the exception. For example, assuming that >> is defined for *Date*:

```
void fill(vector<Date>& aa)
{
    while (cin) {
        Date d;
        try {
            cin >> d;
        }

        catch (Date::Bad_date) {
            // my error handling
            continue;
        }
        aa.push_back(d);    // see §3.7.3
    }
}
```

As is common for such simple concrete types, the definitions of member functions vary between the trivial and the not-too-complicated. For example:

```
inline int Date::day() const
{
    return d;
}
```

```
Date& Date::add_month(int n)
{
    if (n==0) return *this;

    if (n>0) {
        int delta_y = n/12;
        int mm = m+n%12;
        if (12 < mm) {  // note: int(dec)==12
            delta_y++;
            mm -= 12;
        }

        // handle the cases where Month(mm) doesn't have day d

        y += delta_y;
        m = Month(mm);
        return *this;
    }

    // handle negative n

    return *this;
}
```

10.3.2 Helper Functions [class.helper]

Typically, a class has a number of functions associated with it that need not be defined in the class itself because they don't need direct access to the representation. For example:

```
int diff(Date a, Date b); // number of days in the range [a,b) or [b,a)
bool leapyear(int y);
Date next_weekday(Date d);
Date next_saturday(Date d);
```

Defining such functions in the class itself would complicate the class interface and increase the number of functions that would potentially need to be examined when a change to the representation was considered.

How are such functions ''associated'' with class ***Date***? Traditionally, their declarations were simply placed in the same file as the declaration of class ***Date***, and users who needed ***Date***s would make them all available by including the file that defined the interface (§9.2.1). For example:

```
#include "Date.h"
```

In addition to using a specific ***Date.h*** header, or as an alternative, we can make the association explicit by enclosing the class and its helper functions in a namespace (§8.2):

```
namespace Chrono {        // facilities for dealing with time

    class Date { /* ... */};
```

```
        int diff(Date a, Date b);
        bool leapyear(int y);
        Date next_weekday(Date d);
        Date next_saturday(Date d);
        // ...
}
```

The ***Chrono*** namespace would naturally also contain related classes, such as ***Time*** and ***Stopwatch***, and their helper functions. Using a namespace to hold a single class is usually an over-elaboration that leads to inconvenience.

10.3.3 Overloaded Operators [class.over]

It is often useful to add functions to enable conventional notation. For example, the ***operator***== function defines the equality operator == to work for ***Date***s:

```
inline bool operator==(Date a, Date b)  // equality
{
        return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

Other obvious candidates are:

```
bool operator!=(Date, Date);              // inequality
bool operator<(Date, Date);               // less than
bool operator>(Date, Date);               // greater than
// ...

Date& operator++(Date& d);                // increase Date by one day
Date& operator--(Date& d);                // decrease Date by one day

Date& operator+=(Date& d, int n);         // add n days
Date& operator-=(Date& d, int n);         // subtract n days

Date operator+(Date d, int n);            // add n days
Date operator-(Date d, int n);            // subtract n days

ostream& operator<<(ostream&, Date d);    // output d
istream& operator>>(istream&, Date& d);   // read into d
```

For ***Date***, these operators can be seen as mere conveniences. However, for many types – such as complex numbers (§11.3), vectors (§3.7.1), and function-like objects (§18.4) – the use of conventional operators is so firmly entrenched in people's minds that their definition is almost mandatory. Operator overloading is discussed in Chapter 11.

10.3.4 The Significance of Concrete Classes [class.significance]

I call simple user-defined types, such as ***Date***, *concrete types* to distinguish them from abstract classes (§2.5.4) and class hierarchies (§12.3) and also to emphasize their similarity to built-in types such as ***int*** and ***char***. They have also been called *value types*, and their use *value-oriented programming*. Their model of use and the ''philosophy'' behind their design are quite different from what is often advertised as object-oriented programming (§2.6.2).

The intent of a concrete type is to do a single, relatively small thing well and efficiently. It is not usually the aim to provide the user with facilities to modify the behavior of a concrete type. In particular, concrete types are not intended to display polymorphic behavior (see §2.5.5, §12.2.6).

If you don't like some detail of a concrete type, you build a new one with the desired behavior. If you want to ''reuse'' a concrete type, you use it in the implementation of your new type exactly as you would have used an ***int***. For example:

```
class Date_and_time {
private:
        Date d;
        Time t;
public:
        Date_and_time(Date d, Time t);
        Date_and_time(int d, Date::Month m, int y, Time t);
        // ...
};
```

The derived class mechanism discussed in Chapter 12 can be used to define new types from a concrete class by describing the desired differences. The definition of ***Vec*** from ***vector*** (§3.7.2) is an example of this.

With a reasonably good compiler, a concrete class such as ***Date*** incurs no hidden overhead in time or space. The size of a concrete type is known at compile time so that objects can be allocated on the run-time stack (that is, without free-store operations). The layout of each object is known at compile time so that inlining of operations is trivially achieved. Similarly, layout compatibility with other languages, such as C and Fortran, comes without special effort.

A good set of such types can provide a foundation for applications. Lack of suitable ''small efficient types'' in an application can lead to gross run-time and space inefficiencies when overly general and expensive classes are used. Alternatively, lack of concrete types can lead to obscure programs and time wasted when each programmer writes code to directly manipulate ''simple and frequently used'' data structures.

10.4 Objects [class.objects]

Objects can be created in several ways. Some are local variables, some are global variables, some are members of classes, etc. This section discusses these alternatives, the rules that govern them, the constructors used to initialize objects, and the destructors used to clean up objects before they become unusable.

10.4.1 Destructors [class.dtor]

A constructor initializes an object. In other words, it creates the environment in which the member functions operate. Sometimes, creating that environment involves acquiring a resource – such as a file, a lock, or some memory – that must be released after use (§14.4.7). Thus, some classes need a function that is guaranteed to be invoked when an object is destroyed in a manner similar to the way a constructor is guaranteed to be invoked when an object is created. Inevitably, such functions are called *destructors*. They typically clean up and release resources. Destructors are called

implicitly when an automatic variable goes out of scope, an object on the free store is deleted, etc. Only in very unusual circumstances does the user need to call a destructor explicitly (§10.4.11).

The most common use of a destructor is to release memory acquired in a constructor. Consider a simple table of elements of some type ***Name***. The constructor for ***Table*** must allocate memory to hold the elements. When the table is somehow deleted, we must ensure that this memory is reclaimed for further use elsewhere. We do this by providing a special function to complement the constructor:

```
class Name {
    const char* s;
    // ...
};

class Table {
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz = s]; }// constructor

    ~Table() { delete[] p; }                      // destructor

    Name* lookup(const char *);
    bool insert(Name*);
};
```

The destructor notation ***~Table*( )** uses the complement symbol ~ to hint at the destructor's relation to the ***Table*( )** constructor.

A matching constructor/destructor pair is the usual mechanism for implementing the notion of a variably-sized object in C++. Standard library containers, such as ***map***, use a variant of this technique for providing storage for their elements, so the following discussion illustrates techniques you rely on every time you use a standard container (including a standard ***string***). The discussion applies to types without a destructor, also. Such types are seen simply as having a destructor that does nothing.

10.4.2 Default Constructors [class.default]

Similarly, most types can be considered to have a default constructor. A default constructor is a constructor that can be called without supplying an argument. Because of the default argument ***15***, ***Table*::*Table*(*size_t*)** is a default constructor. If a user has declared a default constructor, that one will be used; otherwise, the compiler will try to generate one if needed and if the user hasn't declared other constructors. A compiler-generated default constructor implicitly calls the default constructors for a class' members of class type and bases (§12.2.2). For example:

```
struct Tables {
    int i;
    int vi[10];
    Table t1;
    Table vt[10];
};
```

```
Tables tt;
```

Here, *tt* will be initialized using a generated default constructor that calls *Table*(*15*) for *tt.t1* and each element of *tt.vt*. On the other hand, *tt.i* and the elements of *tt.vi* are not initialized because those objects are not of a class type. The reasons for the dissimilar treatment of classes and built-in types are C compatibility and fear of run-time overhead.

Because *const*s and references must be initialized (§5.5, §5.4), a class containing *const* or reference members cannot be default-constructed unless the programmer explicitly supplies a constructor (§10.4.6.1). For example:

```
struct X {
    const int a;
    const int& r;
};
X x; // error: no default constructor for X
```

Default constructors can be invoked explicitly (§10.4.10). Built-in types also have default constructors (§6.2.8).

10.4.3 Construction and Destruction [class.ctor.dtor]

Consider the different ways an object can be created and how it gets destroyed afterwards. An object can be created as:

- §10.4.4 A named automatic object, which is created each time its declaration is encountered in the execution of the program and destroyed each time the program exits the block in which it occurs
- §10.4.5 A free-store object, which is created using the *new* operator and destroyed using the *delete* operator
- §10.4.6 A nonstatic member object, which is created as a member of another class object and created and destroyed when the object of which it is a member is created and destroyed
- §10.4.7 An array element, which is created and destroyed when the array of which it is an element is created and destroyed
- §10.4.8 A local static object, which is created the first time its declaration is encountered in the execution of the program and destroyed once at the termination of the program
- §10.4.9 A global, namespace, or class static object, which is created once ''at the start of the program'' and destroyed once at the termination of the program
- §10.4.10 A temporary object, which is created as part of the evaluation of an expression and destroyed at the end of the full expression in which it occurs
- §10.4.11 An object placed in memory obtained from a user-supplied function guided by arguments supplied in the allocation operation
- §10.4.12 A *union* member, which may not have a constructor or a destructor

This list is roughly sorted in order of importance. The following subsections explain these various ways of creating objects and their uses.

10.4.4 Local Variables [class.local]

The constructor for a local variable is executed each time the thread of control passes through the declaration of the local variable. The destructor for a local variable is executed each time the local variable's block is exited. Destructors for local variables are executed in reverse order of their construction. For example:

```
void f(int i)
{
    Table aa;
    Table bb;
    if (i>0) {
        Table cc;
        // ...
    }
    Table dd;
    // ...
}
```

Here, ***aa***, ***bb***, and ***dd*** are constructed (in that order) each time ***f*( )** is called, and ***dd***, ***bb***, and ***aa*** are destroyed (in that order) each time we return from ***f*( )**. If ***i>0*** for a call, ***cc*** will be constructed after ***bb*** and destroyed before ***dd*** is constructed.

10.4.4.1 Copying Objects [class.copy]

If ***t1*** and ***t2*** are objects of a class ***Table***, ***t2=t1*** by default means a memberwise copy of ***t1*** into ***t2*** (§10.2.5). Having assignment interpreted this way can cause a surprising (and usually undesired) effect when used on objects of a class with pointer members. Memberwise copy is usually the wrong semantics for copying objects containing resources managed by a constructor/destructor pair. For example:

```
void h()
{
    Table t1;
    Table t2 = t1;   // copy initialization: trouble
    Table t3;

    t3 = t2;         // copy assignment: trouble
}
```

Here, the ***Table*** default constructor is called twice: once each for ***t1*** and ***t3***. It is not called for ***t2*** because that variable was initialized by copying. However, the ***Table*** destructor is called three times: once each for ***t1***, ***t2***, and ***t3***! The default interpretation of assignment is memberwise copy, so ***t1***, ***t2***, and ***t3*** will, at the end of ***h*( )**, each contain a pointer to the array of names allocated on the free store when ***t1*** was created. No pointer to the array of names allocated when ***t3*** was created remains because it was overwritten by the ***t3=t2*** assignment. Thus, in the absence of automatic garbage collection (§10.4.5), its storage will be lost to the program forever. On the other hand, the array created for ***t1*** appears in ***t1***, ***t2***, and ***t3***, so it will be deleted thrice. The result of that is undefined and probably disastrous.

Such anomalies can be avoided by defining what it means to copy a ***Table***:

```
class Table {
    // ...
    Table(const Table&);                    // copy constructor
    Table& operator=(const Table&);         // copy assignment
};
```

The programmer can define any suitable meaning for these copy operations, but the traditional one for this kind of container is to copy the contained elements (or at least to give the user of the container the appearance that a copy has been done; see §11.12). For example:

```
Table::Table(const Table& t)            // copy constructor
{
    p = new Name[sz=t.sz];
    for (int i = 0; i<sz; i++) p[i] = t.p[i];
}

Table& Table::operator=(const Table& t)         // assignment
{
    if (this != &t) {       // beware of self-assignment: t = t
        delete[] p;
        p = new Name[sz=t.sz];
        for (int i = 0; i<sz; i++) p[i] = t.p[i];
    }
    return *this;
}
```

As is almost always the case, the copy constructor and the copy assignment differ considerably. The fundamental reason is that a copy constructor initializes uninitialized memory, whereas the copy assignment operator must correctly deal with a well-constructed object.

Assignment can be optimized in some cases, but the general strategy for an assignment operator is simple: protect against self-assignment, delete old elements, initialize, and copy in new elements. Usually every nonstatic member must be copied (§10.4.6.3).

10.4.5 Free Store [class.free]

An object created on the free store has its constructor invoked by the ***new*** operator and exists until the ***delete*** operator is applied to a pointer to it. Consider:

```
int main()
{
    Table* p = new Table;
    Table* q = new Table;

    delete p;
    delete p; // probably causes run-time error
}
```

The constructor ***Table::Table()*** is called twice. So is the destructor ***Table::~Table()***. Unfortunately, the ***new***s and the ***delete***s in this example don't match, so the object pointed to by ***p*** is

deleted twice and the object pointed to by *q* not at all. Not deleting an object is typically not an error as far as the language is concerned; it is only a waste of space. However, in a program that is meant to run for a long time, such a memory leak is a serious and hard-to-find error. There are tools available for detecting such leaks. Deleting *p* twice is a serious error; the behavior is undefined and most likely disastrous.

Some C++ implementations automatically recycle the storage occupied by unreachable objects (garbage collecting implementations), but their behavior is not standardized. Even when a garbage collector is running, ***delete*** will invoke a destructor if one is defined, so it is still a serious error to delete an object twice. In many cases, that is only a minor inconvenience. In particular, where a garbage collector is known to exist, destructors that do memory management only can be eliminated. This simplification comes at the cost of portability and for some programs, a possible increase in run time and a loss of predictability of run-time behavior (§C.9.1).

After ***delete*** has been applied to an object, it is an error to access that object in any way. Unfortunately, implementations cannot reliably detect such errors.

The user can specify how ***new*** does allocation and how ***delete*** does deallocation (see §6.2.6.2 and §15.6). It is also possible to specify the way an allocation, initialization (construction), and exceptions interact (see §14.4.5 and §19.4.5). Arrays on the free store are discussed in §10.4.7.

10.4.6 Class Objects as Members [class.m]

Consider a class that might be used to hold information for a small organization:

```
class Club {
    string name;
    Table members;
    Table officers;
    Date founded;
    // ...
    Club(const string& n, Date fd);
};
```

The ***Club***'s constructor takes the name of the club and its founding date as arguments. Arguments for a member's constructor are specified in a member initializer list in the definition of the constructor of the containing class. For example:

```
Club::Club(const string& n, Date fd)
    : name(n), members(), officers(), founded(fd)
{
    // ...
}
```

The member initializers are preceded by a colon and the individual member initializers are separated by commas.

The members' constructors are called before the body of the containing class' own constructor is executed. The constructors are called in the order in which they are declared in the class rather than the order in which they appear in the initializer list. To avoid confusion, it is best to specify the initializers in declaration order. The member destructors are called in the reverse order of construction.

If a member constructor needs no arguments, the member need not be mentioned in the member initializer list, so

```
Club::Club(const string& n, Date fd)
    : name(n), founded(fd)
{
    // ...
}
```

is equivalent to the previous version. In each case, *Club::officers* is constructed by *Table::Table* with the default argument *15*.

When a class object containing class objects is destroyed, the body of that object's own destructor (if one is specified) is executed first and then the members' destructors are executed in reverse order of declaration. A constructor assembles the execution environment for the member functions for a class from the bottom up (members first). The destructor disassembles it from the top down (members last).

10.4.6.1 Necessary Member Initialization [class.ref.init]

Member initializers are essential for types for which initialization differs from assignment – that is, for member objects of classes without default constructors, for *const* members, and for reference members. For example:

```
class X {
    const int i;
    Club c;
    Club& pc;
    // ...
    X(int ii, const string& n, Date d, Club& c) : i(ii), c(n,d), pc(c) { }
};
```

There isn't any other way to initialize such members, and it is an error not to initialize objects of those types. For most types, however, the programmer has a choice between using an initializer and using an assignment. In that case, I usually prefer to use the member initializer syntax, thus making explicit the fact that initialization is being done. Often, there also is an efficiency advantage to using the initializer syntax. For example:

```
class Person {
    string name;
    string address;
    // ...
    Person(const Person&);
    Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
    : name(n)
{
    address = a;
}
```

Here ***name*** is initialized with a copy of ***n***. On the other hand, ***address*** is first initialized to the empty string and then a copy of ***a*** is assigned.

10.4.6.2 Member Constants [class.memconst]

It is also possible to initialize a static integral constant member by adding a *constant-expression* initializer to its member declaration. For example:

```
class Curious {
public:
    static const int c1 = 7;          // ok, but remember definition
    static int c2 = 11;               // error: not const
    const int c3 = 13;                // error: not static
    static const int c4 = f(17);      // error: in-class initializer not constant
    static const float c5 = 7.0;      // error: in-class not integral
    // ...
};
```

If (and only if) you use an initialized member in a way that requires it to be stored as an object in memory, the member must be (uniquely) defined somewhere. The initializer may not be repeated:

```
const int Curious::c1;            // necessary, but don't repeat initializer here

const int* p = &Curious::c1;      // ok: Curious::c1 has been defined
```

Alternatively, you can use an enumerator (§4.8, §14.4.6, §15.3) as a symbolic constant within a class declaration. For example:

```
class X {
    enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17 };
    // ...
};
```

In that way, you are not tempted to initialize variables, floating-point numbers, etc. within a class.

10.4.6.3 Copying Members [class.mem.copy]

A default copy constructor or default copy assignment (§10.4.4.1) copies all elements of a class. If this copy cannot be done, it is an error to try to copy an object of such a class. For example:

```
class Unique_handle {
private:        // copy operations are private to prevent copying (§11.2.2)
    Unique_handle(const Unique_handle&);
    Unique_handle& operator=(const Unique_handle&);
public:
    // ...
};

struct Y {
    // ...
    Unique_handle a;    // requires explicit initialization
};
```

```
Y y1;
Y y2 = y1;        // error: cannot copy Y::a
```

In addition, a default assignment cannot be generated if a nonstatic member is a reference, a ***const***, or a user-defined type without a copy assignment.

Note that the default copy constructor leaves a reference member referring to the same object in both the original and the copied object. This can be a problem if the object referred to is supposed to be deleted.

When writing a copy constructor, we must take care to copy every element that needs to be copied. By default, elements are default-initialized, but that is often not what is desired in a copy constructor. For example:

```
Person::Person(const Person& a) : name(a.name) { }    // beware!
```

Here, I forgot to copy the ***address***, so ***address*** is initialized to the empty string by default. When adding a new member to a class, always check if there are user-defined constructors that need to be updated in order to initialize and copy the new member.

10.4.7 Arrays [class.array]

If an object of a class can be constructed without supplying an explicit initializer, then arrays of that class can be defined. For example:

```
Table tbl[10];
```

This will create an array of ***10 Table***s and initialize each ***Table*** by a call of ***Table::Table()*** with the default argument ***15***.

There is no way to specify explicit arguments for a constructor in an array declaration. If you absolutely must initialize members of an array with different values, you can write a default constructor that directly or indirectly reads and writes nonlocal data. For example:

```
class Ibuffer {
    string buf;
public:
    Ibuffer() { cin>>buf; }
    // ...
};

void f()
{
    Ibuffer words[100]; // each word initialized from cin
    // ...
}
```

It is usually best to avoid such subtleties.

The destructor for each constructed element of an array is invoked when that array is destroyed. This is done implicitly for arrays that are not allocated using ***new***. Like C, C++ doesn't distinguish between a pointer to an individual object and a pointer to the initial element of an array (§5.3). Consequently, the programmer must state whether an array or an individual object is being deleted. For example:

```
void f(int sz)
{
    Table* t1 = new Table;
    Table* t2 = new Table[sz];
    Table* t3 = new Table;
    Table* t4 = new Table[sz];

    delete t1;      // right
    delete[] t2;    // right
    delete[] t3;    // wrong: trouble
    delete t4;      // wrong: trouble
}
```

Exactly how arrays and individual objects are allocated is implementation-dependent. Therefore, different implementations will react differently to incorrect uses of the ***delete*** and ***delete*[]** operators. In simple and uninteresting cases like the previous one, a compiler can detect the problem, but generally something nasty will happen at run time.

The special destruction operator for arrays, ***delete*[]**, isn't logically necessary. However, suppose the implementation of the free store had been required to hold sufficient information for every object to tell if it was an individual or an array. The user could have been relieved of a burden, but that obligation would have imposed significant time and space overheads on some C++ implementations.

As always, if you find C-style arrays too cumbersome, use a class such as ***vector*** (§3.7.1, §16.3) instead. For example:

```
void g()
{
    vector<Table>* p1 = new vector<Table>(10);
    Table* p2 = new Table;

    delete p1;
    delete p2;
}
```

10.4.8 Local Static Store [class.obj.static]

The constructor for a local static object (§7.1.2) is called the first time the thread of control passes through the object's definition. Consider this:

```
void f(int i)
{
    static Table tbl;
    // ...
    if (i) {
        static Table tbl2;
        // ...
    }
}
```

```
int main()
{
    f(0);
    f(1);
    f(2);
    // ...
}
```

Here, the constructor is called for ***tbl*** once the first time ***f*( )** is called. Because ***tbl*** is declared ***static***, it does not get destroyed on return from ***f*( )** and it does not get constructed a second time when ***f*( )** is called again. Because the block containing the declaration of ***tbl2*** doesn't get executed for the call ***f*(*0*)**, ***tbl2*** doesn't get constructed until the call ***f*(*1*)**. It does not get constructed again when its block is entered a second time.

The destructors for local static objects are invoked in the reverse order of their construction when the program terminates (§9.4.1.1). Exactly when is unspecified.

10.4.9 Nonlocal Store [class.global]

A variable defined outside any function (that is, global, namespace, and class ***static*** variables) is initialized (constructed) before ***main*( )** is invoked, and any such variable that has been constructed will have its destructor invoked after exit from ***main*( )**. Dynamic linking complicates this picture slightly by delaying the initialization until the code is linked into the running program.

Constructors for nonlocal objects in a translation unit are executed in the order their definitions occur. Consider:

```
class X {
    // ...
    static Table memtbl;
};

Table tbl;

Table X::memtbl;

namespace Z {
    Table tbl2;
}
```

The order of construction is ***tbl***, then ***X*::*memtbl***, and then ***Z*::*tbl2***. Note that a declaration (as opposed to a definition), such as the declaration of ***memtbl*** in ***X***, doesn't affect the order of construction. The destructors are called in the reverse order of construction: ***Z*::*tbl2***, then ***X*::*memtbl***, and then ***tbl***.

No implementation-independent guarantees are made about the order of construction of nonlocal objects in different compilation units. For example:

```
// file1.c:
    Table tbl1;

// file2.c:
    Table tbl2;
```

Whether *tbl1* is constructed before *tbl2* or vice versa is implementation-dependent. The order isn't even guaranteed to be fixed in every particular implementation. Dynamic linking, or even a small change in the compilation process, can alter the sequence. The order of destruction is similarly implementation-dependent.

Sometimes when you design a library, it is necessary, or simply convenient, to invent a type with a constructor and a destructor with the sole purpose of initialization and cleanup. Such a type would be used once only: to allocate a static object so that the constructor and the destructor are called. For example:

```
class Zlib_init {
        Zlib_init();        // get Zlib ready for use
        ~Zlib_init();       // clean up after Zlib
};

class Zlib {
        static Zlib_init x;
        // ...
};
```

Unfortunately, it is not guaranteed that such an object is initialized before its first use and destroyed after its last use in a program consisting of separately compiled units. A particular C++ implementation may provide such a guarantee, but most don't. A programmer may ensure proper initialization by implementing the strategy that the implementations usually employ for local static objects: a first-time switch. For example:

```
class Zlib {
        static bool initialized;
        static void initialize() { /* initialize */ initialized = true; }
public:
        // no constructor

        void f()
        {
                if (initialized == false) initialize();
                // ...
        }
        // ...
};
```

If there are many functions that need to test the first-time switch, this can be tedious, but it is often manageable. This technique relies on the fact that statically allocated objects without constructors are initialized to *0*. The really difficult case is the one in which the first operation may be time-critical so that the overhead of testing and possible initialization can be serious. In that case, further trickery is required (§21.5.2).

An alternative approach for a simple object is to present it as a function (§9.4.1):

```
int& obj() { static int x = 0; return x; }   // initialized upon first use
```

First-time switches do not handle every conceivable situation. For example, it is possible to create objects that refer to each other during construction. Such examples are best avoided. If such

objects are necessary, they must be constructed carefully in stages. Also, there is no similarly simple last-time switch construct. Instead, see §9.4.1.1 and §21.5.2.

10.4.10 Temporary Objects [class.temp]

Temporary objects most often are the result of arithmetic expressions. For example, at some point in the evaluation of *x*y+z* the partial result *x*y* must exist somewhere. Except when performance is the issue (§11.6), temporary objects rarely become the concern of the programmer. However, it happens (§11.6, §22.4.7).

Unless bound to a reference or used to initialize a named object, a temporary object is destroyed at the end of the full expression in which it was created. A *full expression* is an expression that is not a subexpression of some other expression.

The standard ***string*** class has a member function ***c_str()*** that returns a C-style, zero-terminated array of characters (§3.5.1, §20.4.1). Also, the operator + is defined to mean string concatenation. These are very useful facilities for ***strings***. However, in combination they can cause obscure problems. For example:

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str();
    cout << cs;
    if (strlen(cs=(s2+s3).c_str())<8 && cs[0]=='a') {
        // cs used here
    }
}
```

Probably, your first reaction is ‘‘but don’t do that,’’ and I agree. However, such code does get written, so it is worth knowing how it is interpreted.

A temporary object of class ***string*** is created to hold ***s1+s2***. Next, a pointer to a C-style string is extracted from that object. Then – at the end of the expression – the temporary object is deleted. Now, where was the C-style string allocated? Probably as part of the temporary object holding ***s1+s2***, and that storage is not guaranteed to exist after that temporary is destroyed. Consequently, ***cs*** points to deallocated storage. The output operation ***cout<<cs*** might work as expected, but that would be sheer luck. A compiler can detect and warn against many variants of this problem.

The example with the *if-statement* is a bit more subtle. The condition will work as expected because the full expression in which the temporary holding ***s2+s3*** is created is the condition itself. However, that temporary is destroyed before the controlled statement is entered, so any use of ***cs*** there is not guaranteed to work.

Please note that in this case, as in many others, the problems with temporaries arose from using a high-level data type in a low-level way. A cleaner programming style would have not only yielded a more understandable program fragment, but also avoided the problems with temporaries completely. For example:

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;

    if (s.length()<8 && s[0]=='a') {
        // use s here
    }
}
```

A temporary can be used as an initializer for a ***const*** reference or a named object. For example:

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
    const string& s = s1+s2;
    string ss = s1+s2;

    g(s,ss);   // we can use s and ss here
}
```

This is fine. The temporary is destroyed when ''its'' reference or named object go out of scope. Remember that returning a reference to a local variable is an error (§7.3) and that a temporary object cannot be bound to a non-***const*** reference (§5.5).

A temporary object can also be created by explicitly invoking a constructor. For example:

```
void f(Shape& s, int x, int y)
{
    s.move(Point(x,y));      // construct Point to pass to Shape::move()
    // ...
}
```

Such temporaries are destroyed in exactly the same way as the implicitly generated temporaries.

10.4.11 Placement of Objects [class.placement]

Operator ***new*** creates its object on the free store by default. What if we wanted the object allocated elsewhere? Consider a simple class:

```
class X {
public:
    X(int);
    // ...
};
```

We can place objects anywhere by providing an allocator function with extra arguments and then supplying such extra arguments when using ***new***:

```
void* operator new(size_t, void* p) { return p; }    // explicit placement operator

void* buf = reinterpret_cast<void*>(0xF00F);    // significant address
X* p2 = new(buf)X;  // construct an X at 'buf;' invokes: operator new(sizeof(X),buf)
```

Because of this usage, the ***new*(*buf*)*X*** syntax for supplying extra arguments to ***operator new*( )** is known as the *placement syntax*. Note that every ***operator new*( )** takes a size as its first argument and that the size of the object allocated is implicitly supplied (§15.6). The ***operator new*( )** used by the ***new*** operator is chosen by the usual argument matching rules (§7.4); every ***operator new*( )** has a ***size_t*** as its first argument.

The ''placement'' ***operator new*( )** is the simplest such allocator. It is defined in the standard header ***<new>***.

The ***reinterpret_cast*** is the crudest and potentially nastiest of the type conversion operators (§6.2.7). In most cases, it simply yields a value with the same bit pattern as its argument with the type required. Thus, it can be used for the inherently implementation-dependent, dangerous, and occasionally absolutely necessary activity of converting integer values to pointers and vice versa.

The placement ***new*** construct can also be used to allocate memory from a specific arena:

```
class Arena {
public:
        virtual void* alloc(size_t) =0;
        virtual void free(void*) =0;
        // ...
};

void* operator new(size_t sz, Arena* a)
{
        return a->alloc(sz);
}
```

Now objects of arbitrary types can be allocated from different ***Arena***s as needed. For example:

```
extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
        X* p = new(Persistent) X(i);     // X in persistent storage
        X* q = new(Shared) X(i);         // X in shared memory
        // ...
}
```

Placing an object in an area that is not (directly) controlled by the standard free-store manager implies that some care is required when destroying the object. The basic mechanism for that is an explicit call of a destructor:

```
void destroy(X* p, Arena* a)
{
        p->~X();        // call destructor
        a->free(p);     // free memory
}
```

Note that explicit calls of destructors, like the use of special-purpose *global* allocators, should be avoided wherever possible. Occasionally, they are essential. For example, it would be hard to implement an efficient general container along the lines of the standard library ***vector*** (§3.7.1, §16.3.8) without using explicit destructor class. However, a novice should think thrice before

calling a destructor explicitly and also should ask a more experienced colleague before doing so.

See §14.4.7 for an explanation of how placement new interacts with exception handling.

There is no special syntax for placement of arrays. Nor need there be, since arbitrary types can be allocated by placement new. However, a special ***operator delete*( )** can be defined for arrays (§19.4.5).

10.4.12 Unions [class.union]

A named union is defined as a ***struct***, where every member has the same address (see §C.8.2). A union can have member functions but not static members.

In general, a compiler cannot know what member of a union is used; that is, the type of the object stored in a union is unknown. Consequently, a union may not have members with constructors or destructors. It wouldn't be possible to protect that object against corruption or to guarantee that the right destructor is called when the union goes out of scope.

Unions are best used in low-level code, or as part of the implementation of classes that keep track of what is stored in the union (see §10.6[20]).

10.5 Advice [class.advice]

[1] Represent concepts as classes; §10.1.
[2] Use public data (***struct***s) only when it really is just data and no invariant is meaningful for the data members; §10.2.8.
[3] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §10.3.
[4] Make a function a member only if it needs direct access to the representation of a class; §10.3.2.
[5] Use a namespace to make the association between a class and its helper functions explicit; §10.3.2.
[6] Make a member function that doesn't modify the value of its object a ***const*** member function; §10.2.6.
[7] Make a function that needs access to the representation of a class but needn't be called for a specific object a ***static*** member function; §10.2.4.
[8] Use a constructor to establish an invariant for a class; §10.3.1.
[9] If a constructor acquires a resource, its class needs a destructor to release the resource; §10.4.1.
[10] If a class has a pointer member, it needs copy operations (copy constructor and copy assignment); §10.4.4.1.
[11] If a class has a reference member, it probably needs copy operations (copy constructor and copy assignment); §10.4.6.3.
[12] If a class needs a copy operation or a destructor, it probably needs a constructor, a destructor, a copy assignment, and a copy constructor; §10.4.4.1.
[13] Check for self-assignment in copy assignments; §10.4.4.1.
[14] When writing a copy constructor, be careful to copy every element that needs to be copied (beware of default initializers); §10.4.4.1.

[15] When adding a new member to a class, always check to see if there are user-defined constructors that need to be updated to initialize the member; §10.4.6.3.
[16] Use enumerators when you need to define integer constants in class declarations; §10.4.6.1.
[17] Avoid order dependencies when constructing global and namespace objects; §10.4.9.
[18] Use first-time switches to minimize order dependencies; §10.4.9.
[19] Remember that temporary objects are destroyed at the end of the full expression in which they are created; §10.4.10.

10.6 Exercises [class.exercises]

1. (*1) Find the error in ***Date*::*add_year*()** in §10.2.2. Then find two additional errors in the version in §10.2.7.
2. (*2.5) Complete and test ***Date***. Reimplement it with ''number of days after 1/1/1970'' representation.
3. (*2) Find a ***Date*** class that is in commercial use. Critique the facilities it offers. If possible, then discuss that ***Date*** with a real user.
4. (*1) How do you access ***set_default*** from class ***Date*** from namespace ***Chrono*** (§10.3.2)? Give at least three different ways.
5. (*2) Define a class ***Histogram*** that keeps count of numbers in some intervals specified as arguments to ***Histogram***'s constructor. Provide functions to print out the histogram. Handle out-of-range values.
6. (*2) Define some classes for providing random numbers of certain distributions (for example, uniform and exponential). Each class has a constructor specifying parameters for the distribution and a function ***draw*** that returns the next value.
7. (*2.5) Complete class ***Table*** to hold (name,value) pairs. Then modify the desk calculator program from §6.1 to use class ***Table*** instead of ***map***. Compare and contrast the two versions.
8. (*2) Rewrite ***Tnode*** from §7.10[7] as a class with constructors, destructors, etc. Define a tree of ***Tnode***s as a class with constructors, destructors, etc.
9. (*3) Define, implement, and test a set of integers, class ***Intset***. Provide union, intersection, and symmetric difference operations.
10. (*1.5) Modify class ***Intset*** into a set of nodes, where ***Node*** is a structure you define.
11. (*3) Define a class for analyzing, storing, evaluating, and printing simple arithmetic expressions consisting of integer constants and the operators `+`, `-`, `*`, and `/`. The public interface should look like this:

```
class Expr {
    // ...
public:
    Expr(char*);
    int eval();
    void print();
};
```

The string argument for the constructor ***Expr*::*Expr*()** is the expression. The function ***Expr*::*eval*()** returns the value of the expression, and ***Expr*::*print*()** prints a representation

of the expression on ***cout***. A program might look like this:

```
Expr x("123/4+123*4-3");
cout << "x = " << x.eval() << "\n";
x.print();
```

Define class ***Expr*** twice: once using a linked list of nodes as the representation and once using a character string as the representation. Experiment with different ways of printing the expression: fully parenthesized, postfix notation, assembly code, etc.

12. (∗2) Define a class ***Char_queue*** so that the public interface does not depend on the representation. Implement ***Char_queue*** (a) as a linked list and (b) as a vector. Do not worry about concurrency.
13. (∗3) Design a symbol table class and a symbol table entry class for some language. Have a look at a compiler for that language to see what the symbol table really looks like.
14. (∗2) Modify the expression class from §10.6[11] to handle variables and the assignment operator =. Use the symbol table class from §10.6[13].
15. (∗1) Given this program:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

modify it to produce this output:

```
Initialize
Hello, world!
Clean up
```

Do not change ***main*()** in any way.

16. (∗2) Define a ***Calculator*** class for which the calculator functions from §6.1 provide most of the implementation. Create ***Calculator***s and invoke them for input from ***cin***, from command-line arguments, and for strings in the program. Allow output to be delivered to a variety of targets similar to the way input can be obtained from a variety of sources.
17. (∗2) Define two classes, each with a ***static*** member, so that the construction of each ***static*** member involves a reference to the other. Where might such constructs appear in real code? How can these classes be modified to eliminate the order dependence in the constructors?
18. (∗2.5) Compare class ***Date*** (§10.3) with your solution to §5.9[13] and §7.10[19]. Discuss errors found and likely differences in maintenance of the two solutions.
19. (∗3) Write a function that, given an ***istream*** and a ***vector<string>***, produces a ***map<string,vector<int>>*** holding each string and the numbers of the lines on which the string appears. Run the program on a text-file with no fewer than 1,000 lines looking for no fewer than 10 words.
20. (∗2) Take class ***Entry*** from §C.8.2 and modify it so that each union member is always used according to its type.

.

11

Operator Overloading

*When **I** use a word it means just what*
I choose it to mean – neither more nor less.
– Humpty Dumpty

Notation — operator functions — binary and unary operators — predefined meanings for operators — user-defined meanings for operators — operators and namespaces — a complex type — member and nonmember operators — mixed-mode arithmetic — initialization — copying — conversions — literals — helper functions — conversion operators — ambiguity resolution — friends — members and friends — large objects — assignment and initialization — subscripting — function call — dereferencing — increment and decrement — a string class — advice — exercises.

11.1 Introduction [over.intro]

Every technical field – and most nontechnical fields – have developed conventional shorthand notation to make convenient the presentation and discussion involving frequently-used concepts. For example, because of long acquaintance

```
x+y*z
```

is clearer to us than

multiply y by z and add the result to x

It is hard to overestimate the importance of concise notation for common operations.

Like most languages, C++ supports a set of operators for its built-in types. However, most concepts for which operators are conventionally used are not built-in types in C++, so they must be represented as user-defined types. For example, if you need complex arithmetic, matrix algebra, logic signals, or character strings in C++, you use classes to represent these notions. Defining operators

for such classes sometimes allows a programmer to provide a more conventional and convenient notation for manipulating objects than could be achieved using only the basic functional notation. For example,

```
class complex {                // very simplified complex
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) { }
    complex operator+(complex);
    complex operator*(complex);
};
```

defines a simple implementation of the concept of complex numbers. A *complex* is represented by a pair of double-precision floating-point numbers manipulated by the operators + and *. The programmer defines *complex*::*operator*+() and *complex*::*operator**() to provide meanings for + and *, respectively. For example, if *b* and *c* are of type *complex*, *b*+*c* means *b*.*operator*+(*c*). We can now approximate the conventional interpretation of *complex* expressions:

```
void f()
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1,2);
}
```

The usual precedence rules hold, so the second statement means *b*=*b*+(*c**a*), not *b*=(*b*+*c*)**a*.

Many of the most obvious uses of operator overloading are for concrete types (§10.3). However, the usefulness of user-defined operators is not restricted to concrete types. For example, the design of general and abstract interfaces often leads to the use of operators such as ->, [], and ().

11.2 Operator Functions [over.oper]

Functions defining meanings for the following operators (§6.2) can be declared:

```
+      -      *      /      %      ^      &
|      ~      !      =      <      >      +=
-=     *=     /=     %=     ^=     &=     |=
<<     >>     >>=    <<=    ==     !=     <=
>=     &&     ||     ++     --     ->*    ,
->     []     ()     new    new[]  delete delete[]
```

The following operators cannot be defined by a user:

:: (scope resolution; §4.9.4, §10.2.4),
. (member selection; §5.7), and
.* (member selection through pointer to function; §15.5).

They take a name, rather than a value, as their second operand and provide the primary means of referring to members. Allowing them to be overloaded would lead to subtleties [Stroustrup,1994].

It is not possible to define new operator tokens, but you can use the function-call notation when this set of operators is not adequate. For example, use *pow*( ), not **. These restrictions may seem Draconian, but more flexible rules can easily lead to ambiguities. For example, defining an operator ** to mean exponentiation may seem an obvious and easy task at first glance, but think again. Should ** bind to the left (as in Fortran) or to the right (as in Algol)? Should the expression *a***p* be interpreted as *a*(**p*) or as (*a*)**(*p*)?

The name of an operator function is the keyword ***operator*** followed by the operator itself; for example, ***operator*<<**. An operator function is declared and can be called like any other function. A use of the operator is only a shorthand for an explicit call of the operator function. For example:

```
void f(complex a, complex b)
{
    complex c = a + b;              // shorthand
    complex d = a.operator+(b);     // explicit call
}
```

Given the previous definition of ***complex***, the two initializers are synonymous.

11.2.1 Binary and Unary Operators [over.binary]

A binary operator can be defined by either a nonstatic member function taking one argument or a nonmember function taking two arguments. For any binary operator @, ***aa@bb*** can be interpreted as either ***aa.operator@(bb)*** or ***operator@(aa,bb)***. If both are defined, overload resolution (§7.4) determines which, if any, interpretation is used. For example:

```
class X {
public:
    void operator+(int);
    X(int);
};

void operator+(X,X);
void operator+(X,double);

void f(X a)
{
    a+1;        // a.operator+(1)
    1+a;        // ::operator+(X(1),a)
    a+1.0;      // ::operator+(a,1.0)
}
```

A unary operator, whether prefix or postfix, can be defined by either a nonstatic member function taking no arguments or a nonmember function taking one argument. For any prefix unary operator @, ***@aa*** can be interpreted as either ***aa.operator@( )*** or ***operator@(aa)***. If both are defined, overload resolution (§7.4) determines which, if any, interpretation is used. For any postfix unary operator @, ***aa@*** can be interpreted as either ***aa.operator@(int)*** or ***operator@(aa,int)***. This is explained further in §11.11. If both are defined, overload resolution (§7.4) determines which, if

any, interpretation is used. An operator can be declared only for the syntax defined for it in the grammar (§A.5). For example, a user cannot define a unary % or a ternary +. Consider:

```
class X {
    // members (with implicit 'this' pointer):

    X* operator&();        // prefix unary & (address of)
    X operator&(X);        // binary & (and)
    X operator++(int);     // postfix increment (see §11.11)
    X operator&(X,X);      // error: ternary
    X operator/();         // error: unary /
};

// nonmember functions :

X operator-(X);            // prefix unary minus
X operator-(X,X);          // binary minus
X operator--(X&,int);      // postfix decrement
X operator-();             // error: no operand
X operator-(X,X,X);        // error: ternary
X operator%(X);            // error: unary %
```

Operator `[]` is described in §11.8, operator `()` in §11.9, operator `->` in §11.10, operators `++` and `--` in §11.11, and the allocation and deallocation operators in §6.2.6.2, §10.4.11, and §15.6.

11.2.2 Predefined Meanings for Operators [over.predefined]

Only a few assumptions are made about the meaning of a user-defined operator. In particular, *operator*=, *operator*`[]`, *operator*`()`, and *operator*`->` must be nonstatic member functions; this ensures that their first operands will be lvalues (§4.9.6).

The meanings of some built-in operators are defined to be equivalent to some combination of other operators on the same arguments. For example, if *a* is an int, *++a* means *a+=1*, which in turn means *a=a+1*. Such relations do not hold for user-defined operators unless the user happens to define them that way. For example, a compiler will not generate a definition of *Z*`::`*operator*`+=()` from the definitions of *Z*`::`*operator*`+()` and *Z*`::`*operator*`=()`.

Because of historical accident, the operators = (assignment), & (address-of), and `,` (sequencing; §6.2.2) have predefined meanings when applied to class objects. These predefined meanings can be made inaccessible to general users by making them private:

```
class X {
private:
    void operator=(const X&);
    void operator&();
    void operator,(const X&);
    // ...
};
```

```
void f(X a, X b)
{
    a = b;      // error: operator= private
    &a;         // error: operator& private
    a,b;        // error: operator, private
}
```

Alternatively, they can be given new meanings by suitable definitions.

11.2.3 Operators and User-Defined Types [over.user]

An operator function must either be a member or take at least one argument of a user-defined type (functions redefining the *new* and *delete* operators need not). This rule ensures that a user cannot change the meaning of an expression unless the expression contains an object of a user-defined type. In particular, it is not possible to define an operator function that operates exclusively on pointers. This ensures that C++ is extensible but not mutable (with the exception of operators =, &, and , for class objects).

An operator function intended to accept a basic type as its first operand cannot be a member function. For example, consider adding a complex variable *aa* to the integer *2*: *aa+2* can, with a suitably declared member function, be interpreted as *aa.operator+(2)*, but *2+aa* cannot because there is no class *int* for which to define + to mean *2.operator+(aa)*. Even if there were, two different member functions would be needed to cope with *2+aa* and *aa+2*. Because the compiler does not know the meaning of a user-defined +, it cannot assume that it is commutative and so interpret *2+aa* as *aa+2*. This example is trivially handled using nonmember functions (§11.3.2, §11.5).

Enumerations are user-defined types so that we can define operators for them. For example:

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day& operator++(Day& d)
{
    return d = (sat==d) ? sun : Day(d+1);
}
```

Every expression is checked for ambiguities. Where a user-defined operator provides a possible interpretation, the expression is checked according to the rules in §7.4.

11.2.4 Operators in Namespaces [over.namespace]

An operator is either a member of a class or defined in some namespace (possibly the global namespace). Consider this simplified version of string I/O from the standard library:

```
namespace std {             // simplified std

    class ostream {
        // ...
        ostream& operator<<(const char*);
    };

    extern ostream cout;
```

```
        class string {
            // ...
        };

        ostream& operator<<(ostream&, const string&);
    }

    int main()
    {
        char* p = "Hello";
        std::string s = "world";
        std::cout << p << ", " << s << "!\n";
    }
```

Naturally, this writes out ***Hello, world***! But why? Note that I didn't make everything from ***std*** accessible by writing:

```
using namespace std;
```

Instead, I used the ***std*::** prefix for ***string*** and ***cout***. In other words, I was at my best behavior and didn't pollute the global namespace or in other words introduce unnecessary dependencies.

The output operator for C-style strings (***char****) is a member of ***std*::*ostream***, so by definition

```
std::cout << p
```

means

```
std::cout.operator<<(p)
```

However, ***std*::*ostream*** doesn't have a member function to output a ***std*::*string***, so

```
std::cout << s
```

means

```
operator<<(std::cout,s)
```

Operators defined in namespaces can be found based on their operand types just like functions can be found based on their argument types (§8.2.6). In particular, ***cout*** is in namespace ***std***, so ***std*** is considered when looking for a suitable definition of **<<**. In that way, the compiler finds and uses:

```
std::operator<<(std::ostream&, const std::string&)
```

For a binary operator @, ***x@y*** where ***x*** is of type ***X*** and ***y*** is of type ***Y*** is resolved like this:

[1] If ***X*** is a class, determine whether class ***X*** or a base of ***X*** defines ***operator@*** as a member; if so, that is the @ to try to use.

[2] Otherwise,

- look for declarations of @ in the context surrounding ***x@y***; and
- if ***X*** is defined in namespace ***N***, look for declarations of @ in ***N***; and
- if ***Y*** is defined in namespace ***M***, look for declarations of @ in ***M***.

If declarations of ***operator@*** are found in the surrounding context, in ***N***, or in ***M***, we try to use those operators.

In either case, declarations for several ***operator@***s may be found and overload resolution rules

(§7.4) are used to find the best match, if any. This lookup mechanism is applied only if the operator has at least one operand of a user-defined type. Therefore, user-defined conversions (§11.3.2, §11.4) will be considered. Note that a *typedef* name is just a synonym and not a user-defined type (§4.9.7).

11.3 A Complex Number Type [over.complex]

The implementation of complex numbers presented in the introduction is too restrictive to please anyone. For example, from looking at a math textbook we would expect this to work:

```
void f()
{
    complex a = complex(1,2);
    complex b = 3;
    complex c = a+2.3;
    complex d = 2+b;
    complex e = -b-c;
    b = c*2*c;
}
```

In addition, we would expect to be provided with a few additional operators, such as `==` for comparison and `<<` for output, and a suitable set of mathematical functions, such as *sin*( ) and *sqrt*( ).

Class *complex* is a concrete type, so its design follows the guidelines from §10.3. In addition, users of complex arithmetic rely so heavily on operators that the definition of *complex* brings into play most of the basic rules for operator overloading.

11.3.1 Member and Nonmember Operators [over.member]

I prefer to minimize the number of functions that directly manipulate the representation of an object. This can be achieved by defining only operators that inherently modify the value of their first argument, such as `+=`, in the class itself. Operators that simply produce a new value based on the values of its arguments, such as `+`, are then defined outside the class and use the essential operators in their implementation:

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a);    // needs access to representation
    // ...
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b;  // access representation through +=
}
```

Given these declarations, we can write:

```
void f(complex x, complex y, complex z)
{
    complex r1 = x+y+z; // r1 = operator+(x,operator+(y,z))
    complex r2 = x;     // r2 = x
    r2 += y;            // r2.operator+=(y)
    r2 += z;            // r2.operator+=(z)
}
```

Except for possible efficiency differences, the computations of ***r1*** and ***r2*** are equivalent.

Composite assignment operators such as `+=` and `*=` tend to be simpler to define than their ‘‘simple’’ counterparts `+` and `*`. This surprises most people at first, but it follows from the fact that three objects are involved in a `+` operation (the two operands and the result), whereas only two objects are involved in a `+=` operation. In the latter case, run-time efficiency is improved by eliminating the need for temporary variables. For example:

```
inline complex& complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```

does not require a temporary variable to hold the result of the addition and is simple for a compiler to inline perfectly.

A good optimizer will generate close to optimal code for uses of the plain `+` operator also. However, we don’t always have a good optimizer and not all types are as simple as ***complex***, so §11.5 discusses ways of defining operators with direct access to the representation of classes.

11.3.2 Mixed-Mode Arithmetic [over.mixed]

To cope with

```
complex d = 2+b;
```

we need to define operator `+` to accept operands of different types. In Fortran terminology, we need *mixed-mode arithmetic*. We can achieve that simply by adding appropriate versions of the operators:

```
class complex {
    double re, im;

public:
    complex& operator+=(complex a) {
        re += a.re;
        im += a.im;
        return *this;
    }
```

```
        complex& operator+=(double a) {
            re += a;
            return *this;
        }

        // ...
};

complex operator+(complex a, complex b)
{
        complex r = a;
        return r += b;   // calls complex::operator+=(complex)
}

complex operator+(complex a, double b)
{
        complex r = a;
        return r += b;   // calls complex::operator+=(double)
}

complex operator+(double a, complex b)
{
        complex r = b;
        return r += a;   // calls complex::operator+=(double)
}
```

Adding a ***double*** to a complex number is a simpler operation than adding a ***complex***. This is reflected in these definitions. The operations taking ***double*** operands do not touch the imaginary part of a complex number and thus will be more efficient.

Given these declarations, we can write:

```
void f(complex x, complex y)
{
        complex r1 = x+y;     // calls operator+(complex,complex)
        complex r2 = x+2;     // calls operator+(complex,double)
        complex r3 = 2+x;     // calls operator+(double,complex)
}
```

11.3.3 Initialization [over.ctor]

To cope with assignments and initialization of ***complex*** variables with scalars, we need a conversion of a scalar (integer or floating-point number) to a ***complex***. For example:

```
complex b = 3; // should mean b.re=3, b.im=0
```

A constructor taking a single argument specifies a conversion from its argument type to the constructor's type. For example:

```
class complex {
    double re, im;
public:
    complex(double r) :re(r), im(0) { }
    // ...
};
```

The constructor specifies the traditional embedding of the real line in the complex plane.

A constructor is a prescription for creating a value of a given type. The constructor is used when a value of a type is expected and when such a value can be created by a constructor from the value supplied as an initializer or assigned value. Thus, a constructor requiring a single argument need not be called explicitly. For example,

```
complex b = 3;
```

means

```
complex b = complex(3);
```

A user-defined conversion is implicitly applied only if it is unique (§7.4). See §11.7.1 for a way of specifying constructors that can only be explicitly invoked.

Naturally, we still need the constructor that takes two doubles, and a default constructor initializing a *complex* to `(0,0)` is also useful:

```
class complex {
    double re, im;
public:
    complex() : re(0), im(0) { }
    complex(double r) : re(r), im(0) { }
    complex(double r, double i) : re(r), im(i) { }
    // ...
};
```

Using default arguments, we can abbreviate:

```
class complex {
    double re, im;
public:
    complex(double r =0, double i =0) : re(r), im(i) { }
    // ...
};
```

When a constructor is explicitly declared for a type, it is not possible to use an initializer list (§5.7, §4.9.5) as the initializer. For example:

```
complex z1 = { 3 };        // error: complex has a constructor
complex z2 = { 3, 4 };     // error: complex has a constructor
```

11.3.4 Copying [over.copy]

In addition to the explicitly declared constructors, ***complex*** by default gets a copy constructor defined (§10.2.5). A default copy constructor simply copies all members. To be explicit, we could equivalently have written:

```
class complex {
    double re, im;
public:
    complex(const complex& c) : re(c.re), im(c.im) { }
    // ...
};
```

However, for types where the default copy constructor has the right semantics, I prefer to rely on that default. It is less verbose than anything I can write, and people should understand the default. Also, compilers know about the default and its possible optimization opportunities. Furthermore, writing out the memberwise copy by hand is tedious and error-prone for classes with many data members (§10.4.6.3).

I use a reference argument for the copy constructor because I must. The copy constructor defines what copying means – including what copying an argument means – so writing

```
complex::complex(complex c) : re(c.re), im(c.im) { }  // error
```

is an error because any call would have involved an infinite recursion.

For other functions taking ***complex*** arguments, I use value arguments rather than reference arguments. Here, the designer has a choice. From a user's point of view, there is little difference between a function that takes a ***complex*** argument and one that takes a ***const complex&*** argument. This issue is discussed further in §11.6.

In principle, copy constructors are used in simple initializations such as

```
complex x = 2;              // create complex(2); then initialize x with it
complex y = complex(2,0);   // create complex(2,0); then initialize y with it
```

However, the calls to the copy constructor are trivially optimized away. We could equivalently have written:

```
complex x(2);       // initialize x by 2
complex y(2,0);     // initialize x by (2,0)
```

For arithmetic types, such as ***complex***, I like the look of the version using = better. It is possible to restrict the set of values accepted by the = style of initialization compared to the ( ) style by making the copy constructor private (§11.2.2) or by declaring a constructor ***explicit*** (§11.7.1).

Similar to initialization, assignment of two objects of the same class is by default defined as memberwise assignment (§10.2.5). We could explicitly define ***complex::operator=*** to do that. However, for a simple type like ***complex*** there is no reason to do so. The default is just right.

The copy constructor – whether user-defined or compiler-generated – is used not only for the initialization of variables, but also for argument passing, value return, and exception handling (see §11.7). The semantics of these operations is defined to be the semantics of initialization (§7.1, §7.3, §14.2.1).

11.3.5 Constructors and Conversions [over.conv]

We defined three versions of each of the four standard arithmetic operators:

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);
// ...
```

This can get tedious, and what is tedious easily becomes error-prone. What if we had three alternatives for the type of each argument for each function? We would need three versions of each single-argument function, nine versions of each two-argument function, twenty-seven versions of each three-argument function, etc. Often these variants are very similar. In fact, almost all variants involve a simple conversion of arguments to a common type followed by a standard algorithm.

The alternative to providing different versions of a function for each combination of arguments is to rely on conversions. For example, our ***complex*** class provides a constructor that converts a ***double*** to a ***complex***. Consequently, we could simply declare only one version of the equality operator for ***complex***:

```
bool operator==(complex,complex);

void f(complex x, complex y)
{
    x==y;      // means operator==(x,y)
    x==3;      // means operator==(x,complex(3))
    3==y;      // means operator==(complex(3),y)
}
```

There can be reasons for preferring to define separate functions. For example, in some cases the conversion can impose overheads, and in other cases, a simpler algorithm can be used for specific argument types. Where such issues are not significant, relying on conversions and providing only the most general variant of a function – plus possibly a few critical variants – contains the combinatorial explosion of variants that can arise from mixed-mode arithmetic.

Where several variants of a function or an operator exist, the compiler must pick ‘‘the right’’ variant based on the argument types and the available (standard and user-defined) conversions. Unless a best match exists, an expression is ambiguous and is an error (see §7.4).

An object constructed by explicit or implicit use of a constructor is automatic and will be destroyed at the first opportunity (see §10.4.10).

No implicit user-defined conversions are applied to the left-hand side of a `.` (or a `->`). This is the case even when the `.` is implicit. For example:

```
void g(complex z)
{
    3+z;                  // ok: complex(3)+z
    3.operator+=(z);      // error: 3 is not a class object
    3+=z;                 // error: 3 is not a class object
}
```

Thus, you can express the notion that an operator requires an lvalue as their left-hand operand by making that operator a member.

11.3.6 Literals [over.literals]

It is not possible to define literals of a class type in the sense that *1.2* and *12e3* are literals of type *double*. However, literals of the basic types can often be used instead if class member functions are used to provide an interpretation for them. Constructors taking a single argument provide a general mechanism for this. When constructors are simple and inline, it is quite reasonable to think of constructor invocations with literal arguments as literals. For example, I think of *complex(3)* as a literal of type *complex*, even though technically it isn't.

11.3.7 Additional Member Functions [over.additional]

So far, we have provided class *complex* with constructors and arithmetic operators only. That is not quite sufficient for real use. In particular, we often need to be able to examine the value of the real and imaginary parts:

```
class complex {
    double re, im;
public:
    double real() const { return re; }
    double imag() const { return im; }
    // ...
};
```

Unlike the other members of *complex*, *real()* and *imag()* do not modify the value of a *complex*, so they can be declared *const*.

Given *real()* and *imag()*, we can define all kinds of useful operations without granting them direct access to the representation of *complex*. For example:

```
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

Note that we need only to be able to read the real and imaginary parts; writing them is less often needed. If we must do a ''partial update,'' we can:

```
void f(complex& z, double d)
{
    // ...
    z = complex(z.real(), d); // assign d to z.im
}
```

A good optimizer generates a single assignment for that statement.

11.3.8 Helper Functions [over.helpers]

If we put all the bits and pieces together, the *complex* class becomes:

```
class complex {
    double re, im;
public:
    complex(double r=0, double i=0) : re(r), im(i) { }

    double real() const { return re; }
    double imag() const { return im; }

    complex& operator+=(complex);
    complex& operator+=(double);
    // -=, *=, and /=
};
```

In addition, we must provide a number of helper functions:

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);

// -, *, and /

complex operator-(complex);    // unary minus
complex operator+(complex);    // unary plus

bool operator==(complex,complex);
bool operator!=(complex,complex);

istream& operator>>(istream&,complex&); // input
ostream& operator<<(ostream&,complex); // output
```

Note that the members *real*( ) and *imag*( ) are essential for defining the comparisons. The definition of most of the following helper functions similarly relies on *real*( ) and *imag*( ).

We might provide functions to allow users to think in terms of polar coordinates:

```
complex polar(double rho, double theta);
complex conj(complex);

double abs(complex);
double arg(complex);
double norm(complex);

double real(complex);     // for notational convenience
double imag(complex);     // for notational convenience
```

Finally, we must provide an appropriate set of standard mathematical functions:

```
complex acos(complex);
complex asin(complex);
complex atan(complex);
// ...
```

From a user's point of view, the complex type presented here is almost identical to the *complex<double>* found in *<complex>* in the standard library (§22.5).

11.4 Conversion Operators [over.conversion]

Using a constructor to specify type conversion is convenient but has implications that can be undesirable. A constructor cannot specify

[1] an implicit conversion from a user-defined type to a basic type (because the basic types are not classes), or

[2] a conversion from a new class to a previously defined class (without modifying the declaration for the old class).

These problems can be handled by defining a *conversion operator* for the source type. A member function ***X*::*operator T*( )**, where ***T*** is a type name, defines a conversion from ***X*** to ***T***. For example, one could define a 6-bit non-negative integer, ***Tiny***, that can mix freely with integers in arithmetic operations:

```
class Tiny {
        char v;
        void assign(int i) { if (i&~077) throw Bad_range(); v=i; }
public:
        class Bad_range { };

        Tiny(int i) { assign(i); }
        Tiny& operator=(int i) { assign(i); return *this; }

        operator int() const { return v; }     // conversion to int function
};
```

The range is checked whenever a ***Tiny*** is initialized by an ***int*** and whenever an ***int*** is assigned to one. No range check is needed when we copy a ***Tiny***, so the default copy constructor and assignment are just right.

To enable the usual integer operations on ***Tiny*** variables, we define the implicit conversion from ***Tiny*** to ***int***, ***Tiny*::*operator int*( )**. Note that the type being converted to is part of the name of the operator and cannot be repeated as the return value of the conversion function:

```
Tiny::operator int() const { return v; }        // right
int Tiny::operator int() const { return v; }    // error
```

In this respect also, a conversion operator resembles a constructor.

Whenever a ***Tiny*** appears where an ***int*** is needed, the appropriate ***int*** is used. For example:

```
int main()
{
        Tiny c1 = 2;
        Tiny c2 = 62;
        Tiny c3 = c2-c1;        // c3 = 60
        Tiny c4 = c3;           // no range check (not necessary)
        int i = c1+c2;          // i = 64

        c1 = c1+c2;             // range error: c1 can't be 64
        i = c3-64;              // i = –4
        c2 = c3-64;             // range error: c2 can't be –4
        c3 = c4;                // no range check (not necessary)
}
```

Conversion functions appear to be particularly useful for handling data structures when reading (implemented by a conversion operator) is trivial, while assignment and initialization are distinctly less trivial.

The ***istream*** and ***ostream*** types rely on a conversion function to enable statements such as

```
while (cin>>x) cout<<x;
```

The input operation ***cin>>x*** returns an ***istream&***. That value is implicitly converted to a value indicating the state of ***cin***. This value can then be tested by the ***while*** (see §21.3.3). However, it is typically *not* a good idea to define an implicit conversion from one type to another in such a way that information is lost in the conversion.

In general, it is wise to be sparing in the introduction of conversion operators. When used in excess, they lead to ambiguities. Such ambiguities are caught by the compiler, but they can be a nuisance to resolve. Probably the best idea is initially to do conversions by named functions, such as ***X::make_int()***. If such a function becomes popular enough to make explicit use inelegant, it can be replaced by a conversion operator ***X::operator int()***.

If both user-defined conversions and user-defined operators are defined, it is possible to get ambiguities between the user-defined operators and the built-in operators. For example:

```
int operator+(Tiny, Tiny);

void f(Tiny t, int i)
{
        t+i;  // error, ambiguous: operator+(t,Tiny(i)) or int(t)+i ?
}
```

It is therefore often best to rely on user-defined conversions or user-defined operators for a given type, but not both.

11.4.1 Ambiguities [over.ambig]

An assignment of a value of type ***V*** to an object of class ***X*** is legal if there is an assignment operator ***X::operator=(Z)*** so that ***V*** is ***Z*** or there is a unique conversion of ***V*** to ***Z***. Initialization is treated equivalently.

In some cases, a value of the desired type can be constructed by repeated use of constructors or conversion operators. This must be handled by explicit conversions; only one level of user-defined implicit conversion is legal. In some cases, a value of the desired type can be constructed in more than one way; such cases are illegal. For example:

```
class X { /* ... */ X(int); X(char*); };
class Y { /* ... */ Y(int); };
class Z { /* ... */ Z(X); };

X f(X);
Y f(Y);

Z g(Z);
```

```
void k1()
{
    f(1);            // error: ambiguous f(X(1)) or f(Y(1))?
    f(X(1));         // ok
    f(Y(1));         // ok

    g("Mack");       // error: two user-defined conversions needed; g(Z(X("Mack"))) not tried
    g(X("Doc"));     // ok: g(Z(X("Doc")))
    g(Z("Suzy"));    // ok: g(Z(X("Suzy")))
}
```

User-defined conversions are considered only if they are necessary to resolve a call. For example:

```
class XX { /* ... */ XX(int); };

void h(double);
void h(XX);

void k2()
{
    h(1);        // h(double(1)) or h(XX(1))? h(double(1))!
}
```

The call `h(1)` means `h(double(1))` because that alternative uses only a standard conversion rather than a user-defined conversion (§7.4).

The rules for conversion are neither the simplest to implement, the simplest to document, nor the most general that could be devised. They are, however, considerably safer, and the resulting resolutions are less surprising. It is far easier to manually resolve an ambiguity than to find an error caused by an unsuspected conversion.

The insistence on strict bottom-up analysis implies that the return type is not used in overloading resolution. For example:

```
class Quad {
public:
    Quad(double);
    // ...
};

Quad operator+(Quad,Quad);

void f(double a1, double a2)
{
    Quad r1 = a1+a2;          // double-precision add
    Quad r2 = Quad(a1)+a2;    // force quad arithmetic
}
```

The reason for this design choice is partly that strict bottom-up analysis is more comprehensible and partly that it is not considered the compiler's job to decide which precision the programmer might want for the addition.

Once the types of both sides of an initialization or assignment have been determined, both types are used to resolve the initialization or assignment. For example:

```
class Real {
public:
    operator double();
    operator int();
    // ...
};

void g(Real a)
{
    double d = a;    // d = a.double();
    int i = a;       // i = a.int();

    d = a;           // d = a.double();
    i = a;           // i = a.int();
}
```

In these cases, the type analysis is still bottom-up, with only a single operator and its argument types considered at any one time.

11.5 Friends [over.friends]

An ordinary member function declaration specifies three logically distinct things:

[1] The function can access the private part of the class declaration, and
[2] the function is in the scope of the class, and
[3] the function must be invoked on an object (has a ***this*** pointer).

By declaring a member function ***static*** (§10.2.4), we can give it the first two properties only. By declaring a function a ***friend***, we can give it the first property only.

For example, we could define an operator that multiplies a ***Matrix*** by a ***Vector***. Naturally, ***Vector*** and ***Matrix*** each hide their representation and provide a complete set of operations for manipulating objects of their type. However, our multiplication routine cannot be a member of both. Also, we don't really want to provide low-level access functions to allow every user to both read and write the complete representation of both ***Matrix*** and ***Vector***. To avoid this, we declare the ***operator**** a friend of both:

```
class Matrix;

class Vector {
    float v[4];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};

class Matrix {
    Vector v[4];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};
```

```
Vector operator*(const Matrix& m, const Vector& v)
{
    Vector r;
    for (int i = 0; i<4; i++) {        // r[i] = m[i] * v;
        r.v[i] = 0;
        for (int j = 0; j<4; j++) r.v[i] += m.v[i].v[j] * v.v[j];
    }
    return r;
}
```

A ***friend*** declaration can be placed in either the private or the public part of a class declaration; it does not matter where. Like a member function, a friend function is explicitly declared in the declaration of the class of which it is a friend. It is therefore as much a part of that interface as is a member function.

A member function of one class can be the friend of another. For example:

```
class List_iterator {
    // ...
    int* next();
};

class List {
    friend int* List_iterator::next();
    // ...
};
```

It is not unusual for all functions of one class to be friends of another. There is a shorthand for this:

```
class List {
    friend class List_iterator;
    // ...
};
```

This friend declaration makes all of ***List_iterator***'s member functions friends of ***List***.

Clearly, ***friend*** classes should be used only to express closely connected concepts. Often, there is a choice between making a class a member (a nested class) or a friend (§24.4).

11.5.1 Finding Friends [over.lookup]

Like a member declaration, a ***friend*** declaration does not introduce a name into an enclosing scope. For example:

```
class Matrix {
    friend class Xform;
    friend Matrix invert(const Matrix&);
    // ...
};

Xform x;                                  // error: no Xform in scope
Matrix (*p)(const Matrix&) = &invert;     // error: no invert() in scope
```

For large programs and large classes, it is nice that a class doesn't ''quietly'' add names to its

enclosing scope. For a template class that can be instantiated in many different contexts (Chapter 13), this is very important.

A friend class must be previously declared in an enclosing scope or defined in the non-class scope immediately enclosing the class that is declaring it a friend. For example:

```
class X { /* ... */ };              // Y's friend

namespace N {
    class Y {
        friend class X;
        friend class Z;
        friend class AE;
    };
    class Z { /* ... */ };          // Y's friend
}

class AE { /* ... */ };             // not a friend of Y
```

A friend function can be explicitly declared just like friend classes, or it can be found through its argument types (§8.2.6) as if it was declared in the non-class scope immediately enclosing its class. For example:

```
void f(Matrix& m)
{
    invert(m);        // Matrix's friend invert()
}
```

It follows that a friend function should either be explicitly declared in an enclosing scope or take an argument of its class. If not, the friend cannot be called. For example:

```
// no f() here

void g();                           // X's friend

class X {
    friend void f();                // useless
    friend void g();
    friend void h(const X&);        // can be found through its argument
};

void f() { /* ... */ }              // not a friend of X
```

11.5.2 Friends and Members [over.friends.members]

When should we use a friend function, and when is a member function the better choice for specifying an operation? First, we try to minimize the number of functions that access the representation of a class and try to make the set of access functions as appropriate as possible. Therefore, the first question is not, ''Should it be a member, a static member, or a friend?'' but rather, ''Does it really need access?'' Typically, the set of functions that need access is smaller than we are willing to believe at first.

Some operations must be members – for example, constructors, destructors, and virtual

functions (§12.2.6) – but typically there is a choice. Because member names are local to the class, a function should be a member unless there is a specific reason for it to be a nonmember.

Consider a class *X* presenting alternative ways of presenting an operation:

```
class X {
    // ...
    X(int);

    int m1();
    int m2() const;

    friend int f1(X&);
    friend int f2(const X&);
    friend int f3(X);
};
```

Member functions can be invoked for objects of their class only; no user-defined conversions are applied. For example:

```
void g()
{
    99.m1(); // error: X(99).m1() not tried
    99.m2(); // error: X(99).m2() not tried
}
```

The conversion *X*(*int*) is not applied to make an *X* out of *99*.

The global function *f1*() has a similar property because implicit conversions are not used for non-*const* reference arguments (§5.5, §11.3.5). However, conversions may be applied to the arguments of *f2*() and *f3*():

```
void h()
{
    f1(99);    // error: f1(X(99)) not tried
    f2(99);    // ok: f2(X(99));
    f3(99);    // ok: f3(X(99));
}
```

An operation modifying the state of a class object should therefore be a member or a global function taking a non-*const* reference argument (or a non-*const* pointer argument). Operators that require lvalue operands for the fundamental types (=, *=, ++, etc.) are most naturally defined as members for user-defined types.

Conversely, if implicit type conversion is desired for all operands of an operation, the function implementing it must be a nonmember function taking a *const* reference argument or a non-reference argument. This is often the case for the functions implementing operators that do not require lvalue operands when applied to fundamental types (+, -, ||, etc.). Such operators often need access to the representations of their operand class. Consequently, binary operators are the most common source of *friend* functions.

If no type conversions are defined, there appears to be no compelling reason to choose a member over a friend taking a reference argument, or vice versa. In some cases, the programmer may have a preference for one call syntax over another. For example, most people seem to prefer the

notation *inv*(*m*) for inverting a ***Matrix*** ***m*** to the alternative ***m*.*inv*()**. Naturally, if ***inv*()** really does invert ***m*** itself, rather than return a new ***Matrix*** that is the inverse of ***m***, it should be a member.

All other things considered equal, choose a member. It is not possible to know if someone someday will define a conversion operator. It is not always possible to predict if a future change may require changes to the state of the object involved. The member function call syntax makes it clear to the user that the object may be modified; a reference argument is far less obvious. Furthermore, expressions in the body of a member can be noticeably shorter than the equivalent expressions in a global function; a nonmember function must use an explicit argument, whereas the member can use ***this*** implicitly. Also, because member names are local to the class they tend to be shorter than the names of nonmember functions.

11.6 Large Objects [over.large]

We defined the ***complex*** operators to take arguments of type ***complex***. This implies that for each use of a ***complex*** operator, each operand is copied. The overhead of copying two ***doubles*** can be noticeable but often less than what a pair of pointers impose. Unfortunately, not all classes have a conveniently small representation. To avoid excessive copying, one can declare functions to take reference arguments. For example:

```
class Matrix {
    double m[4][4];
public:
    Matrix();
    friend Matrix operator+(const Matrix&, const Matrix&);
    friend Matrix operator*(const Matrix&, const Matrix&);
};
```

References allow the use of expressions involving the usual arithmetic operators for large objects without excessive copying. Pointers cannot be used because it is not possible to redefine the meaning of an operator applied to a pointer. Addition could be defined like this:

```
Matrix operator+(const Matrix& arg1, const Matrix& arg2)
{
    Matrix sum;
    for (int i=0; i<4; i++)
        for (int j=0; j<4; j++)
            sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
    return sum;
}
```

This ***operator+*()** accesses the operands of + through references but returns an object value. Returning a reference would appear to be more efficient:

```
class Matrix {
    // ...
    friend Matrix& operator+(const Matrix&, const Matrix&);
    friend Matrix& operator*(const Matrix&, const Matrix&);
};
```

This is legal, but it causes a memory allocation problem. Because a reference to the result will be passed out of the function as a reference to the return value, the return value cannot be an automatic variable (§7.3). Since an operator is often used more than once in an expression, the result cannot be a ***static*** local variable. The result would typically be allocated on the free store. Copying the return value is often cheaper (in execution time, code space, and data space) than allocating and (eventually) deallocating an object on the free store. It is also much simpler to program.

There are techniques you can use to avoid copying the result. The simplest is to use a buffer of static objects. For example:

```
const max_matrix_temp = 7;

Matrix& get_matrix_temp()
{
    static int nbuf = 0;
    static Matrix buf[max_matrix_temp];

    if (nbuf == max_matrix_temp) nbuf = 0;
    return buf[nbuf++];
}

Matrix& operator+(const Matrix& arg1, const Matrix& arg2)
{
    Matrix& res = get_matrix_temp();
    // ...
    return res;
}
```

Now a ***Matrix*** is copied only when the result of an expression is assigned. However, heaven help you if you write an expression that involves more than ***max_matrix_temp*** temporaries!

A less error-prone technique involves defining the matrix type as a handle (§25.7) to a representation type that really holds the data. In that way, the matrix handles can manage the representation objects in such a way that allocation and copying are minimized (see §11.12 and §11.14[18]). However, that strategy relies on operators returning objects rather than references or pointers. Another technique is to define ternary operations and have them automatically invoked for expressions such as ***a=b+c*** and ***a+b*i*** (§21.4.6.3, §22.4.7).

11.7 Essential Operators [over.essential]

In general, for a type ***X***, the copy constructor ***X(const X&)*** takes care of initialization by an object of the same type ***X***. It cannot be overemphasized that *assignment and initialization are different operations* (§10.4.4.1). This is especially important when a destructor is declared. If a class ***X*** has a destructor that performs a nontrivial task, such as free-store deallocation, the class is likely to need the full complement of functions that control construction, destruction, and copying:

```
class X {
    // ...
    X(Sometype);                // constructor: create objects
    X(const X&);                // copy constructor
    X& operator=(const X&);     // copy assignment: cleanup and copy
    ~X();                       // destructor: cleanup
};
```

There are three more cases in which an object is copied: as a function argument, as a function return value, and as an exception. When an argument is passed, a hitherto uninitialized variable – the formal parameter – is initialized. The semantics are identical to those of other initializations. The same is the case for function return values and exceptions, although that is less obvious. In such cases, the copy constructor will be applied. For example:

```
string g(string arg)
{
    return arg;
}

int main ()
{
    string s = "Newton";
    s = g(s);
}
```

Clearly, the value of *s* ought to be *"Newton"* after the call of *g*( ). Getting a copy of the value of *s* into the argument *arg* is not difficult; a call of *string*'s copy constructor does that. Getting a copy of that value out of *g*( ) takes another call of *string*(*const string*&); this time, the variable initialized is a temporary one, which is then assigned to *s*. Often one, but not both, of these copy operations can be optimized away. Such temporary variables are, of course, destroyed properly using *string*::~*string*( ) (see §10.4.10).

For a class *X* for which the assignment operator *X*::*operator*=(*const X*&) and the copy constructor *X*::*X*(*const X*&) are not explicitly declared by the programmer, the missing operation or operations will be generated by the compiler (§10.2.5).

11.7.1 Explicit Constructors [over.explicit]

By default, a single argument constructor also defines an implicit conversion. For some types, that is ideal. For example:

```
complex z = 2; // initialize z with complex(2)
```

In other cases, the implicit conversion is undesirable and error-prone. For example:

```
string s = 'a'; // make s a string with int('a') elements
```

It is quite unlikely that this was what the person defining *s* meant.

Implicit conversion can be suppressed by declaring a constructor *explicit*. That is, an *explicit* constructor will be invoked only explicitly. In particular, where a copy constructor is in principle needed (§11.3.4), an *explicit* constructor will not be implicitly invoked. For example:

```
class String {
    // ...
    explicit String(int n);      // preallocate n bytes
    String(const char* p);       // initial value is the C-style string p
};

String s1 = 'a';                 // error: no implicit char->String conversion
String s2(10);                   // ok: String with space for 10 characters
String s3 = String(10);          // ok: String with space for 10 characters
String s4 = "Brian";             // ok: s4 = String("Brian")
String s5("Fawlty");

void f(String);

String g()
{
    f(10);                 // error: no implicit int->String conversion
    f(String(10));
    f("Arthur");           // ok: f(String("Arthur"))
    f(s1);

    String* p1 = new String("Eric");
    String* p2 = new String(10);

    return 10;             // error: no implicit int->String conversion
}
```

The distinction between

```
String s1 = 'a';           // error: no implicit char->String conversion
```

and

```
String s2(10);             // ok: string with space for 10 characters
```

may seem subtle, but it is less so in real code than in contrived examples.

In ***Date***, we used a plain ***int*** to represent a year (§10.3). Had ***Date*** been critical in our design, we might have introduced a ***Year*** type to allow stronger compile-time checking. For example:

```
class Year {
    int y;
public:
    explicit Year(int i) : y(i) { }        // construct Year from int
    operator int() const { return y; }     // conversion: Year to int
};

class Date {
public:
    Date(int d, Month m, Year y);
    // ...
};

Date d3(1978,feb,21);          // error: 21 is not a Year
Date d4(21,feb,Year(1978));    // ok
```

The ***Year*** class is a simple ''wrapper'' around an ***int***. Thanks to the ***operator int*()**, a ***Year*** is implicitly converted into an ***int*** wherever needed. By declaring the constructor ***explicit***, we make sure that the ***int*** to ***Year*** happens only when we ask for it and that ''accidental'' assignments are caught at compile time. Because ***Year***'s member functions are easily inlined, no run-time or space costs are added.

A similar technique can be used to define range types (§25.6.1).

11.8 Subscripting [over.subscript]

An ***operator*[]** function can be used to give subscripts a meaning for class objects. The second argument (the subscript) of an ***operator*[]** function may be of any type. This makes it possible to define ***vector***s, associative arrays, etc.

As an example, let us recode the example from §5.5 in which an associative array is used to write a small program for counting the number of occurrences of words in a file. There, a function is used. Here, an associative array type is defined:

```
class Assoc {
    struct Pair {
        string name;
        double val;
        Pair(string n ="", double v =0) :name(n), val(v) { }
    };
    vector<Pair> vec;

    Assoc(const Assoc&);                    // private to prevent copying
    Assoc& operator=(const Assoc&);         // private to prevent copying
public:
    Assoc() {}
    double& operator[](const string&);
    void print_all() const;
};
```

An ***Assoc*** keeps a vector of ***Pair***s. The implementation uses the same trivial and inefficient search method as in §5.5:

```
double& Assoc::operator[](const string& s)
    // search for s; return its value if found; otherwise, make a new Pair and return the default value 0
{
    for (vector<Pair>::const_iterator p = vec.begin(); p!=vec.end(); ++p)
        if (s == p->name) return p->val;

    vec.push_back(Pair(s,0));       // initial value: 0

    return vec.back().val;          // return last element (§16.3.3)
}
```

Because the representation of an ***Assoc*** is hidden, we need a way of printing it:

```
void Assoc::print_all() const
{
    for (vector<Pair>::const_iterator p = vec.begin(); p!=vec.end(); ++p)
        cout << p->name << ": " << p->val << '\n';
}
```

Finally, we can write the trivial main program:

```
int main()      // count the occurrences of each word on input
{
    string buf;
    Assoc vec;
    while (cin>>buf) vec[buf]++;
    vec.print_all();
}
```

A further development of the idea of an associative array can be found in §17.4.1.

An `operator[]()` must be a member function.

11.9 Function Call [over.call]

Function call, that is, the notation *expression(expression-list)*, can be interpreted as a binary operation with the *expression* as the left-hand operand and the *expression-list* as the right-hand operand. The call operator `()` can be overloaded in the same way as other operators can. An argument list for an `operator()()` is evaluated and checked according to the usual argument-passing rules. Overloading function call seems to be useful primarily for defining types that have only a single operation and for types for which one operation is predominant.

The most obvious, and probably also the most important, use of the `()` operator is to provide the usual function call syntax for objects that in some way behave like functions. An object that acts like a function is often called a *function-like object* or simply a *function object* (§18.4). Such function objects are important because they allow us to write code that takes nontrivial operations as parameters. For example, the standard library provides many algorithms that invoke a function for each element of a container. Consider:

```
void negate(complex& c) { c = -c; }

void f(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), negate);  // negate all vector elements
    for_each(ll.begin(), ll.end(), negate);  // negate all list elements
}
```

This negates every element in the vector and the list.

What if we wanted to add `complex(2,3)` to every element? That is easily done like this:

```
void add23(complex& c)
{
    c += complex(2,3);
}

void g(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), add23);
    for_each(ll.begin(), ll.end(), add23);
}
```

How would we write a function to repeatedly add an arbitrary complex value? We need something to which we can pass that arbitrary value and which can then use that value each time it is called. That does not come naturally for functions. Typically, we end up ''passing'' the arbitrary value by leaving it in the function's surrounding context. That's messy. However, we can write a class that behaves in the desired way:

```
class Add {
    complex val;
public:
    Add(complex c) { val = c; }                          // save value
    Add(double r, double i) { val = complex(r,i); }

    void operator()(complex& c) const { c += val; }      // add value to argument
};
```

An object of class *Add* is initialized with a complex number, and when invoked using `()`, it adds that number to its argument. For example:

```
void h(vector<complex>& aa, list<complex>& ll, complex z)
{
    for_each(aa.begin(), aa.end(), Add(2,3));
    for_each(ll.begin(), ll.end(), Add(z));
}
```

This will add `complex(2,3)` to every element of the array and `z` to every element on the list. Note that `Add(z)` constructs an object that is used repeatedly by `for_each()`. It is not simply a function that is called once or even called repeatedly. The function that is called repeatedly is `Add(z)`'s `operator()()`.

This all works because *for_each* is a template that applies `()` to its third argument without caring exactly what that third argument really is:

```
template<class Iter, class Fct> Iter for_each(Iter b, Iter e, Fct f)
{
    while (b != e) f(*b++);
    return b;
}
```

At first glance, this technique may look esoteric, but it is simple, efficient, and extremely useful (see §3.8.5, §18.4).

Other popular uses of *operator*( )( ) are as a substring operator and as a subscripting operator for multidimensional arrays (§22.4.5).

An *operator*( )( ) must be a member function.

11.10 Dereferencing [over.deref]

The dereferencing operator -> can be defined as a unary postfix operator. That is, given a class

```
class Ptr {
    // ...
    X* operator->();
};
```

objects of class *Ptr* can be used to access members of class *X* in a very similar manner to the way pointers are used. For example:

```
void f(Ptr p)
{
    p->m = 7;        // (p.operator–>())–>m = 7
}
```

The transformation of the object *p* into the pointer *p*.*operator*->( ) does not depend on the member *m* pointed to. That is the sense in which *operator*->( ) is a unary postfix operator. However, there is no new syntax introduced, so a member name is still required after the ->. For example:

```
void g(Ptr p)
{
    X* q1 = p->;                  // syntax error
    X* q2 = p.operator->();       // ok
}
```

Overloading -> is primarily useful for creating ‘‘smart pointers,’’ that is, objects that act like pointers and in addition perform some action whenever an object is accessed through them. For example, one could define a class *Rec_ptr* for accessing objects of class *Rec* stored on disk. *Rec_ptr*’s constructor takes a name that can be used to find the object on disk, *Rec_ptr*::*operator*->( ) brings the object into main memory when accessed through its *Rec_ptr*, and *Rec_ptr*’s destructor eventually writes the updated object back out to disk:

```
class Rec_ptr {
    Rec* in_core_address;
    const char* identifier;
    // ...

public:
    Rec_ptr(const char* p) : identifier(p), in_core_address(0) { }
    ~Rec_ptr( ) { write_to_disk(in_core_address, identifier); }
    Rec* operator->();
};
```

```
Rec* Rec_ptr::operator->()
{
    if (in_core_address == 0) in_core_address = read_from_disk(identifier);
    return in_core_address;
}
```

Rec_ptr might be used like this:

```
struct Rec {    // the Rec that a Rec_ptr points to
    string name;
    // ...
};

void update(const char* s)
{
    Rec_ptr p(s);               // get Rec_ptr for s

    p->name = "Roscoe";         // update s; if necessary, first retrieve from disk
    // ...
}
```

Naturally, a real *Rec_ptr* would be a template so that the *Rec* type is a parameter. Also, a realistic program would contain error-handling code and use a less naive way of interacting with the disk.

For ordinary pointers, use of `->` is synonymous with some uses of unary `*` and `[]`. Given

```
Y* p;
```

it holds that

```
p->m == (*p).m == p[0].m
```

As usual, no such guarantee is provided for user-defined operators. The equivalence can be provided where desired:

```
class Ptr_to_Y {
    Y* p;
public:
    Y* operator->() { return p; }
    Y& operator*() { return *p; }
    Y& operator[](int i) { return p[i]; }
};
```

If you provide more than one of these operators, it might be wise to provide the equivalence, just as it is wise to ensure that `++x` and `x+=1` have the same effect as `x=x+1` for a simple variable `x` of some class if `++`, `+=`, `=`, and `+` are provided.

The overloading of `->` is important to a class of interesting programs and not just a minor curiosity. The reason is that *indirection* is a key concept and that overloading `->` provides a clean, direct, and efficient way of representing indirection in a program. Iterators (Chapter 19) provide an important example of this. Another way of looking at operator `->` is to consider it as a way of providing C++ with a limited, but useful, form of *delegation* (§24.2.4).

Operator `->` must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply `->`. When declared for a template class, *operator*`->()` is

frequently unused, so it makes sense to postpone checking the constraint on the return type until actual use.

11.11 Increment and Decrement [over.incr]

Once people invent ‘‘smart pointers,’’ they often decide to provide the increment operator ++ and the decrement operator -- to mirror these operators’ use for built-in types. This is especially obvious and necessary where the aim is to replace an ordinary pointer type with a ‘‘smart pointer’’ type that has the same semantics, except that it adds a bit of run-time error checking. For example, consider a troublesome traditional program:

```
void f1(T a)            // traditional use
{
        T v[200];
        T* p = &v[0];
        p--;
        *p = a;     // Oops: ‘p’ out of range, uncaught
        ++p;
        *p = a;     // ok
}
```

We might want to replace the pointer ***p*** with an object of a class ***Ptr_to_T*** that can be dereferenced only provided it actually points to an object. We would also like to ensure that ***p*** can be incremented and decremented, only provided it points to an object within an array and the increment and decrement operations yield an object within the array. That is we would like something like this:

```
class Ptr_to_T {
        // ...
};

void f2(T a)            // checked
{
        T v[200];
        Ptr_to_T p(&v[0],v,200);
        p--;
        *p = a;     // run-time error: ‘p’ out of range
        ++p;
        *p = a;     // ok
}
```

The increment and decrement operators are unique among C++ operators in that they can be used as both prefix and postfix operators. Consequently, we must define prefix and postfix increment and decrement ***Ptr_to_T***. For example:

```
class Ptr_to_T {
        T* p;
        T* array;
        int size;
public:

        Ptr_to_T(T* p, T* v, int s);     // bind to array v of size s, initial value p
        Ptr_to_T(T* p);                  // bind to single object, initial value p

        Ptr_to_T& operator++();          // prefix
        Ptr_to_T operator++(int);        // postfix

        Ptr_to_T& operator--();          // prefix
        Ptr_to_T operator--(int);        // postfix

        T& operator*();        // prefix
};
```

The ***int*** argument is used to indicate that the function is to be invoked for postfix application of ++. This ***int*** is never used; the argument is simply a dummy used to distinguish between prefix and postfix application. The way to remember which version of an ***operator++*** is prefix is to note that the version without the dummy argument is prefix, exactly like all the other unary arithmetic and logical operators. The dummy argument is used only for the ''odd'' postfix ++ and --.

Using ***Ptr_to_T***, the example is equivalent to:

```
void f3(T a)          // checked
{
        T v[200];
        Ptr_to_T p(&v[0],v,200);
        p.operator--(0);
        p.operator*() = a;   // run-time error: 'p' out of range
        p.operator++();
        p.operator*() = a;   // ok
}
```

Completing class ***Ptr_to_T*** is left as an exercise (§11.14[19]). Its elaboration into a template using exceptions to report the run-time errors is another exercise (§14.12[2]). An example of operators ++ and -- for iteration can be found in §19.3. A pointer template that behaves correctly with respect to inheritance is presented in (§13.6.3).

11.12 A String Class [over.string]

Here is a more realistic version of class ***String***. I designed it as the minimal string that served my needs. This string provides value semantics, character read and write operations, checked and unchecked access, stream I/O, literal strings as literals, and equality and concatenation operators. It represents strings as C-style, zero-terminated arrays of characters and uses reference counts to minimize copying. Writing a better string class and/or one that provides more facilities is a good exercise (§11.14[7-12]). That done, we can throw away our exercises and use the standard library string (Chapter 20).

My almost-real ***String*** employs three auxiliary classes: ***Srep***, to allow an actual representation to be shared between several ***String***s with the same value; ***Range***, to be thrown in case of range errors, and ***Cref***, to help implement a subscript operator that distinguishes between reading and writing:

```
class String {
    struct Srep;                // representation
    Srep *rep;
public:
    class Cref;                 // reference to char

    class Range { };            // for exceptions

    // ...
};
```

Like other members, a *member class* (often called a *nested class*) can be declared in the class itself and defined later:

```
struct String::Srep {
    char* s;        // pointer to elements
    int sz;         // number of characters
    int n;          // reference count

    Srep(int nsz, const char* p)
    {
        n = 1;
        sz = nsz;
        s = new char[sz+1]; // add space for terminator
        strcpy(s,p);
    }

    ~Srep() { delete[] s; }

    Srep* get_own_copy()    // clone if necessary
    {
        if (n==1) return this;
        n--;
        return new Srep(sz,s);
    }

    void assign(int nsz, const char* p)
    {
        if (sz != nsz) {
            delete[] s;
            sz = nsz;
            s = new char[sz+1];
        }
        strcpy(s,p);
    }
```

```
private:                    // prevent copying:
        Srep(const Srep&);
        Srep& operator=(const Srep&);
};
```

Class ***String*** provides the usual set of constructors, destructor, and assignment operations:

```
class String {
        // ...

        String();                   // x = ""
        String(const char*);        // x = "abc"
        String(const String&);      // x = other_string
        String& operator=(const char *);
        String& operator=(const String&);
        ~String();

        // ...
};
```

This ***String*** has value semantics. That is, after an assignment ***s1=s2***, the two strings ***s1*** and ***s2*** are fully distinct and subsequent changes to the one have no effect on the other. The alternative would be to give ***String*** pointer semantics. That would be to let changes to ***s2*** after ***s1=s2*** also affect the value of ***s1***. For types with conventional arithmetic operations, such as complex, vector, matrix, and string, I prefer value semantics. However, for the value semantics to be affordable, a ***String*** is implemented as a handle to its representation and the representation is copied only when necessary:

```
String::String()            // the empty string is the default value
{
        rep = new Srep(0,"");
}

String::String(const String& x) // copy constructor
{
        x.rep->n++;
        rep = x.rep;    // share representation
}

String::~String()
{
        if (--rep->n == 0) delete rep;
}

String& String::operator=(const String& x)      // copy assignment
{
        x.rep->n++;                     // protects against ‘‘st = st’’
        if (--rep->n == 0) delete rep;
        rep = x.rep;                    // share representation
        return *this;
}
```

Pseudo-copy operations taking ***const char**** arguments are provided to allow string literals:

```
String::String(const char* s)
{
    rep = new Srep(strlen(s), s);
}

String& String::operator=(const char* s)
{
    if (rep->n == 1)                    // recycle Srep
        rep->assign(strlen(s), s);
    else {                              // use new Srep
        rep->n--;
        rep = new Srep(strlen(s), s);
    }
    return *this;
}
```

The design of access operators for a string is a difficult topic because ideally access is by conventional notation (that is, using `[]`), maximally efficient, and range checked. Unfortunately, you cannot have all of these properties simultaneously. My choice here has been to provide efficient unchecked operations with a slightly inconvenient notation plus slightly less efficient checked operators with the conventional notation:

```
class String {
    // ...

    void check(int i) const { if (i<0 || rep->sz<=i) throw Range(); }

    char read(int i) const { return rep->s[i]; }
    void write(int i, char c) { rep=rep->get_own_copy(); rep->s[i]=c; }

    Cref operator[](int i) { check(i); return Cref(*this, i); }
    char operator[](int i) const { check(i); return rep->s[i]; }

    int size() const { return rep->sz; }

    // ...
};
```

The idea is to use `[]` to get checked access for ordinary use, but to allow the user to optimize by checking the range once for a set of accesses. For example:

```
int hash(const String& s)
{
    int h = s.read(0);
    const int max = s.size();
    for (int i = 1; i<max; i++) h ^= s.read(i)>>1; // unchecked access to s
    return h;
}
```

Defining an operator, such as `[]`, to be used for both reading and writing is difficult where it is not acceptable simply to return a reference and let the user decide what to do with it. Here, that is not a reasonable alternative because I have defined ***String*** so that the representation is shared between ***String***s that have been assigned, passed as value arguments, etc., until someone actually writes to a

String. Then, and only then, is the representation copied. This technique is usually called *copy-on-write*. The actual copy is done by *String*::*Srep*::*get_own_copy*( ).

To get these access functions inlined, their definitions must be placed so that the definition of *Srep* is in scope. This implies that either *Srep* is defined within *String* or the access functions are defined *inline* outside *String* and after *String*::*Srep* (§11.14[2]).

To distinguish between a read and a write, *String*::*operator*[ ]( ) returns a *Cref* when called for a non-*const* object. A *Cref* behaves like a *char*&, except that it calls *String*::*Srep*::*get_own_copy*( ) when written to:

```
class String::Cref {          // reference to s[i]
friend class String;
        String& s;
        int i;
        Cref(String& ss, int ii) : s(ss), i(ii) { }
public:
        operator char() { return s.read(i); }          // yield value
        void operator=(char c) { s.write(i,c); }       // change value
};
```

For example:

```
void f(String s, const String& r)
{
        int c1 = s[1];   // c1 = s.operator[](1).operator char()
        s[1] = 'c';      // s.operator[](1).operator='c')

        int c2 = r[1];   // c2 = r.operator[](1)
        r[1] = 'd';      // error: assignment to char, r.operator[](1) = 'd'
}
```

Note that for a non-*const* object *s*.*operator*[ ](*1*) is *Cref*(*s*, *1*).

To complete class *String*, I provide a set of useful functions:

```
class String {
        // ...

        String& operator+=(const String&);
        String& operator+=(const char*);

        friend ostream& operator<<(ostream&, const String&);
        friend istream& operator>>(istream&, String&);

        friend bool operator==(const String& x, const char* s)
                { return strcmp(x.rep->s, s) == 0; }

        friend bool operator==(const String& x, const String& y)
                { return strcmp(x.rep->s, y.rep->s) == 0; }

        friend bool operator!=(const String& x, const char* s)
                { return strcmp(x.rep->s, s) != 0; }
```

```
        friend bool operator!=(const String& x, const String& y)
            { return strcmp(x.rep->s, y.rep->s) != 0; }
};

String operator+(const String&, const String&);
String operator+(const String&, const char*);
```

To save space, I have left the I/O and concatenation operations as exercises.

The main program simply exercises the ***String*** operators a bit:

```
String f(String a, String b)
{
        a[2] = 'x';
        char c = b[3];
        cout << "in f: " << a << ' ' << b << ' ' << c << '\n';
        return b;
}

int main()
{
        String x, y;
        cout << "Please enter two strings\n";
        cin >> x >> y;
        cout << "input: " << x << ' ' << y << '\n';
        String z = x;
        y = f(x,y);
        if (x != z) cout << "x corrupted!\n";
        x[0] = '!';
        if (x == z) cout << "write failed!\n";
        cout << "exit: " << x << ' ' << y << ' ' << z << '\n';
}
```

This ***String*** lacks many features that you might consider important or even essential. For example, it offers no operation of producing a C-string representation of its value (§11.14[10], Chapter 20).

11.13 Advice [class.advice]

[1] Define operators primarily to mimic conventional usage; §11.1.
[2] For large operands, use ***const*** reference argument types; §11.6.
[3] For large results, consider optimizing the return; §11.6.
[4] Prefer the default copy operations if appropriate for a class; §11.3.4.
[5] Redefine or prohibit copying if the default is not appropriate for a type; §11.2.2.
[6] Prefer member functions over nonmembers for operations that need access to the representation; §11.5.2.
[7] Prefer nonmember functions over members for operations that do not need access to the representation; §11.5.2.
[8] Use namespaces to associate helper functions with ‘‘their’’ class; §11.2.4.
[9] Use nonmember functions for symmetric operators; §11.3.2.
[10] Use `()` for subscripting multidimensional arrays; §11.9.

[11] Make constructors that take a single ''size argument'' ***explicit***; §11.7.1.
[12] For non-specialized uses, prefer the standard ***string*** (Chapter 20) to the result of your own exercises; §11.12.
[13] Be cautious about introducing implicit conversions; §11.4.
[14] Use member functions to express operators that require an lvalue as its left-hand operand; §11.3.5.

11.14 Exercises [over.exercises]

1. (∗2) In the following program, which conversions are used in each expression?

```
struct X {
    int i;
    X(int);
    operator+(int);
};

struct Y {
    int i;
    Y(X);
    operator+(X);
    operator int();
};

extern X operator*(X, Y);
extern int f(X);

X x = 1;
Y y = x;
int i = 2;

int main()
{
    i + 10;      y + 10;      y + 10 * y;
    x + y + i;   x * x + i;   f(7);
    f(y);        y + y;       106 + y;
}
```

   Modify the program so that it will run and print the values of each legal expression.
2. (∗2) Complete and test class ***String*** from §11.12.
3. (∗2) Define a class ***INT*** that behaves exactly like an ***int***. Hint: Define ***INT*****::*****operator int*****()**.
4. (∗1) Define a class ***RINT*** that behaves like an ***int*** except that the only operations allowed are **+** (unary and binary), **-** (unary and binary), *****, **/**, and **%**. Hint: Do not define ***RINT*****::*****operator int*****()**.
5. (∗3) Define a class ***LINT*** that behaves like a ***RINT***, except that it has at least 64 bits of precision.
6. (∗4) Define a class implementing arbitrary precision arithmetic. Test it by calculating the factorial of ***1000***. Hint: You will need to manage storage in a way similar to what was done for class ***String***.

7. (*2) Define an external iterator for class ***String***:

```
class String_iter {
        // refer to string and string element
public:
        String_iter(String& s);          // iterator for s
        char& next();                    // reference to next element

        // more operations of your choice
};
```

   Compare this in utility, programming style, and efficiency to having an internal iterator for ***String*** (that is, a notion of a current element for the ***String*** and operations relating to that element).
8. (*1.5) Provide a substring operator for a string class by overloading `()`. What other operations would you like to be able to do on a string?
9. (*3) Design class ***String*** so that the substring operator can be used on the left-hand side of an assignment. First, write a version in which a string can be assigned to a substring of the same length. Then, write a version in which the lengths may differ.
10. (*2) Define an operation for ***String*** that produces a C-string representation of its value. Discuss the pros and cons of having that operation as a conversion operator. Discuss alternatives for allocating the memory for that C-string representation.
11. (*2.5) Define and implement a simple regular expression pattern match facility for class ***String***.
12. (*1.5) Modify the pattern match facility from §11.14[11] to work on the standard library ***string***. Note that you cannot modify the definition of ***string***.
13. (*2) Write a program that has been rendered unreadable through use of operator overloading and macros. An idea: Define `+` to mean `-` and vice versa for ***INTs***. Then, use a macro to define ***int*** to mean ***INT***. Redefine popular functions using reference type arguments. Writing a few misleading comments can also create great confusion.
14. (*3) Swap the result of §11.14[13] with a friend. Without running it, figure out what your friend's program does. When you have completed this exercise, you'll know what to avoid.
15. (*2) Define a type ***Vec4*** as a vector of four ***float***s. Define ***operator***`[]` for ***Vec4***. Define operators `+`, `-`, `*`, `/`, `=`, `+=`, `-=`, `*=`, and `/=` for combinations of vectors and floating-point numbers.
16. (*3) Define a class ***Mat4*** as a vector of four ***Vec4***s. Define ***operator***`[]` to return a ***Vec4*** for ***Mat4***. Define the usual matrix operations for this type. Define a function doing Gaussian elimination for a ***Mat4***.
17. (*2) Define a class ***Vector*** similar to ***Vec4*** but with the size given as an argument to the constructor ***Vector***`::`***Vector***`(`***int***`)`.
18. (*3) Define a class ***Matrix*** similar to ***Mat4*** but with the dimensions given as arguments to the constructor ***Matrix***`::`***Matrix***`(`***int***`,` ***int***`)`.
19. (*2) Complete class ***Ptr_to_T*** from §11.11 and test it. To be complete, ***Ptr_to_T*** must have at least the operators `*`, `->`, `=`, `++`, and `--` defined. Do not cause a run-time error until a wild pointer is actually dereferenced.

20. (∗1) Given two structures:

```
struct S { int x, y; };
struct T { char* p; char* q; };
```

write a class *C* that allows the use of *x* and *p* from some *S* and *T*, much as if *x* and *p* had been members of *C*.

21. (∗1.5) Define a class *Index* to hold the index for an exponentiation function *mypow*(*double*, *Index*). Find a way to have *2**I* call *mypow*(*2*, *I*).

22. (∗2) Define a class *Imaginary* to represent imaginary numbers. Define class *Complex* based on that. Implement the fundamental arithmetic operators.

12

Derived Classes

Do not multiply objects without necessity.
– W. Occam

Concepts and classes — derived classes — member functions — construction and destruction — class hierarchies — type fields — virtual functions — abstract classes — traditional class hierarchies — abstract classes as interfaces — localizing object creation — abstract classes and class hierarchies — advice — exercises.

12.1 Introduction [derived.intro]

From Simula, C++ borrowed the concept of a class as a user-defined type and the concept of class hierarchies. In addition, it borrowed the idea for system design that classes should be used to model concepts in the programmer's and the application's world. C++ provides language constructs that directly support these design notions. Conversely, using the language features in support of design concepts distinguishes effective use of C++. Using language constructs only as notational props for more traditional types of programming is to miss key strengths of C++.

A concept does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with related concepts. For example, try to explain what a car is. Soon you'll have introduced the notions of wheels, engines, drivers, pedestrians, trucks, ambulances, roads, oil, speeding tickets, motels, etc. Since we use classes to represent concepts, the issue becomes how to represent relationships between concepts. However, we can't express arbitrary relationships directly in a programming language. Even if we could, we wouldn't want to. Our classes should be more narrowly defined than our everyday concepts – and more precise. The notion of a derived class and its associated language mechanisms are provided to express hierarchical relationships, that is, to express commonality between classes. For example, the concepts of a circle and a triangle are related in that they are both shapes; that is, they have the concept of a shape in common. Thus, we must explicitly define class ***Circle*** and class ***Triangle*** to have class ***Shape*** in

common. Representing a circle and a triangle in a program without involving the notion of a shape would be to lose something essential. This chapter is an exploration of the implications of this simple idea, which is the basis for what is commonly called object-oriented programming.

The presentation of language features and techniques progress from the simple and concrete to the more sophisticated and abstract. For many programmers, this will also be a progression from the familiar towards the less well known. This is not a simple journey from ''bad old techniques'' towards ''the one right way.'' When I point out limitations of one technique as a motivation for another, I do so in the context of specific problems; for different problems or in other contexts, the first technique may indeed be the better choice. Useful software has been constructed using all of the techniques presented here. The aim is to help you attain sufficient understanding of the techniques to be able to make intelligent and balanced choices among them for real problems.

In this chapter, I first introduce the basic language features supporting object-oriented programming. Next, the use of those features to develop well-structured programs is discussed in the context of a larger example. Further facilities supporting object-oriented programming, such as multiple inheritance and run-time type identification, are discussed in Chapter 15.

12.2 Derived Classes [derived.derived]

Consider building a program dealing with people employed by a firm. Such a program might have a data structure like this:

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
```

Next, we might try to define a manager:

```
struct Manager {
    Employee emp;              // manager's employee record
    set<Employee*> group;      // people managed
    short level;
    // ...
};
```

A manager is also an employee; the ***Employee*** data is stored in the ***emp*** member of a ***Manager*** object. This may be obvious to a human reader – especially a careful reader – but there is nothing that tells the compiler and other tools that ***Manager*** is also an ***Employee***. A ***Manager**** is not an ***Employee****, so one cannot simply use one where the other is required. In particular, one cannot put a ***Manager*** onto a list of ***Employee***s without writing special code. We could either use explicit type conversion on a ***Manager**** or put the address of the ***emp*** member onto a list of ***employee***s. However, both solutions are inelegant and can be quite obscure. The correct approach is to explicitly state that a ***Manager*** *is* an ***Employee***, with a few pieces of information added:

```
struct Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
};
```

The ***Manager*** is *derived* from ***Employee***, and conversely, ***Employee*** is a *base class* for ***Manager***. The class ***Manager*** has the members of class ***Employee*** (***name***, ***age***, etc.) in addition to its own members (***group***, ***level***, etc.).

Derivation is often represented graphically by a pointer from the derived class to its base class indicating that the derived class refers to its base (rather than the other way around):

A derived class is often said to inherit properties from its base, so the relationship is also called *inheritance*. A base class is sometimes called a *superclass* and a derived class a *subclass*. This terminology, however, is confusing to people who observe that the data in a derived class object is a superset of the data of an object of its base class. A derived class is larger than its base class in the sense that it holds more data and provides more functions.

A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end. For example:

Deriving ***Manager*** from ***Employee*** in this way makes ***Manager*** a subtype of ***Employee*** so that a ***Manager*** can be used wherever an ***Employee*** is acceptable. For example, we can now create a list of ***Employee***s, some of whom are ***Manager***s:

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist;

    elist.push_front(&m1);
    elist.push_front(&e1);
    // ...
}
```

A ***Manager*** is (also) an ***Employee***, so a ***Manager**** can be used as a ***Employee****. However, an ***Employee*** is not necessarily a ***Manager***, so an ***Employee**** cannot be used as a ***Manager****. In general, if a class ***Derived*** has a public base class (§15.3) ***Base***, then a ***Derived**** can be assigned to a variable of type ***Base**** without the use of explicit type conversion. The opposite conversion, from ***Base**** to ***Derived****, must be explicit. For example:

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;        // ok: every Manager is an Employee
    Manager* pm = &ee;         // error: not every Employee is a Manager

    pm->level = 2;             // disaster: ee doesn't have a 'level'

    pm = static_cast<Manager*>(pe);    // brute force: works because pe points
                                       // to the Manager mm

    pm->level = 2;         // fine: pm points to the Manager mm that has a 'level'
}
```

In other words, an object of a derived class can be treated as an object of its base class when manipulated through pointers and references. The opposite is not true. The use of ***static_cast*** and ***dynamic_cast*** is discussed in §15.4.2.

Using a class as a base is equivalent to declaring an (unnamed) object of that class. Consequently, a class must be defined in order to be used as a base (§5.7):

```
class Employee;        // declaration only, no definition

class Manager : public Employee {    // error: Employee not defined
    // ...
};
```

12.2.1 Member Functions [derived.member]

Simple data structures, such as ***Employee*** and ***Manager***, are really not that interesting and often not particularly useful. We need to give the information as a proper type that provides a suitable set of operations that present the concept, and we need to do this without tying us to the details of a particular representation. For example:

```
class Employee {
    string first_name, family_name;
    char middle_initial;
    // ...
public:
    void print() const;
    string full_name() const
        { return first_name + ´ ´ + middle_initial + ´ ´ + family_name; }
    // ...
};
```

```
class Manager : public Employee {
        // ...
public:
        void print( ) const;
        // ...
};
```

A member of a derived class can use the public – and protected (see §15.3) – members of its base class as if they were declared in the derived class itself. For example:

```
void Manager::print( ) const
{
        cout << "name is " << full_name( ) << '\n' ;
        // ...
}
```

However, a derived class cannot use a base class' private names:

```
void Manager::print( ) const
{
        cout << " name is " << family_name << '\n' ;     // error!
        // ...
}
```

This second version of ***Manager*::*print*( )** will not compile. A member of a derived class has no special permission to access private members of its base class, so ***family_name*** is not accessible to ***Manager*::*print*( )**.

This comes as a surprise to some, but consider the alternative: that a member function of a derived class could access the private members of its base class. The concept of a private member would be rendered meaningless by allowing a programmer to gain access to the private part of a class simply by deriving a new class from it. Furthermore, one could no longer find all uses of a private name by looking at the functions declared as members and friends of that class. One would have to examine every source file of the complete program for derived classes, then examine every function of those classes, then find every class derived from those classes, etc. This is, at best, tedious and often impractical. Where it is acceptable, ***protected*** – rather than ***private*** – members can be used. A protected member is like a public member to a member of a derived class, yet it is like a private member to other functions (see §15.3).

Typically, the cleanest solution is for the derived class to use only the public members of its base class. For example:

```
void Manager::print( ) const
{
        Employee::print( );   // print Employee information

        cout << level;        // print Manager-specific information
        // ...
}
```

Note that **::** must be used because ***print*( )** has been redefined in ***Manager***. Such reuse of names is typical. The unwary might write this:

```
void Manager::print() const
{
    print();    // oops!

    // print Manager-specific information
}
```

and find the program involved in an unexpected sequence of recursive calls.

12.2.2 Constructors and Destructors [derived.ctor]

Some derived classes need constructors. If a base class has constructors, then a constructor must be invoked. Default constructors can be invoked implicitly. However, if all constructors for a base require arguments, then a constructor for that base must be explicitly called. Consider:

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& n, int d);
    // ...
};

class Manager : public Employee {
    set<Employee*> group;    // people managed
    short level;
    // ...
public:
    Manager(const string& n, int d, int lvl);
    // ...
};
```

Arguments for the base class' constructor are specified in the definition of a derived class' constructor. In this respect, the base class acts exactly like a member of the derived class (§10.4.6). For example:

```
Employee::Employee(const string& n, int d)
    : family_name(n), department(d)        // initialize members
{
    // ...
}

Manager::Manager(const string& n, int d, int lvl)
    : Employee(n,d),          // initialize base
      level(lvl)              // initialize members
{
    // ...
}
```

A derived class constructor can specify initializers for its own members and immediate bases only; it cannot directly initialize members of a base. For example:

```
Manager::Manager(const string& n, int d, int lvl)
    : family_name(n),    // error: family_name not declared in manager
      department(d),     // error: department not declared in manager
      level(lvl)
{
    // ...
}
```

This definition contains three errors: it fails to invoke *Employee´s* constructor, and twice it attempts to initialize members of *Employee* directly.

Class objects are constructed from the bottom up: first the base, then the members, and then the derived class itself. They are destroyed in the opposite order: first the derived class itself, then the members, and then the base. Members and bases are constructed in order of declaration in the class and destroyed in the reverse order. See also §10.4.6 and §15.2.4.1.

12.2.3 Copying [derived.copy]

Copying of class objects is defined by the copy constructor and assignments (§10.4.4.1). Consider:

```
class Employee {
    // ...
    Employee& operator=(const Employee&);
    Employee(const Employee&);
};

void f(const Manager& m)
{
    Employee e = m;      // construct e from Employee part of m
    e = m;               // assign Employee part of m to e
}
```

Because the *Employee* copy functions do not know anything about *Manager*s, only the *Employee* part of a *Manager* is copied. This is commonly referred to as *slicing* and can be a source of surprises and errors. One reason to pass pointers and references to objects of classes in a hierarchy is to avoid slicing. Other reasons are to preserve polymorphic behavior (§2.5.4, §12.2.6) and to gain efficiency.

12.2.4 Class Hierarchies [derived.hierarchy]

A derived class can itself be a base class. For example:

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

Such a set of related classes is traditionally called a *class hierarchy*. Such a hierarchy is most often a tree, but it can also be a more general graph structure. For example:

```
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
```

```
class Tsec : public Temporary, public Secretary { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

Or graphically:

Thus, as is explained in detail in §15.2, C++ can express a directed acyclic graph of classes.

12.2.5 Type Fields [derived.typefield]

To use derived classes as more than a convenient shorthand in declarations, we must solve the following problem: Given a pointer of type *base**, to which derived type does the object pointed to really belong? There are four fundamental solutions to the problem:

[1] Ensure that only objects of a single type are pointed to (§2.7, Chapter 13).
[2] Place a type field in the base class for the functions to inspect.
[3] Use *dynamic_cast* (§15.4.2, §15.4.5).
[4] Use virtual functions (§2.5.5, §12.2.6).

Pointers to base classes are commonly used in the design of *container classes* such as set, vector, and list. In this case, solution 1 yields homogeneous lists, that is, lists of objects of the same type. Solutions 2, 3, and 4 can be used to build heterogeneous lists, that is, lists of (pointers to) objects of several different types. Solution 3 is a language-supported variant of solution 2. Solution 4 is a special type-safe variation of solution 2. Combinations of solutions 1 and 4 are particularly interesting and powerful; in almost all situations, they yield cleaner code than do solutions 2 and 3.

Let us first examine the simple type-field solution to see why it is most often best avoided. The manager/employee example could be redefined like this:

```
struct Employee {
    enum Empl_type { M, E };
    Empl_type type;

    Employee() : type(E) { }

    string first_name, family_name;
    char middle_initial;

    Date hiring_date;
    short department;
    // ...
};
```

```
struct Manager : public Employee {
    Manager() { type = M; }

    set<Employee*> group;    // people managed
    short level;
    // ...
};
```

Given this, we can now write a function that prints information about each ***Employee***:

```
void print_employee(const Employee* e)
{
    switch (e->type) {
    case Employee::E:
        cout << e->family_name << '\t' << e->department << '\n';
        // ...
        break;
    case Employee::M:
    {   cout << e->family_name << '\t' << e->department << '\n';
        // ...
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
        break;
    }
    }
}
```

and use it to print a list of ***Employee***s, like this:

```
void print_list(const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
        print_employee(*p);
}
```

This works fine, especially in a small program maintained by a single person. However, it has the fundamental weakness in that it depends on the programmer manipulating types in a way that cannot be checked by the compiler. This problem is usually made worse because functions such as ***print_employee()*** are organized to take advantage of the commonality of the classes involved. For example:

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::M) {
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
    }
}
```

Finding all such tests on the type field buried in a large function that handles many derived classes can be difficult. Even when they have been found, understanding what is going on can be difficult. Furthermore, any addition of a new kind of ***Employee*** involves a change to all the key functions in the system – the ones containing the tests on the type field. The programmer must consider every function that could conceivably need a test on the type field after a change. This implies the need to access critical source code and the resulting necessary overhead of testing the affected code. The use of an explicit type conversion is a strong hint that improvement is possible.

In other words, use of a type field is an error-prone technique that leads to maintenance problems. The problems increase in severity as the size of the program increases because the use of a type field causes a violation of the ideals of modularity and data hiding. Each function using a type field must know about the representation and other details of the implementation of every class derived from the one containing the type field.

It also seems that the existence of any common data accessible from every derived class, such as a type field, tempts people to add more such data. The common base thus becomes the repository of all kinds of ''useful information.'' This, in turn, gets the implementation of the base and derived classes intertwined in ways that are most undesirable. For clean design and simpler maintenance, we want to keep separate issues separate and avoid mutual dependencies.

12.2.6 Virtual Functions [derived.virtual]

Virtual functions overcome the problems with the type-field solution by allowing the programmer to declare functions in a base class that can be redefined in each derived class. The compiler and loader will guarantee the correct correspondence between objects and the functions applied to them. For example:

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& name, int dept);
    virtual void print() const;
    // ...
};
```

The keyword ***virtual*** indicates that ***print***( ) can act as an interface to the ***print***( ) function defined in this class and the ***print***( ) functions defined in classes derived from it. Where such ***print***( ) functions are defined in derived classes, the compiler ensures that the right ***print***( ) for the given ***Employee*** object is invoked in each case.

To allow a virtual function declaration to act as an interface to functions defined in derived classes, the argument types specified for a function in a derived class cannot differ from the argument types declared in the base, and only very slight changes are allowed for the return type (§15.6.2). A virtual member function is sometimes called a *method*.

A virtual function *must* be defined for the class in which it is first declared (unless it is declared to be a pure virtual function; see §12.3). For example:

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    // ...
}
```

A virtual function can be used even if no class is derived from its class, and a derived class that does not need its own version of a virtual function need not provide one. When deriving a class, simply provide an appropriate function, if it is needed. For example:

```
class Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    // ...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tlevel " << level << '\n';
    // ...
}
```

A function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to *override* the base class version of the virtual function. Except where we explicitly say which version of a virtual function is called (as in the call ***Employee*::*print*()**), the overriding function is chosen as the most appropriate for the object for which it is called.

The global function ***print_employee*()** (§12.2.5) is now unnecessary because the ***print*()** member functions have taken its place. A list of ***Employee***s can be printed like this:

```
void print_list(set<Employee*>& s)
{
    for (set<Employee*>::const_iterator p = s.begin(); p!=s.end(); ++p) // see §2.7.2
        (*p)->print();
}
```

or even

```
void print_list(set<Employee*>& s)
{
    for_each(s.begin(), s.end(), mem_fun(&Employee::print));    // see §3.8.5
}
```

Each ***Employee*** will be written out according to its type. For example:

```
int main()
{
    Employee e("Brown",1234);
    Manager m("Smith",1234,2);
    set<Employee*> empl;
    empl.push_front(&e);      // see §2.5.4
    empl.push_front(&m);
    print_list(empl);
}
```

produced:

```
Smith 1234
    level 2
Brown 1234
```

Note that this will work even if ***Employee*::*print_list*()** was written and compiled before the specific derived class ***Manager*** was even conceived of! This is a key aspect of classes. When used properly, it becomes the cornerstone of object-oriented designs and provides a degree of stability to an evolving program.

Getting ''the right'' behavior from ***Employee***'s functions independently of exactly what kind of ***Employee*** is actually used is called *polymorphism*. A type with virtual functions is called a *polymorphic type*. To get polymorphic behavior in C++, the member functions called must be ***virtual*** and objects must be manipulated through pointers or references. When manipulating an object directly (rather than through a pointer or reference), its exact type is known by the compilation so that run-time polymorphism is not needed.

Clearly, to implement polymorphism, the compiler must store some kind of type information in each object of class ***Employee*** and use it to call the right version of the virtual function ***print*()**. In a typical implementation, the space taken is just enough to hold a pointer (§2.5.5). This space is taken only in objects of a class with virtual functions – not in every object, or even in every object of a derived class. You pay this overhead only for classes for which you declare virtual functions. Had you chosen to use the alternative type-field solution, a comparable amount of space would have been needed for the type field.

Calling a function using the scope resolution operator **::** as is done in ***Manager*::*print*()** ensures that the virtual mechanism is not used. Otherwise, ***Manager*::*print*()** would suffer an infinite recursion. The use of a qualified name has another desirable effect. That is, if a ***virtual*** function is also ***inline*** (as is not uncommon), then inline substitution can be used for calls specified using **::**. This provides the programmer with an efficient way to handle some important special cases in which one virtual function calls another for the same object. The ***Manager*::*print*()** function is an example of this. Because the type of the object is determined in the call of ***Manager*::*print*()**, it need not be dynamically determined again for the resulting call of ***Employee*::*print*()**.

It is worth remembering that the traditional and obvious implementation of a virtual function call is simply an indirect function call (§2.5.5), so efficiency concerns should not deter anyone from using a virtual function where an ordinary function call would be acceptably efficient.

12.3 Abstract Classes [derived.abstract]

Many classes resemble class ***Employee*** in that they are useful both as themselves and also as bases for derived classes. For such classes, the techniques described in the previous section suffice. However, not all classes follow that pattern. Some classes, such as class ***Shape***, represent abstract concepts for which objects cannot exist. A ***Shape*** makes sense only as the base of some class derived from it. This can be seen from the fact that it is not possible to provide sensible definitions for its virtual functions:

```
class Shape {
public:
    virtual void rotate(int) { error("Shape::rotate"); } // inelegant
    virtual void draw() { error("Shape::draw"); }
    // ...
};
```

Trying to make a shape of this unspecified kind is silly but legal:

```
Shape s;  // silly: ''shapeless shape''
```

It is silly because every operation on ***s*** will result in an error.

A better alternative is to declare the virtual functions of class ***Shape*** to be *pure virtual functions*. A virtual function is ''made pure'' by the initializer **= 0**:

```
class Shape {          // abstract class
public:
    virtual void rotate(int) = 0;     // pure virtual function
    virtual void draw() = 0;          // pure virtual function
    virtual bool is_closed() = 0;     // pure virtual function
    // ...
};
```

A class with one or more pure virtual functions is an *abstract class*, and no objects of that abstract class can be created:

```
Shape s;  // error: variable of abstract class Shape
```

An abstract class can be used only as an interface and as a base for other classes. For example:

```
class Point { /* ... */ };

class Circle : public Shape {
public:
    void rotate(int) { }                  // override Shape::rotate
    void draw();                          // override Shape::draw
    bool is_closed() { return true; }     // override Shape::is_closed

    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

A pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is also an abstract class. This allows us to build implementations in stages:

```
class Polygon : public Shape {                // abstract class
public:
    bool is_closed() { return true; }         // override Shape::is_closed
    // ... draw and rotate not overridden ...
};

Polygon b;      // error: declaration of object of abstract class Polygon

class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    void draw();                              // override Shape::draw
    void rotate(int);                         // override Shape::rotate
    // ...
};

Irregular_polygon poly(some_points);          // fine (assume suitable constructor)
```

An important use of abstract classes is to provide an interface without exposing any implementation details. For example, an operating system might hide the details of its device drivers behind an abstract class:

```
class Character_device {
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int ...) = 0;
    virtual ~Character_device() { }           // virtual destructor
};
```

We can then specify drivers as classes derived from ***Character_device***, and manipulate a variety of drivers through that interface. The importance of virtual destructors is explained in §12.4.2.

With the introduction of abstract classes, we have the basic facilities for writing a complete program in a modular fashion using classes as building blocks.

12.4 Design of Class Hierarchies [derived.design]

Consider a simple design problem: provide a way for a program to get an integer value from a user interface. This can be done in a bewildering number of ways. To insulate our program from this variety, and also to get a chance to explore the possible design choices, let us start by defining our program's model of this simple input operation. We will leave until later the details of implementing it using a real user-interface system.

The idea is to have a class ***Ival_box*** that knows what range of input values it will accept. A program can ask an ***Ival_box*** for its value and ask it to prompt the user if necessary. In addition, a program can ask an ***Ival_box*** if a user changed the value since the program last looked at it.

Because there are many ways of implementing this basic idea, we must assume that there will be many different kinds of ***Ival_boxes***, such as sliders, plain boxes in which a user can type a number, dials, and voice interaction.

The general approach is to build a ''virtual user-interface system'' for the application to use. This system provides some of the services provided by existing user-interface systems. It can be implemented on a wide variety of systems to ensure the portability of application code. Naturally, there are other ways of insulating an application from a user-interface system. I chose this approach because it is general, because it allows me to demonstrate a variety of techniques and design tradeoffs, because those techniques are also the ones used to build ''real'' user-interface systems, and – most important – because these techniques are applicable to problems far beyond the narrow domain of interface systems.

12.4.1 A Traditional Class Hierarchy [derived.traditional]

Our first solution is a traditional class hierarchy as is commonly found in Simula, Smalltalk, and older C++ programs.

Class ***Ival_box*** defines the basic interface to all ***Ival_box***es and specifies a default implementation that more specific kinds of ***Ival_box***es can override with their own versions. In addition, we declare the data needed to implement the basic notion:

```
class Ival_box {
protected:
        int val;
        int low, high;
        bool changed;
public:
        Ival_box(int ll, int hh) { changed = false; val = low = ll; high = hh; }

        virtual int get_value() { changed = false; return val; }
        virtual void set_value(int i) { changed = true; val = i; }          // for user
        virtual void reset_value(int i) { changed = false; val = i; }       // for application
        virtual void prompt() { }
        virtual bool was_changed() const { return changed; }
};
```

The default implementation of the functions is pretty sloppy and is provided here primarily to illustrate the intended semantics. A realistic class would, for example, provide some range checking.

A programmer might use these ''***ival*** classes'' like this:

```
void interact(Ival_box* pb)
{
        pb->prompt(); // alert user
        // ...
        int i = pb->get_value();
        if (pb->was_changed()) {
                // new value; do something
        }
```

```
        else {
                // old value was fine; do something else
        }
        // ...
}

void some_fct()
{
        Ival_box* p1 = new Ival_slider(0,5);        // Ival_slider derived from Ival_box
        interact(p1);

        Ival_box* p2 = new Ival_dial(1,12);
        interact(p2);
}
```

Most application code is written in terms of (pointers to) plain ***Ival_box***es the way ***interact***( ) is. That way, the application doesn't have to know about the potentially large number of variants of the ***Ival_box*** concept. The knowledge of such specialized classes is isolated in the relatively few functions that create such objects. This isolates users from changes in the implementations of the derived classes. Most code can be oblivious to the fact that there are different kinds of ***Ival_box***es.

To simplify the discussion, I do not address issues of how a program waits for input. Maybe the program really does wait for the user in ***get_value***( ), maybe the program associates the ***Ival_box*** with an event and prepares to respond to a callback, or maybe the program spawns a thread for the ***Ival_box*** and later inquires about the state of that thread. Such decisions are crucial in the design of user-interface systems. However, discussing them here in any realistic detail would simply distract from the presentation of programming techniques and language facilities. The design techniques described here and the language facilities that support them are not specific to user interfaces. They apply to a far greater range of problems.

The different kinds of ***Ival_box***es are defined as classes derived from ***Ival_box***. For example:

```
class Ival_slider : public Ival_box {
        // graphics stuff to define what the slider looks like, etc.
public:
        Ival_slider(int, int);

        int get_value();
        void prompt();
};
```

The data members of ***Ival_box*** were declared ***protected*** to allow access from derived classes. Thus, ***Ival_slider***::***get_value***( ) can deposit a value in ***Ival_box***::***val***. A ***protected*** member is accessible from a class' own members and from members of derived classes, but not to general users (see §15.3).

In addition to ***Ival_slider***, we would define other variants of the ***Ival_box*** concept. These could include ***Ival_dial***, which lets you select a value by turning a knob; ***flashing_ival_slider***, which flashes when you ask it to ***prompt***( ); and ***popup_ival_slider***, which responds to ***prompt***( ) by appearing in some prominent place, thus making it hard for the user to ignore.

From where would we get the graphics stuff? Most user-interface systems provide a class defining the basic properties of being an entity on the screen. So, if we use the system from ''Big

Bucks Inc.,'' we would have to make each of our ***Ival_slider***, ***Ival_dial***, etc., classes a kind of ***BBwindow***. This would most simply be achieved by rewriting our ***Ival_box*** so that it derives from ***BBwindow***. In that way, all our classes inherit all the properties of a ***BBwindow***. For example, every ***Ival_box*** can be placed on the screen, obey the graphical style rules, be resized, be dragged around, etc., according to the standard set by the ***BBwindow*** system. Our class hierarchy would look like this:

```
class Ival_box : public BBwindow { /* ... */ }; // rewritten to use BBwindow
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

or graphically:

12.4.1.1 Critique [derived.critique]

This design works well in many ways, and for many problems this kind of hierarchy is a good solution. However, there are some awkward details that could lead us to look for alternative designs.

We retrofitted ***BBwindow*** as the base of ***Ival_box***. This is not quite right. The use of ***BBwindow*** isn't part of our basic notion of an ***Ival_box***; it is an implementation detail. Deriving ***Ival_box*** from ***BBwindow*** elevated an implementation detail to a first-level design decision. That can be right. For example, using the environment defined by ''Big Bucks Inc.'' may be a key decision of how our organization conducts its business. However, what if we also wanted to have implementations of our ***Ival_box***es for systems from ''Imperial Bananas,'' ''Liberated Software,'' and ''Compiler Whizzes?'' We would have to maintain four distinct versions of our program:

```
class Ival_box : public BBwindow { /* ... */ };      // BB version
class Ival_box : public CWwindow { /* ... */ };      // CW version
class Ival_box : public IBwindow { /* ... */ };      // IB version
class Ival_box : public LSwindow { /* ... */ };      // LS version
```

Having many versions could result in a version-control nightmare.

Another problem is that every derived class shares the basic data declared in ***Ival_box***. That data is, of course, an implementation detail that also crept into our ***Ival_box*** interface. From a practical point of view, it is also the wrong data in many cases. For example, an ***Ival_slider*** doesn't need the value stored specifically. It can easily be calculated from the position of the slider when someone executes ***get_value***( ). In general, keeping two related, but different, sets of data is

asking for trouble. Sooner or later someone will get them out of sync. Also, experience shows that novice programmers tend to mess with protected data in ways that are unnecessary and that cause maintenance problems. Data is better kept private so that writers of derived classes cannot mess with them. Better still, data should be in the derived classes, where it can be defined to match requirements exactly and cannot complicate the life of unrelated derived classes. In almost all cases, a protected interface should contain only functions, types, and constants.

Deriving from ***BBwindow*** gives the benefit of making the facilities provided by ***BBwindow*** available to users of ***Ival_box***. Unfortunately, it also means that changes to class ***BBwindow*** may force users to recompile or even rewrite their code to recover from such changes. In particular, the way most C++ implementations work implies that a change in the size of a base class requires a recompilation of all derived classes.

Finally, our program may have to run in a mixed environment in which windows of different user-interface systems coexist. This could happen either because two systems somehow share a screen or because our program needs to communicate with users on different systems. Having our user-interface systems ''wired in'' as the one and only base of our one and only ***Ival_box*** interface just isn't flexible enough to handle those situations.

12.4.2 Abstract Classes [derived.interface]

So, let's start again and build a new class hierarchy that solves the problems presented in the critique of the traditional hierarchy:

[1] The user-interface system should be an implementation detail that is hidden from users who don't want to know about it.

[2] The ***Ival_box*** class should contain no data.

[3] No recompilation of code using the ***Ival_box*** family of classes should be required after a change of the user-interface system.

[4] ***Ival_box***es for different interface systems should be able to coexist in our program.

Several alternative approaches can be taken to achieve this. Here, I present one that maps cleanly into the C++ language.

First, I specify class ***Ival_box*** as a pure interface:

```
class Ival_box {
public:
        virtual int get_value() = 0;
        virtual void set_value(int i) = 0;
        virtual void reset_value(int i) = 0;
        virtual void prompt() = 0;
        virtual bool was_changed() const = 0;
        virtual ~Ival_box() { }
};
```

This is much cleaner than the original declaration of ***Ival_box***. The data is gone and so are the simplistic implementations of the member functions. Gone, too, is the constructor, since there is no data for it to initialize. Instead, I added a virtual destructor to ensure proper cleanup of the data that will be defined in the derived classes.

The definition of ***Ival_slider*** might look like this:

```
class Ival_slider : public Ival_box, protected BBwindow {
public:
    Ival_slider(int, int);
    ~Ival_slider();

    int get_value();
    void set_value(int i);
    // ...
protected:
    // functions overriding BBwindow virtual functions
    // e.g. BBwindow::draw(), BBwindow::mouse1hit()
private:
    // data needed for slider
};
```

The derived class ***Ival_slider*** inherits from an abstract class (***Ival_box***) that requires it to implement the base class' pure virtual functions. It also inherits from ***BBwindow*** that provides it with the means of doing so. Since ***Ival_box*** provides the interface for the derived class, it is derived using ***public***. Since ***BBwindow*** is only an implementation aid, it is derived using ***protected*** (§15.3.2). This implies that a programmer using ***Ival_slider*** cannot directly use facilities defined by ***BBwindow***. The interface provided by ***Ival_slider*** is the one inherited by ***Ival_box***, plus what ***Ival_slider*** explicitly declares. I used ***protected*** derivation instead of the more restrictive (and usually safer) ***private*** derivation to make ***BBwindow*** available to classes derived from ***Ival_slider***.

Deriving directly from more than one class is usually called *multiple inheritance* (§15.2). Note that ***Ival_slider*** must override functions from both ***Ival_box*** and ***BBwindow***. Therefore, it must be derived directly or indirectly from both. As shown in §12.4.1.1, deriving ***Ival_slider*** indirectly from ***BBwindow*** by making ***BBwindow*** a base of ***Ival_box*** is possible, but doing so has undesirable side effects. Similarly, making the ''implementation class'' ***BBwindow*** a member of ***Ival_box*** is not a solution because a class cannot override virtual functions of its members (§24.3.4). Representing the window by a ***BBwindow**** member in ***Ival_box*** leads to a completely different design with a separate set of tradeoffs (§12.7[14], §25.7).

Interestingly, this declaration of ***Ival_slider*** allows application code to be written exactly as before. All we have done is to restructure the implementation details in a more logical way.

Many classes require some form of cleanup for an object before it goes away. Since the abstract class ***Ival_box*** cannot know if a derived class requires such cleanup, it must assume that it does require some. We ensure proper cleanup by defining a virtual destructor ***Ival_box::~Ival_box()*** in the base and overriding it suitably in derived classes. For example:

```
void f(Ival_box* p)
{
    // ...
    delete p;
}
```

The ***delete*** operator explicitly destroys the object pointed to by ***p***. We have no way of knowing exactly to which class the object pointed to by ***p*** belongs, but thanks to ***Ival_box***'s virtual destructor, proper cleanup as (optionally) defined by that class' destructor will be called.

The ***Ival_box*** hierarchy can now be defined like this:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class Ival_dial : public Ival_box, protected BBwindow { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

or graphically using obvious abbreviations:

I used a dashed line to represent protected inheritance. As far as general users are concerned, doing that is simply an implementation detail.

12.4.3 Alternative Implementations [derived.alt]

This design is cleaner and more easily maintainable than the traditional one – and no less efficient. However, it still fails to solve the version control problem:

```
class Ival_box { /* ... */ };      // common
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ }; // for BB
class Ival_slider : public Ival_box, protected CWwindow { /* ... */ }; // for CW
// ...
```

In addition, there is no way of having an ***Ival_slider*** for ***BBwindow***s coexist with an ***Ival_slider*** for ***CWwindow***s, even if the two user-interface systems could themselves coexist.

The obvious solution is to define several different ***Ival_slider*** classes with separate names:

```
class Ival_box { /* ... */ };
class BB_ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_box, protected CWwindow { /* ... */ };
// ...
```

or graphically:

To further insulate our application-oriented ***Ival_box*** classes from implementation details, we can derive an abstract ***Ival_slider*** class from ***Ival_box*** and then derive the system-specific ***Ival_sliders*** from that:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
```

```
class BB_ival_slider : public Ival_slider, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWwindow { /* ... */ };
// ...
```

or graphically:

Usually, we can do better yet by utilizing more-specific classes in the implementation hierarchy. For example, if the ‘‘Big Bucks Inc.’’ system has a slider class, we can derive our ***Ival_slider*** directly from the ***BBslider***:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
```

or graphically:

This improvement becomes significant where – as is not uncommon – our abstractions are not too different from the ones provided by the system used for implementation. In that case, programming is reduced to mapping between similar concepts. Derivation from general base classes, such as ***BBwindow***, is then done only rarely.

The complete hierarchy will consist of our original application-oriented conceptual hierarchy of interfaces expressed as derived classes:

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

followed by the implementations of this hierarchy for various graphical user-interface systems, expressed as derived classes:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class BB_flashing_ival_slider : public Flashing_ival_slider,
        protected BBwindow_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

Using obvious abbreviations, this hierarchy can be represented graphically like this:

The original ***Ival_box*** class hierarchy appears unchanged surrounded by implementation classes.

12.4.3.1 Critique [derived.critique2]

The abstract class design is flexible and almost as simple to deal with as the equivalent design that relies on a common base defining the user-interface system. In the latter design, the windows class is the root of a tree. In the former, the original application class hierarchy appears unchanged as the root of classes that supply its implementations. From the application's point of view, these designs are equivalent in the strong sense that almost all code works unchanged and in the same way in the two cases. In either case, you can look at the ***Ival_box*** family of classes without bothering with the window-related implementation details most of the time. For example, we would not need to rewrite ***interact*( )** from §12.4.1 if we switched from the one class hierarchy to the other.

In either case, the implementation of each ***Ival_box*** class must be rewritten when the public interface of the user-interface system changes. However, in the abstract class design, almost all user code is protected against changes to the implementation hierarchy and requires no recompilation after such a change. This is especially important when the supplier of the implementation hierarchy issues a new ''almost compatible'' release. In addition, users of the abstract class hierarchy are in less danger of being locked into a proprietary implementation than are users of a classical hierarchy. Users of the ***Ival_box*** abstract class application hierarchy cannot accidentally use facilities from the implementation because only facilities explicitly specified in the ***Ival_box*** hierarchy are accessible; nothing is implicitly inherited from an implementation-specific base class.

12.4.4 Localizing Object Creation [derived.local]

Most of an application can be written using the ***Ival_box*** interface. Further, should the derived interfaces evolve to provide more facilities than plain ***Ival_box***, then most of an application can be written using the ***Ival_box***, ***Ival_slider***, etc., interfaces. However, the creation of objects must be

done using implementation-specific names such as ***CW_ival_dial*** and ***BB_flashing_ival_slider***. We would like to minimize the number of places where such specific names occur, and object creation is hard to localize unless it is done systematically.

As usual, the solution is to introduce an indirection. This can be done in many ways. A simple one is to introduce an abstract class to represent the set of creation operations:

```
class Ival_maker {
public:
    virtual Ival_dial* dial(int, int) =0;                    // make dial
    virtual Popup_ival_slider* popup_slider(int, int) =0; // make popup slider
    // ...
};
```

For each interface from the ***Ival_box*** family of classes that a user should know about, class ***Ival_maker*** provides a function that makes an object. Such a class is sometimes called a *factory*, and its functions are (somewhat misleadingly) sometimes called *virtual constructors* (§15.6.2).

We now represent each user-interface system by a class derived from ***Ival_maker***:

```
class BB_maker : public Ival_maker {          // make BB versions
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};

class LS_maker : public Ival_maker {          // make LS versions
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};
```

Each function creates an object of the desired interface and implementation type. For example:

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a,b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a,b);
}
```

Given a pointer to a ***Ival_maker***, a user can now create objects without having to know exactly which user-interface system is used. For example:

```
void user(Ival_maker* pim)
{
    Ival_box* pb = pim->dial(0,99);     // create appropriate dial
    // ...
}
```

```
BB_maker BB_impl; // for BB users
LS_maker LS_impl; // for LS users

void driver()
{
    user(&BB_impl);     // use BB
    user(&LS_impl);     // use LS
}
```

12.5 Class Hierarchies and Abstract Classes [derived.hier]

An abstract class is an interface. A class hierarchy is a means of building classes incrementally. Naturally, every class provides an interface to users and some abstract classes provide significant functionality to build from, but ''interface'' and ''building block'' are the primary roles of abstract classes and class hierarchies.

A classical hierarchy is a hierarchy in which the individual classes both provide useful functionality for users and act as building blocks for the implementation of more advanced or specialized classes. Such hierarchies are ideal for supporting programming by incremental refinement. They provide the maximum support for the implementation of new classes as long as the new class relates strongly to the existing hierarchy.

Classical hierarchies do tend to couple implementation concerns rather strongly with the interfaces provided to users. Abstract classes can help here. Hierarchies of abstract classes provide a clean and powerful way of expressing concepts without encumbering them with implementation concerns or significant run-time overheads. After all, a virtual function call is cheap and independent of the kind of abstraction barrier it crosses. It costs no more to call a member of an abstract class than to call any other ***virtual*** function.

The logical conclusion of this line of thought is a system represented to users as a hierarchy of abstract classes and implemented by a classical hierarchy.

12.6 Advice [derived.advice]

[1] Avoid type fields; §12.2.5.
[2] Use pointers and references to avoid slicing; §12.2.3.
[3] Use abstract classes to focus design on the provision of clean interfaces; §12.3.
[4] Use abstract classes to minimize interfaces; §12.4.2.
[5] Use abstract classes to keep implementation details out of interfaces; §12.4.2.
[6] Use virtual functions to allow new implementations to be added without affecting user code; §12.4.1.
[7] Use abstract classes to minimize recompilation of user code; §12.4.2.
[8] Use abstract classes to allow alternative implementations to coexist; §12.4.3.
[9] A class with a virtual function should have a virtual destructor; §12.4.2.
[10] An abstract class typically doesn't need a constructor; §12.4.2.
[11] Keep the representations of distinct concepts distinct; §12.4.1.1.

12.7 Exercises [derived.exercises]

1. (*1) Define

```
class base {
public:
    virtual void iam() { cout << "base\n"; }
};
```

   Derive two classes from *base*, and for each define *iam*( ) to write out the name of the class. Create objects of these classes and call *iam*( ) for them. Assign pointers to objects of the derived classes to *base** pointers and call *iam*( ) through those pointers.

2. (*3.5) Implement a simple graphics system using whatever graphics facilities are available on your system (if you don't have a good graphics system or have no experience with one, you might consider a simple ''huge bit ASCII implementation'' where a point is a character position and you write by placing a suitable character, such as * in a position): *Window*(*n*,*m*) creates an area of size *n* times *m* on the screen. Points on the screen are addressed using (x,y) coordinates (Cartesian). A *Window w* has a current position *w*.*current*( ). Initially, *current* is *Point*(*0*,*0*). The current position can be set by *w*.*current*(*p*) where *p* is a *Point*. A *Point* is specified by a coordinate pair: *Point*(*x*,*y*). A *Line* is specified by a pair of *Point*s: *Line*(*w*.*current*( ),*p2*); class *Shape* is the common interface to *Dot*s, *Line*s, *Rectangle*s, *Circle*s, etc. A *Point* is not a *Shape*. A *Dot*, *Dot*(*p*) can be used to represent a *Point p* on the screen. A *Shape* is invisible unless *draw*( )n. For example: *w*.*draw*(*Circle*(*w*.*current*( ),*10*)). Every *Shape* has 9 contact points: *e* (east), *w* (west), *n* (north), *s* (south), *ne*, *nw*, *se*, *sw*, and *c* (center). For example, *Line*(*x*.*c*( ),*y*.*nw*( )) creates a line from *x*'s center to *y*'s top left corner. After *draw*( )ing a *Shape* the current position is the *Shape*'s *se*( ). A *Rectangle* is specified by its bottom left and top right corner: *Rectangle*(*w*.*current*( ),*Point*(*10*,*10*)). As a simple test, display a simple ''child's drawing of a house'' with a roof, two windows, and a door.

3. (*2) Important aspects of a *Shape* appear on the screen as a set of line segments. Implement operations to vary the appearance of these segments: *s*.*thickness*(*n*) sets the line thickness to *0*, *1*, *2*, or *3*, where *2* is the default and *0* means invisible. In addition, a line segment can be *solid*, *dashed*, or *dotted*. This is set by the function *Shape*::*outline*( ).

4. (*2.5) Provide a function *Line*::*arrowhead*( ) that adds arrow heads to an end of a line. A line has two ends and an arrowhead can point in two directions relative to the line, so the argument or arguments to *arrowhead*( ) must be able to express at least four alternatives.

5. (*3.5) Make sure that points and line segments that fall outside the *Window* do not appear on the screen. This is often called ''clipping.'' As an exercise only, do not rely on the implementation graphics system for this.

6. (*2.5) Add a *Text* type to the graphics system. A *Text* is a rectangular *Shape* displaying characters. By default, a character takes up one coordinate unit along each coordinate axis.

7. (*2) Define a function that draws a line connecting two shapes by finding the two closest ''contact points'' and connecting them.

8. (*3) Add a notion of color to the simple graphics system. Three things can be colored: the background, the inside of a closed shape, and the outlines of shapes.

9. (*2) Consider:

```
class Char_vec {
        int sz;
        char element[1];
public:
        static Char_vec* new_char_vec(int s);
        char& operator[](int i) { return element[i]; }
        // ...
};
```

Define ***new_char_vec*()** to allocate contiguous memory for a ***Char_vec*** object so that the elements can be indexed through ***element*** as shown. Under what circumstances does this trick cause serious problems?

10. (*2.5) Given classes ***Circle***, ***Square***, and ***Triangle*** derived from a class ***Shape***, define a function ***intersect*()** that takes two ***Shape**** arguments and calls suitable functions to determine if the two shapes overlap. It will be necessary to add suitable (virtual) functions to the classes to achieve this. Don't bother to write the code that checks for overlap; just make sure the right functions are called. This is commonly referred to as *double* ***dispatch*** or a *multi-method*.
11. (*5) Design and implement a library for writing event-driven simulations. Hint: ***<task.h>***. However, that is an old program, and you can do better. There should be a class ***task***. An object of class ***task*** should be able to save its state and to have that state restored (you might define ***task::save*()** and ***task::restore*()**) so that it can operate as a coroutine. Specific tasks can be defined as objects of classes derived from class ***task***. The program to be executed by a task might be specified as a virtual function. It should be possible to pass arguments to a new task as arguments to its constructor(s). There should be a scheduler implementing a concept of virtual time. Provide a function ***task::delay*(*long*)** that ''consumes'' virtual time. Whether the scheduler is part of class ***task*** or separate will be one of the major design decisions. The tasks will need to communicate. Design a class ***queue*** for that. Devise a way for a task to wait for input from several queues. Handle run-time errors in a uniform way. How would you debug programs written using such a library?
12. (*2) Define interfaces for ***Warrior***, ***Monster***, and ***Object*** (that is a thing you can pick up, drop, use, etc.) classes for an adventure-style game.
13. (*1.5) Why is there both a ***Point*** and a ***Dot*** class in §12.7[2]? Under which circumstances would it be a good idea to augment the ***Shape*** classes with concrete versions of key classes such as ***Line***.
14. (*3) Outline a different implementation strategy for the ***Ival_box*** example (§12.4) based on the idea that every class seen by an application is an interface containing a single pointer to the implementation. Thus, each "interface class" will be a handle to an "implementation class," and there will be an interface hierarchy and an implementation hierarchy. Write code fragments that are detailed enough to illustrate possible problems with type conversion. Consider ease of use, ease of programming, ease of reusing implementations and interfaces when adding a new concept to the hierarchy, ease of making changes to interfaces and implementations, and need for recompilation after change in the implementation.

13

Templates

Your quote here.
– B. Stroustrup

Templates — a string template — instantiation — template parameters — type checking — function templates — template argument deduction — specifying template arguments — function template overloading — policy as template arguments — default template arguments — specialization — derivation and templates — member templates — conversions — source code organization — advice — exercises.

13.1 Introduction [temp.intro]

Independent concepts should be independently represented and should be combined only when needed. Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies. Either way, you get a less flexible set of components out of which to compose systems. Templates provide a simple way to represent a wide range of general concepts and simple ways to combine them. The resulting classes and functions can match hand-written, more-specialized code in run-time and space efficiency.

Templates provide direct support for generic programming (§2.7), that is, programming using types as parameters. The C++ template mechanism allows a type to be a parameter in the definition of a class or a function. A template depends only on the properties that it actually uses from its parameter types and does not require different types used as arguments to be explicitly related. In particular, the argument types used for a template need not be from a single inheritance hierarchy.

Here, templates are introduced with the primary focus on techniques needed for the design, implementation, and use of the standard library. The standard library requires a greater degree of generality, flexibility, and efficiency than does most software. Consequently, techniques that can be used in the design and implementation of the standard library are effective and efficient in the design of solutions to a wide variety of problems. These techniques enable an implementer to hide

sophisticated implementations behind simple interfaces and to expose complexity to the user only when the user has a specific need for it. For example, ***sort***(***v***) can be the interface to a variety of sort algorithms for elements of a variety of types held in a variety of containers. The sort function that is most appropriate for the particular ***v*** will be automatically chosen.

Every major standard library abstraction is represented as a template (for example, ***string***, ***ostream***, ***complex***, ***list***, and ***map***) and so are the key operations (for example, ***string*** compare, the output operator <<, ***complex*** addition, getting the next element from a ***list***, and ***sort***( )). This makes the library chapters (Part 3) of this book a rich source of examples of templates and programming techniques relying on them. Consequently, this chapter concentrates on smaller examples illustrating technical aspects of templates and fundamental techniques for using them:

§13.2: The basic mechanisms for defining and using class templates
§13.3: Function templates, function overloading, and type deduction
§13.4: Template parameters used to specify policies for generic algorithms
§13.5: Multiple definitions providing alternative implementations for a template
§13.6: Derivation and templates (run-time and compile-time polymorphism)
§13.7: Source code organization

Templates were introduced in §2.7.1 and §3.8. Detailed rules for template name resolution, template syntax, etc., can be found in §C.13.

13.2 A Simple String Template [temp.string]

Consider a string of characters. A string is a class that holds characters and provides operations such as subscripting, concatenation, and comparison that we usually associate with the notion of a ‘‘string.’’ We would like to provide that behavior for many different kinds of characters. For example, strings of signed characters, of unsigned characters, of Chinese characters, of Greek characters, etc., are useful in various contexts. Thus, we want to represent the notion of ‘‘string’’ with minimal dependence on a specific kind of character. The definition of a string relies on the fact that a character can be copied, and little else. Thus, we can make a more general string type by taking the string of ***char*** from §11.12 and making the character type a parameter:

```
template<class C> class String {
    struct Srep;
    Srep *rep;
public:
    String();
    String(const C*);
    String(const String&);

    C read(int i) const;
    // ...
};
```

The ***template <class C>*** prefix specifies that a template is being declared and that a type argument ***C*** will be used in the declaration. After its introduction, ***C*** is used exactly like other type names. The scope of ***C*** extends to the end of the declaration prefixed by ***template <class C>***. Note that ***template<class C>*** says that ***C*** is a *type* name; it need not be the name of a *class*.

The name of a class template followed by a type bracketed by < > is the name of a class (as defined by the template) and can be used exactly like other class names. For example:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
        // Japanese character
};

String<Jchar> js;
```

Except for the special syntax of its name, ***String<char>*** works exactly as if it had been defined using the definition of class ***String*** in §11.12. Making ***String*** a template allows us to provide the facilities we had for ***String*** of ***char*** for ***String***s of any kind of character. For example, if we use the standard library ***map*** and the ***String*** template, the word-counting example from §11.8 becomes:

```
int main()        // count the occurrences of each word on input
{
        String<char> buf;
        map<String<char>, int> m;
        while (cin>>buf) m[buf]++;
        // write out result
}
```

The version for our Japanese-character type ***Jchar*** would be:

```
int main()        // count the occurrences of each word on input
{
        String<Jchar> buf;
        map<String<Jchar>, int> m;
        while (cin>>buf) m[buf]++;
        // write out result
}
```

The standard library provides the template class ***basic_string*** that is similar to the templatized ***String*** (§11.12, §20.3). In the standard library, ***string*** is defined as a synonym for ***basic_string<char>***:

```
typedef basic_string<char> string;
```

This allows us to write the word-counting program like this:

```
int main()        // count the occurrences of each word on input
{
        string buf;
        map<string, int> m;
        while (cin>>buf) m[buf]++;
        // write out result
}
```

In general, ***typedef***s are useful for shortening the long names of classes generated from templates.

Also, we often prefer not to know the details of how a type is defined, and a ***typedef*** allows us to hide the fact that a type is generated from a template.

13.2.1 Defining a Template [temp.string.details]

A class generated from a class template is a perfectly ordinary class. Thus, use of a template does not imply any run-time mechanisms beyond what is used for an equivalent ''hand-written'' class. Nor does it necessarily imply any reduction in the amount of code generated.

It is usually a good idea to debug a particular class, such as ***String***, before turning it into a template such as ***String<C>***. By doing so, we handle many design problems and most of the code errors in the context of a concrete example. This kind of debugging is familiar to all programmers, and most people cope better with a concrete example than with an abstract concept. Later, we can deal with any problems that might arise from generalization without being distracted by more conventional errors. Similarly, when trying to understand a template, it is often useful to imagine its behavior for a particular type argument such as ***char*** before trying to comprehend the template in its full generality.

Members of a template class are declared and defined exactly as they would have been for a non-template class. A template member need not be defined within the template class itself. In that case, its definition must be provided somewhere else, as for non-template class members (§C.13.7). Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template. For example:

```
template<class C> struct String<C>::Srep {
    C* s;           // pointer to elements
    int sz;         // number of elements
    int n;          // reference count
    // ...
};

template<class C> C String<C>::read(int i) const { return rep->s[i]; }

template<class C> String<C>::String()
{
    p = new Srep(0, C());
}
```

Template parameters, such as ***C***, are parameters rather than names of types defined externally to the template. However, that doesn't affect the way we write the template code using them. Within the scope of ***String<C>***, qualification with ***<C>*** is redundant for the name of the template itself, so ***String<C>::String*** is the name for the constructor. If you prefer, you can be explicit:

```
template<class C> String<C>::String<C>()
{
    p = new Srep(0, C());
}
```

Just as there can be only one function defining a class member function in a program, there can be only one function template defining a class template member function in a program. However,

overloading is a possibility for functions only (§13.3.2), while specialization (§13.5) enables us to provide alternative implementations for a template.

It is not possible to overload a class template name, so if a class template is declared in a scope, no other entity can be declared there with the same name (see also §13.5). For example:

```
template<class T> class String { /* ... */ };

class String { /* ... */ }; // error: double definition
```

A type used as a template argument must provide the interface expected by the template. For example, a type used as an argument to ***String*** must provide the usual copy operations (§10.4.4.1, §20.2.1). Note that there is no requirement that different arguments for the same template parameter should be related by inheritance.

13.2.2 Template Instantiation [temp.string.inst]

The process of generating a class declaration from a template class and a template argument is often called *template instantiation* (§C.13.7). Similarly, a function is generated (''instantiated'') from a template function plus a template argument. A version of a template for a particular template argument is called a *specialization*.

In general, it is the implementation's job – *not* the programmer's – to ensure that versions of a template function are generated for each set of template arguments used (§C.13.7). For example:

```
String<char> cs;

void f()
{
    String<Jchar> js;

    cs = "It´s the implementation´s job to figure out what code needs to be generated";
}
```

For this, the implementation generates declarations for ***String<char>*** and ***String<Jchar>***, for their corresponding ***Srep*** types, for their destructors and default constructors, and for the assignment ***String<char>::operator=(char*)***. Other member functions are not used and should not be generated. The generated classes are perfectly ordinary classes that obey all the usual rules for classes. Similarly, generated functions are ordinary functions that obey all the usual rules for functions.

Obviously, templates provide a powerful way of generating code from relatively short definitions. Consequently, a certain amount of caution is in order to avoid flooding memory with almost identical function definitions (§13.5).

13.2.3 Template Parameters [temp.param]

A template can take type parameters, parameters of ordinary types such as ***int***s, and template parameters (§C.13.3). Naturally, a template can take several parameters. For example:

```
template<class T, T def_val> class Cont { /* ... */ };
```

As shown, a template parameter can be used in the definition of subsequent template parameters.

Integer arguments come in handy for supplying sizes and limits. For example:

```
template<class T, int i> class Buffer {
    T v[i];
    int sz;
public:
    Buffer() : sz(i) {}
    // ...
};

Buffer<char,127> cbuf;
Buffer<Record,8> rbuf;
```

Simple and constrained containers such as ***Buffer*** can be important where run-time efficiency and compactness are paramount (thus preventing the use of a more general ***string*** or ***vector***). Passing a size as a template argument allows ***Buffer***'s implementer to avoid free store use. Another example is the ***Range*** type in §25.6.1.

A template argument can be a constant expression (§C.5), the address of an object or function with external linkage (§9.2), or a non-overloaded pointer to member (§15.5). A pointer used as a template argument must be of the form ***&of***, where ***of*** is the name of an object or a function, or of the form ***f***, where ***f*** is the name of a function. A pointer to member must be of the form ***&X::of***, where ***of*** is the name of an member. In particular, a string literal is *not* acceptable as a template argument.

An integer template argument must be a constant:

```
void f(int i)
{
    Buffer<int,i> bx;    // error: constant expression expected
}
```

Conversely, a non-type template parameter is a constant within the template so that an attempt to change the value of a parameter is an error.

13.2.4 Type Equivalence [temp.equiv]

Given a template, we can generate types by supplying template arguments. For example:

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;

typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;

Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

When using the same set of template arguments for a template, we always refer to the same generated type. However, what does ''the same'' mean in this context? As usual, ***typedef***s do not introduce new types, so ***String<Uchar>*** is the same type as ***String<unsigned char>***. Conversely,

because *char* and *unsigned char* are different types (§4.3), *String<char>* and *String<unsigned char>* are different types.

The compiler can evaluate constant expressions (§C.5), so *Buffer<char,20-10>* is recognized to be the same type as *Buffer<char,10>*.

13.2.5 Type Checking [temp.check]

A template is defined and then later used in combination with a set of template arguments. When the template is defined, the definition is checked for syntax errors and possibly also for other errors that can be detected in isolation from a particular set of template arguments. For example:

```
template<class T> class List {
    struct Link {
        Link* pre;
        Link* suc;
        T val;
        Link(Link* p, Link* s,const T& v) :pre(p), suc(s), val(v) { }
    }           // syntax error: missing semicolon
    Link* head;
public:
    List() : head(7) { }        // error: pointer initialized with int
    List(const T& t) : head(new Link(0,o,t)) { }        // error: undefined identifier 'o'
    // ...
    void print_all() { for (Link* p = head; p; p=p->suc) cout << p->val << '\n'; }
};
```

A compiler can catch simple semantic errors at the point of definition or later at the point of use. Users generally prefer early detection, but not all ''simple'' errors are easy to detect. Here, I made three ''mistakes.'' Independently of what the template parameter is, a pointer *T** cannot be initialized by the integer *7*. Similarly, the identifier *o* (a mistyped *0*, of course) cannot be an argument to *List<T>::Link*'s constructor because there is no such name in scope.

A name used in a template definition must either be in scope or in some reasonably obvious way depend on a template parameter (§C.13.8.1). The most common and obvious way of depending on a template parameter *T* is to use a member of a *T* or to take an argument of type *T*. In *List<T>::print_all()*, *cout<<p->val* is a slightly more subtle example.

Errors that relate to the use of template parameters cannot be detected until the template is used. For example:

```
class Rec { /* ... */ };

void f(List<int>& li, List<Rec>& lr)
{
    li.print_all();
    lr.print_all();
}
```

The *li.print_all()* checks out fine, but *lr.print_all()* gives a type error because there is no << output operator defined for *Rec*. The earliest that errors relating to a template parameter can be detected is at the first point of use of the template for a particular template argument. That point is

usually called the *first point of instantiation*, or simply the *point of instantiation* (see §C.13.7). The implementation is allowed to postpone this checking until the program is linked. If we had only a declaration of ***print_all*( )** available in this translation unit, rather than its definition, the implementation might have had to delay type checking (see §13.7). Independently of when checking is done, the same set of rules is checked. Again, users prefer early checking. It is possible to express constraints on template arguments in terms of member functions (see §13.9[16]).

13.3 Function Templates [temp.fct]

For most people, the first and most obvious use of templates is to define and use container classes such as ***basic_string*** (§20.3), ***vector*** (§16.3), ***list*** (§17.2.2), and ***map*** (§17.4.1). Soon after, the need for template functions arises. Sorting an array is a simple example:

```
template<class T> void sort(vector<T>&);        // declaration

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);   // sort(vector<int>&);
    sort(vs);   // sort(vector<string>&);
}
```

When a template function is called, the types of the function arguments determine which version of the template is used; that is, the template arguments are deduced from the function arguments (§13.3.1).

Naturally, the template function must be defined somewhere (§C.13.7):

```
template<class T> void sort(vector<T>& v)        // definition
    // Shell sort (Knuth, Vol. 3, pg. 84).
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (v[j+gap]<v[j]) {        // swap v[j] and v[j+gap]
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

Please compare this definition to the ***sort*( )** defined in (§7.7). This templatized version is cleaner and shorter because it can rely on more information about the type of the elements it sorts. Most likely, it is also faster because it doesn't rely on a pointer to function for the comparison. This implies that no indirect function calls are needed and that inlining of a simple **<** is easy.

A further simplification is to use the standard library template ***swap*( )** (§18.6.8) to reduce the action to its natural form:

```
if (v[j+gap]<v[j]) swap(v[j],v[j+gap]);
```

This does not introduce any new overheads.

In this example, operator < is used for comparison. However, not every type has a < operator. This limits the use of this version of *sort*( ), but the limitation is easily avoided (see §13.4).

13.3.1 Function Template Arguments [temp.deduce]

Function templates are essential for writing generic algorithms to be applied to a wide variety of container types (§2.7.2, §3.8, Chapter 18). The ability to deduce the template arguments for a call from the function arguments is crucial.

A compiler can deduce type and non-type arguments from a call, provided the function argument list uniquely identifies the set of template arguments (§C.13.4). For example:

```
template<class T, int i> T lookup(Buffer<T,i>& b, const char* p);

class Record {
    const char[12];
    // ...
};

Record f(Buffer<Record,128>& buf, const char* p)
{
    return lookup(buf,p); // use the lookup() where T is Record and i is 128
}
```

Here, ***T*** is deduced to be ***Record*** and ***i*** is deduced to be ***128***.

Note that class template parameters are never deduced. The reason is that the flexibility provided by several constructors for a class would make such deduction impossible in many cases and obscure in many more. Specialization provides a mechanism for implicitly choosing between different implementations of a class (§13.5). If we need to create an object of a deduced type, we can often do that by calling a function to do the creation; see *make_pair*( ) in §17.4.1.2.

If a template argument cannot be deduced from the template function arguments (§C.13.4), we must specify it explicitly. This is done in the same way template arguments are explicitly specified for a template class. For example:

```
template<class T> class vector { /* ... */ };
template<class T> T* create(); // make a T and return a pointer to it

void f()
{
    vector<int> v;              // class, template argument 'int'
    int* p = create<int>();     // function, template argument 'int'
}
```

One common use of explicit specification is to provide a return type for a template function:

```
template<class T, class U> T implicit_cast(U u) { return u; }

void g(int i)
{
    implicit_cast(i);               // error: can't deduce T
    implicit_cast<double>(i);       // T is double; U is int
```

```
        implicit_cast<char,double>(i);  // T is char; U is double
        implicit_cast<char*,int>(i);    // T is char*; U is int; error: cannot convert int to char*
    }
```

As with default function arguments (§7.5), only trailing arguments can be left out of a list of explicit template arguments.

Explicit specification of template arguments allows the definition of families of conversion functions and object creation functions (§13.3.2, §C.13.1, §C.13.5). An explicit version of the implicit conversions (§C.6), such as *implicit_cast*( ), is frequently useful. The syntax for *dynamic_cast*, *static_cast*, etc., (§6.2.7, §15.4.1) matches the explicitly qualified template function syntax. However, the built-in type conversion operators supply operations that cannot be expressed by other language features.

13.3.2 Function Template Overloading [temp.over]

One can declare several function templates with the same name and even declare a combination of function templates and ordinary functions with the same name. When an overloaded function is called, overload resolution is necessary to find the right function or template function to invoke. For example:

```
    template<class T> T sqrt(T);
    template<class T> complex<T> sqrt(complex<T>);
    double sqrt(double);

    void f(complex<double> z)
    {
        sqrt(2);        // sqrt<int>(int)
        sqrt(2.0);      // sqrt(double)
        sqrt(z);        // sqrt<double>(complex<double>)
    }
```

In the same way that a template function is a generalization of the notion of a function, the rules for resolution in the presence of function templates are generalizations of the function overload resolution rules. Basically, for each template we find the specialization that is best for the set of function arguments. Then, we apply the usual function overload resolution rules to these specializations and all ordinary functions:

[1] Find the set of function template specializations (§13.2.2) that will take part in overload resolution. Do this by considering each function template and deciding which template arguments, if any, would be used if no other function templates or functions of the same name were in scope. For the call *sqrt*(*z*), this makes *sqrt*<*double*>(*complex*<*double*>) and *sqrt*< *complex*<*double*> >(*complex*<*double*>) candidates.

[2] If two template functions can be called and one is more specialized than the other (§13.5.1), consider only the most specialized template function in the following steps. For the call *sqrt*(*z*), this means that *sqrt*<*double*>(*complex*<*double*>) is preferred over *sqrt*< *complex*<*double*> >(*complex*<*double*>): any call that matches *sqrt*<*T*>(*complex*<*T*>) also matches *sqrt*<*T*>(*T*).

[3] Do overload resolution for this set of functions, plus any ordinary functions as for ordinary functions (§7.4). If a template function argument has been determined by template argument deduction (§13.3.1), that argument cannot also have promotions, standard conversions, or user-defined conversions applied. For `sqrt(2)`, `sqrt<int>(int)` is an exact match, so it is preferred over `sqrt(double)`.

[4] If a function and a specialization are equally good matches, the function is preferred. Consequently, `sqrt(double)` is preferred over `sqrt<double>(double)` for `sqrt(2.0)`.

[5] If no match is found, the call is an error. If we end up with two or more equally good matches, the call is ambiguous and is an error.

For example:

```
template<class T> T max(T,T);

const int s = 7;

void k()
{
    max(1,2);           // max<int>(1,2)
    max('a','b');       // max<char>('a','b')
    max(2.7,4.9);       // max<double>(2.7,4.9)
    max(s,7);           // max<int>(int(s),7) (trivial conversion used)

    max('a',1);         // error: ambiguous (no standard conversion)
    max(2.7,4);         // error: ambiguous (no standard conversion)
}
```

We could resolve the two ambiguities either by explicit qualification:

```
void f()
{
    max<int>('a',1);        // max<int>(int('a'),1)
    max<double>(2.7,4);     // max<double>(2.7,double(4))
}
```

or by adding suitable declarations:

```
inline int max(int i, int j) { return max<int>(i,j); }
inline double max(int i, double d) { return max<double>(i,d); }
inline double max(double d, int i) { return max<double>(d,i); }
inline double max(double d1, double d2) { return max<double>(d1,d2); }

void g()
{
    max('a',1);   // max(int('a'),1)
    max(2.7,4);   // max(2.7,double(4))
}
```

For ordinary functions, ordinary overloading rules (§7.4) apply, and the use of ***inline*** ensures that no extra overhead is imposed.

The definition of `max()` is trivial, so we could have written it explicitly. However, using a specialization of the template is an easy and general way of defining such resolution functions.

The overload resolution rules ensure that template functions interact properly with inheritance:

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb);      // f<int>(pb)
    f(pd);      // f<int>(static_cast<B<int>*>(pd)); standard conversion D<int>* to B<int>* used
}
```

In this example, the template function ***f*( )** accepts a ***B<T>**** for any type ***T***. We have an argument of type ***D<int>****, so the compiler easily deduces that by choosing ***T*** to be ***int***, the call can be uniquely resolved to a call of ***f*(*B<int>****)**.

A function argument that is not involved in the deduction of a template parameter is treated exactly as an argument of a non-template function. In particular, the usual conversion rules hold. Consider:

```
template<class C> int get_nth(C& p, int n);     // get n-th element
```

This function presumably returns the value of the n-th element of a container of type ***C***. Because ***C*** has to be deduced from an actual argument of ***get_nth*( )** in a call, conversions are not applicable to the first argument. However, the second argument is perfectly ordinary, so the full range of possible conversions is considered. For example:

```
class Index {
public:
    operator int();
    // ...
};

void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth(v,2);      // exact match
    int i2 = get_nth(v,s);      // standard conversion: short to int
    int i3 = get_nth(v,i);      // user-defined conversion: Index to int
}
```

13.4 Using Template Arguments to Specify Policy [temp.policy]

Consider how to sort strings. Three concepts are involved: the string, the element type, and the criteria used by the sort algorithm for comparing string elements.

We can't hardwire the sorting criteria into the container because the container can't (in general) impose its needs on the element types. We can't hardwire the sorting criteria into the element type because there are many different ways of sorting elements.

Consequently, the sorting criteria are built neither into the container nor into the element type. Instead, the criteria must be supplied when a specific operation needs to be performed. For example, if I have strings of characters representing names of Swedes, what collating criteria would I

like to use for a comparison? Two different collating sequences (numerical orderings of the characters) are commonly used for sorting Swedish names. Naturally, neither a general string type nor a general sort algorithm should know about the conventions for sorting names in Sweden. Therefore, any general solution requires that the sorting algorithm be expressed in general terms that can be defined not just for a specific type but also for a specific use of a specific type. For example, let us generalize the standard C library function ***strcmp*( )** for ***String***s of any type ***T*** (§13.2):

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
        if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
    return str1.length()-str2.length();
}
```

If someone wants ***compare*( )** to ignore case, to reflect locale, etc., that can be done by defining suitable ***C*::*eq*( )** and ***C*::*lt*( )**. This allows any (comparison, sorting, etc.) algorithm that can be described in terms of the operations supplied by the ''***C***-operations'' and the container to be expressed. For example:

```
template<class T> class Cmp { // normal, default compare
public:
    static int eq(T a, T b) { return a==b; }
    static int lt(T a, T b) { return a<b; }
};

class Literate { // compare Swedish names according to literary conventions
public:
    static int eq(char a, char b) { return a==b; }
    static int lt(char,char); // a table lookup based on character value (§13.9[14])
};
```

We can now choose the rules for comparison by explicit specification of the template arguments:

```
void f(String<char> swede1, String<char> swede2)
{
    compare< char,Cmp<char> >(swede1,swede2);
    compare< char,Literate >(swede1,swede2);
}
```

Passing the comparison operations as a template parameter has two significant benefits compared to alternatives such as passing pointers to functions. Several operations can be passed as a single argument with no run-time cost. In addition, the comparison operators ***eq*( )** and ***lt*( )** are trivial to inline, whereas inlining a call through a pointer to function requires exceptional attention from a compiler.

Naturally, comparison operations can be provided for user-defined types as well as built-in types. This is essential to allow general algorithms to be applied to types with nontrivial comparison criteria (see §18.4).

Each class generated from a class template gets a copy of each ***static*** member of the class template (see §C.13.1).

13.4.1 Default Template Parameters [temp.default]

Explicitly specifying the comparison criteria for each call is tedious. Fortunately, it is easy to pick a default so that only uncommon comparison criteria have to be explicitly specified. This can be implemented through overloading:

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2); // compare using C

template<class T>
int compare(const String<T>& str1, const String<T>& str2); // compare using Cmp<T>
```

Alternatively, we can supply the normal convention as a default template argument:

```
template<class T, class C = Cmp<T> >
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
        if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
    return str1.length()-str2.length();
}
```

Given that, we can write:

```
void f(String<char> swede1, String<char> swede2)
{
    compare(swede1,swede2);                    // use Cmp<char>
    compare<char,Literate>(swede1,swede2);     // use Literate
}
```

A less esoteric example (for non-Swedes) is comparing with and without taking case into account:

```
class No_case { /* ... */ };

void f(String<char> s1, String<char> s2)
{
    compare(s1,s2);                    // case sensitive
    compare<char,No_case>(s1,s2);      // not sensitive to case
}
```

The technique of supplying a policy through a template argument and then defaulting that argument to supply the most common policy is widely used in the standard library (e.g., §18.4). Curiously enough, it is not used for ***basic_string*** (§13.2, Chapter 20) comparisons. Template parameters used to express policies are often called ‘‘traits.’’ For example, the standard library string relies on ***char_traits*** (§20.2.1), the standard algorithms on iterator traits (§19.2.2), and the standard library containers on ***allocators*** (§19.4).

The semantic checking of a default argument for a template parameter is done if and (only) when that default argument is actually used. In particular, as long as we refrain from using the default template argument ***Cmp<T>*** we can ***compare*( )** strings of a type ***X*** for which ***Cmp<X>*** wouldn’t compile (say, because **<** wasn’t defined for an ***X***). This point is crucial in the design of the standard containers, which rely on a template argument to specify default values (§16.3.4).

13.5 Specialization [temp.special]

By default, a template gives a single definition to be used for every template argument (or combination of template arguments) that a user can think of. This doesn't always make sense for someone writing a template. I might want to say, ''if the template argument is a pointer, use this implementation; if it is not, use that implementation'' or ''give an error unless the template argument is a pointer derived from class *My_base*.'' Many such design concerns can be addressed by providing alternative definitions of the template and having the compiler choose between them based on the template arguments provided where they are used. Such alternative definitions of a template are called *user-defined specializations*, or simply, *user specializations*.

Consider likely uses of a *Vector* template:

```
template<class T> class Vector {        // general vector type
        T* v;
        int sz;
public:
        Vector();
        Vector(int);

        T& elem(int i) { return v[i]; }
        T& operator[](int i);

        void swap(Vector&);
        // ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Most *Vector*s will be *Vector*s of some pointer type. There are several reasons for this, but the primary reason is that to preserve run-time polymorphic behavior, we must use pointers (§2.5.4, §12.2.6). That is, anyone who practices object-oriented programming and also uses type-safe containers (such as the standard library containers) will end up with a lot of containers of pointers.

The default behavior of most C++ implementations is to replicate the code for template functions. This is good for run-time performance, but unless care is taken it leads to code bloat in critical cases such as the *Vector* example.

Fortunately, there is an obvious solution. Containers of pointers can share a single implementation. This can be expressed through specialization. First, we define a version (a specialization) of *Vector* for pointers to *void*:

```
template<> class Vector<void*> {
        void** p;
        // ...
        void*& operator[](int i);
};
```

This specialization can then be used as the common implementation for all *Vector*s of pointers.

The *template<>* prefix says that this is a specialization that can be specified without a template parameter. The template arguments for which the specialization is to be used are specified in <> brackets after the name. That is, the *<void*>* says that this definition is to be used as the implementation of every *Vector* for which *T* is void* .

The *Vector<void*>* is a complete specialization. That is, there is no template parameter to specify or deduce when we use the specialization; *Vector<void*>* is used for *Vector*s declared like this:

```
Vector<void*> vpv;
```

To define a specialization that is used for every *Vector* of pointers and only for *Vector*s of pointers, we need a *partial specialization*:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector() : Base() {}
    explicit Vector(int i) : Base(i) {}

    T*& elem(int i) { return static_cast<T*&>(Base::elem(i)); }
    T*& operator[](int i) { return static_cast<T*&>(Base::operator[](i)); }

    // ...
};
```

The specialization pattern *<T*>* after the name says that this specialization is to be used for every pointer type; that is, this definition is to be used for every *Vector* with a template argument that can be expressed as *T**. For example:

```
Vector<Shape*> vps; // <T*> is <Shape*> so T is Shape
Vector<int**> vppi;  // <T*> is <int**> so T is int*
```

Note that when a partial specialization is used, a template parameter is deduced from the specialization pattern; the template parameter is not simply the actual template argument. In particular, for *Vector<Shape*>*, *T* is *Shape* and not *Shape**.

Given this partial specialization of *Vector*, we have a shared implementation for all *Vector*s of pointers. The *Vector<T*>* class is simply an interface to *void** implemented exclusively through derivation and inline expansion.

It is important that this refinement of the implementation of *Vector* is achieved without affecting the interface presented to users. Specialization is a way of specifying alternative implementations for different uses of a common interface. Naturally, we could have given the general *Vector* and the *Vector* of pointers different names. However, when I tried that, many people who should have known better forgot to use the pointer classes and found their code much larger than expected. In this case, it is much better to hide the crucial implementation details behind a common interface.

This technique proved successful in curbing code bloat in real use. People who do not use a technique like this (in C++ or in other languages with similar facilities for type parameterization) have found that replicated code can cost megabytes of code space even in moderately-sized programs. By eliminating the time needed to compile those additional versions of the vector operations, this technique can also cut compile and link times dramatically. Using a single specialization

to implement all lists of pointers is an example of the general technique of minimizing code bloat by maximizing the amount of shared code.

The general template must be declared before any specialization. For example:

```
template<class T> class List<T*> { /* ... */ };
template<class T> class List { /* ... */ }; // error: general template after specialization
```

The critical information supplied by the general template is the set of template parameters that the user must supply to use it or any of its specializations. Consequently, a declaration of the general case is sufficient to allow the declaration or definition of a specialization:

```
template<class T> class List;
template<class T> class List<T*> { /* ... */ };
```

If used, the general template needs to be defined somewhere (§13.7).

If a user specializes a template somewhere, that specialization must be in scope for every use of the template with the type for which it was specialized. For example:

```
template<class T> class List { /* ... */ };
List<int*> li;
template<class T> class List<T*> { /* ... */ }; // error
```

Here, ***List*** was specialized for ***int**** after ***List<int*>*** had been used.

All specializations of a template must be declared in the same namespace as the template itself. If used, a specialization that is explicitly declared (as opposed to generated from a more general template) must also be explicitly defined somewhere (§13.7). In other words, explicitly specializing a template implies that no definition is generated for that specialization.

13.5.1 Order of Specializations [temp.special.order]

One specialization is *more specialized* than another if every argument list that matches its specialization pattern also matches the other, but not vice versa. For example:

```
template<class T> class Vector;          // general
template<class T> class Vector<T*>;      // specialized for any pointer
template<> class Vector<void*>;          // specialized for void*
```

Every type can be used as a template argument for the most general ***Vector***, but only pointers can be used for ***Vector<T*>*** and only ***void****s can be used for ***Vector<void*>***.

The most specialized version will be preferred over the others in declarations of objects, pointers, etc., (§13.5) and in overload resolution (§13.3.2).

A specialization pattern can be specified in terms of types composed using the constructs allowed for template parameter deduction (§13.3.1).

13.5.2 Template Function Specialization [temp.special.fct]

Naturally, specialization is also useful for template functions. Consider the Shell sort from §7.7 and §13.3. It compares elements using < and swaps elements using detailed code. A better definition would be:

```
template<class T> bool less(T a, T b) { return a<b; }

template<class T> void sort(Vector<T>& v)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (less(v[j+gap],v[j])) swap(v[j],v[j+gap]);
}
```

This does not improve the algorithm itself, but it allows improvements to its implementation. As written, ***sort*( )** will not sort a ***Vector<char*>*** correctly because < will compare the two ***char****s. That is, it will compare the addresses of the first ***char*** in each string. Instead, we would like it to compare the characters pointed to. A simple specialization of ***less*( )** for ***const char**** will take care of that:

```
template<> bool less<const char*>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

As for classes (§13.5), the ***template*<>** prefix says that this is a specialization that can be specified without a template parameter. The ***<const char*>*** after the template function name means that this specialization is to be used in cases where the template argument is ***const char****. Because the template argument can be deduced from the function argument list, we need not specify it explicitly. So, we could simplify the definition of the specialization:

```
template<> bool less<>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

Given the ***template*<>** prefix, the second empty <> is redundant, so we would typically simply write:

```
template<> bool less(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

I prefer this shorter form of declaration.

Consider the obvious definition of ***swap*( )**:

```
template<class T> void swap(T& x, T& y)
{
    T t = x;        // copy x to temporary
    x = y;          // copy y to x
    y = t;          // copy temporary to y
}
```

This is rather inefficient when invoked for *Vector*s of *Vector*s; it swaps *Vector*s by copying all elements. This problem can also be solved by appropriate specialization. A *Vector* object will itself hold only sufficient data to give indirect access to the elements (like *string*; §11.12, §13.2). Thus, a swap can be done by swapping those representations. To be able to manipulate that representation, I provided *Vector* with a member function *swap*( ) (§13.5):

```
template<class T> void Vector<T>::swap(Vector & a)      // swap representations
{
    swap(v,a.v);
    swap(sz,a.sz);

}
```

This member *swap*( ) can now be used to define a specialization of the general *swap*( ):

```
template<class T> void swap(Vector<T>& a, Vector<T>& b)
{
    a.swap(b);
}
```

These specializations of *less*( ) and *swap*( ) are used in the standard library (§16.3.9, §20.3.16). In addition, they are examples of widely applicable techniques. Specialization is useful when there is a more efficient alternative to a general algorithm for a set of template arguments (here, *swap*( )). In addition, specialization comes in handy when an irregularity of an argument type causes the general algorithm to give an undesired result (here, *less*( )). These ''irregular types'' are often the built-in pointer and array types.

13.6 Derivation and Templates [temp.derive]

Templates and derivation are mechanisms for building new types out of existing ones, and generally for writing useful code that exploits various forms of commonality. As shown in §3.7.1, §3.8.5, and §13.5, combinations of the two mechanisms are the basis for many useful techniques.

Deriving a template class from a non-template class is a way of providing a common implementation for a set of templates. The list from §13.5 is a good example of this:

```
template<class T> class list<T*> : private list<void*> { /* ... */ };
```

Another way of looking at such examples is that a template is used to provide an elegant and type-safe interface to an otherwise unsafe and inconvenient-to-use facility.

Naturally, it is often useful to derive one template class from another. One use of a base class is as a building block in the implementation of further classes. If the data or operations in such a base

class depend on a template parameter of a derived class, the base itself must be parameterized; ***Vec*** from §3.7.1 is an example of this:

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

The overload resolution rules for template functions ensure that functions work ''correctly'' for such derived types (§13.3.2).

Having the same template parameter for the base and derived class is the most common case, but it is not a requirement. Interesting, although less frequently used, techniques rely on passing the derived type itself to the base class. For example:

```
template <class C> class Basic_ops { // basic operators on containers
    bool operator==(const C&) const;    // compare all elements
    bool operator!=(const C&) const;
    // ...
};

template<class T> class Math_container : public Basic_ops< Math_container<T> > {
public:
    size_t size() const;
    T& operator[](size_t);
    // ...
};
```

This allows the definition of the basic operations on containers to be separate from the definition of the containers themselves and defined once only. However, the definition of operations such as `==` and `!=` must be expressed in terms of both the container and its elements, so the base class needs to be passed to the container template.

Assuming that a ***Math_container*** is similar to a traditional vector, the definitions of a ***Basic_ops*** member would look something like this:

```
template <class C> bool Basic_ops<C>::operator==(const C& a) const
{
    if (size() != a.size()) return false;
    for (int i = 0; i<size(); ++i)
        if ((*this)[i] != a[i]) return false;
    return true;
}
```

An alternative technique for keeping the containers and operations separate would be to combine them from template arguments rather than use derivation:

```
template<class T, class C> class Mcontainer {
    C elements;
public:
    // ...
    T& operator[](size_t i) { return elements[i]; }
```

```
        friend bool operator==(const Mcontainer&, const Mcontainer&); // compare elements
        friend bool operator!=(const Mcontainer&, const Mcontainer&);
        // ...
};

template<class T> class My_array { /* ... */ };

Mcontainer< double, My_array<double> > mc;
```

A class generated from a class template is a perfectly ordinary class. Consequently, it can have ***friend*** functions (§C.13.2). In this case, I used ***friend***s to achieve the conventional symmetric argument style for == and != (§11.3.2). One might also consider passing a template rather than a container as the *C* argument in such cases (§13.2.3).

13.6.1 Parameterization and Inheritance [temp.inherit]

A template parameterizes the definition of a type or a function with another type. Code implementing the template is identical for all parameter types, as is most code using the template. An abstract class defines an interface. Much code for different implementations of the abstract class can be shared in class hierarchies, and most code using the abstract class doesn't depend on its implementation. From a design perspective, the two approaches are so close that they deserve a common name. Since both allow an algorithm to be expressed once and applied to a variety of types, people sometimes refer to both as ***polymorphic***. To distinguish them, what virtual functions provide is called *run-time polymorphism*, and what templates offer is called *compile-time polymorphism* or *parametric polymorphism*.

So when do we choose to use a template and when do we rely on an abstract class? In either case, we manipulate objects that share a common set of operations. If no hierarchical relationship is required between these objects, they are best used as template arguments. If the actual types of these objects cannot be known at compile-time, they are best represented as classes derived from a common abstract class. If run-time efficiency is at a premium, that is, if inlining of operations is essential, a template should be used. This issue is discussed in greater detail in §24.4.1.

13.6.2 Member Templates [temp.member]

A class or a class template can have members that are themselves templates. For example:

```
template<class Scalar> class complex {
        Scalar re, im;
public:
        template<class T>
                complex(const complex<T>& c) : re(c.re), im(c.im) { }
        // ...
};

complex<float> cf(0,0);
complex<double> cd = cf;   // ok: uses float to double conversion
```

```
class Quad {
    // no conversion to int
};

complex<Quad> cq;
complex<int> ci = cq;          // error: no Quad to int conversion
```

In other words, you can construct a ***complex<T1>*** from a ***complex<T2>*** if and only if you can initialize a ***T1*** by a ***T2***. That seems reasonable.

Unfortunately, C++ accepts some unreasonable conversions between built-in types, such as from ***double*** to ***int***. Truncation problems could be caught at run time using a checked conversion in the style of ***implicit_cast*** (§13.3.1) and ***checked*** (§C.6.2.6):

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    complex() : re(0), im(0) { }
    complex(const complex<Scalar>& c) : re(c.re), im(c.im) { }

    template<class T2> complex(const complex<T2>& c)
        : re(checked_cast<Scalar>(c.real())), im(checked_cast<Scalar>(c.imag())) { }
    // ...
};
```

For completeness, I added a default constructor and a copy constructor. Curiously enough, a template constructor is never used to generate a copy constructor, so without the explicitly declared copy constructor, a default copy constructor would have been generated. In that case, that generated copy constructor would have been identical to the one I explicitly specified.

A member template cannot be ***virtual***. For example:

```
class Shape {
    // ...
    template<class T> virtual bool intersect(const T&) const =0; // error: virtual template
};
```

This must be illegal. If it were allowed, the traditional virtual function table technique for implementing virtual functions (§2.5.5) could not be used. The linker would have to add a new entry to the virtual table for class ***Shape*** each time someone called ***intersect()*** with a new argument type.

13.6.3 Inheritance Relationships [temp.rel.inheritance]

A class template is usefully understood as a specification of how particular types are to be created. In other words, the template implementation is a mechanism that generates types when needed based on the user's specification. Consequently, a class template is sometimes called a *type generator*.

As far as the C++ language rules are concerned, there is no relationship between two classes generated from a single class template. For example:

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

Given these declarations, people sometimes try to treat a ***set<Circle*>*** as a ***set<Shape*>***. This is a serious logical error based on a flawed argument: ''A ***Circle*** is a ***Shape***, so a set of ***Circles*** is also a set of ***Shapes***; therefore, I should be able to use a set of ***Circles*** as a set of ***Shapes***.'' The ''therefore'' part of this argument doesn't hold. The reason is that a set of ***Circle***s guarantees that the member of the set are ***Circles***; a set of ***Shape***s does not provide that guarantee. For example:

```
class Triangle : public Shape { /* ... */ };

void f(set<Shape*>& s)
{
    // ...
    s.insert(new Triangle());
    // ...
}

void g(set<Circle*>& s)
{
    f(s); // error, type mismatch: s is a set<Circle*>, not a set<Shape*>
}
```

This won't compile because there is no built-in conversion from ***set<Circle*>&*** to ***set<Shape*>&***. Nor should there be. The guarantee that the members of a ***set<Circle*>*** are ***Circle***s allows us to safely and efficiently apply ***Circle***-specific operations, such as determining the radius, to members of the set. If we allowed a ***set<Circle*>*** to be treated as a ***set<Shape*>***, we could no longer maintain that guarantee. For example, ***f()*** inserts a ***Triangle**** into its ***set<Shape*>*** argument. If the ***set<Shape*>*** could have been a ***set<Circle*>***, the fundamental guarantee that a ***set<Circle*>*** contains ***Circle***s only would have been violated.

13.6.3.1 Template Conversions [temp.mem.temp]

The example in the previous section demonstrates that there cannot be any *default* relationship between classes generated from the same templates. However, for some templates we would like to express such a relationship. For example, when we define a pointer template, we would like to reflect inheritance relationships among the objects pointed to. Member templates (§13.6.2) allow us to specify many such relationships where desired. Consider:

```
template<class T> class Ptr {    // pointer to T
    T* p;
public:
    Ptr(T*);
    template<class T2> operator Ptr<T2> (); // convert Ptr<T> to Ptr<T2>
    // ...
};
```

We would like to define the conversion operators to provide the inheritance relationships we are accustomed to for built-in pointers for these user-defined ***Ptr***s. For example:

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc;        // should work
    Ptr<Circle> pc2 = ps;      // should give error
}
```

We want to allow the first initialization if and only if ***Shape*** really is a direct or indirect public base class of ***Circle***. In general, we need to define the conversion operator so that the ***Ptr<T>*** to ***Ptr<T2>*** conversion is accepted if and only if a ***T**** can be assigned to a ***T2****. That can be done like this:

```
template<class T>
    template<class T2>
        Ptr<T>::operator Ptr<T2> () { return Ptr<T2>(p); }
```

The return statement will compile if and only if ***p*** (which is a ***T****) can be an argument to the ***Ptr<T2>(T2*)*** constructor. Therefore, if ***T**** can be implicitly converted into a ***T2****, the ***Ptr<T>*** to ***Ptr<T2>*** conversion will work. For example

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc;        // ok: can convert Circle* to Shape*
    Ptr<Circle> pc2 = ps;      // error: cannot convert Shape* to Circle*
}
```

Be careful to define logically meaningful conversions only.

Note that the template parameter lists of a template and its template member cannot be combined. For example:

```
template<class T, class T2>    // error
    Ptr<T>::operator Ptr<T2> () { return Ptr<T2>(p); }
```

13.7 Source Code Organization [temp.source]

There are two obvious ways of organizing code using templates:

[1] Include template definitions before their use in a translation unit.

[2] Include template declarations (only) before their use in a translation unit, and compile their definitions separately.

In addition, template functions are sometimes first declared, then used, and finally defined in a single translation unit.

To see the differences between the two main approaches, consider a simple template:

```
#include<iostream>

template<class T> void out(const T& t) { std::cerr << t; }
```

We could call this ***out.c*** and ***#include*** it wherever ***out()*** was needed. For example:

```
// user1.c:
        #include "out.c"
        // use out()

// user2.c:
        #include "out.c"
        // use out()
```

That is, the definition of ***out***( ) and all declarations it depends on are ***#include***d in several different compilation units. It is up to the compiler to generate code when needed (only) and to optimize the process of reading redundant definitions. This strategy treats template functions the same way as inline functions.

One obvious problem with this is that everything on which the definition of ***out***( ) depends is added to each file using ***out***( ), thus increasing the amount of information that the compiler must process. Another problem is that users may accidentally come to depend on declarations included only for the benefit of the definition of ***out***( ). This danger can be minimized by using namespaces, by avoiding macros, and generally by reducing the amount of information included.

The separate compilation strategy is the logical conclusion of this line of thinking: if the template definition isn't included in the user code, none of its dependencies can affect that code. Thus we split the original ***out.c*** into two files:

```
// out.h:
        template<class T> void out(const T& t);

// out.c:
        #include<iostream>
        #include "out.h"

        export template<class T> void out(const T& t) { std::cerr << t; }
```

The file ***out.c*** now holds all of the information needed to define ***out***( ), and ***out.h*** holds only what is needed to call it. A user ***#include***s only the declaration (the interface):

```
// user1.c:
        #include "out.h"
        // use out()

// user2.c:
        #include "out.h"
        // use out()
```

This strategy treats template functions the same way it does non-inline functions. The definition (in ***out.c***) is compiled separately, and it is up to the implementation to find the definition of ***out***( ) when needed. This strategy also puts a burden on the implementation. Instead of having to filter out redundant copies of a template definition, the implementation must find the unique definition when needed.

Note that to be accessible from other compilation units, a template definition must be explicitly declared ***export*** (§9.2.3). This can be done by adding ***export*** to the definition or to a preceding declaration. Otherwise, the definition must be in scope wherever the template is used.

Which strategy or combination of strategies is best depends on the compilation and linkage

system used, the kind of application you are building, and the external constraints on the way you build systems. Generally, inline functions and other small functions that primarily call other template functions are candidates for inclusion into every compilation unit in which they are used. On an implementation with average support from the linker for template instantiation, doing this can speed up compilation and improve error messages.

Including a definition makes it vulnerable to having its meaning affected by macros and declarations in the context into which it is included. Consequently, larger template functions and template functions with nontrivial context dependencies are better compiled separately. Also, if the definition of a template requires a large number of declarations, these declarations can have undesirable side effects if they are included into the context in which the template is used.

I consider the approach of separately compiling template definitions and including declarations only in user code ideal. However, the application of ideals must be tempered by practical constraints, and separate compilation of templates is expensive on some implementations.

Whichever strategy is used, non-***inline static*** members (§C.13.1) must have a unique definition in some compilation unit. This implies that such members are best not used for templates that are otherwise included in many translation units.

One ideal is for code to work the same whether it is compiled as a single unit or separated into several separately translated units. That ideal should be approached by restricting a template definition's dependency on its environment rather than by trying to carry as much as possible of its definition context with it into the instantiation process.

13.8 Advice [temp.advice]

[1] Use templates to express algorithms that apply to many argument types; §13.3.
[2] Use templates to express containers; §13.2.
[3] Provide specializations for containers of pointers to minimize code size; §13.5.
[4] Always declare the general form of a template before specializations; §13.5.
[5] Declare a specialization before its use; §13.5.
[6] Minimize a template definition's dependence on its instantiation contexts; §13.2.5, §C.13.8.
[7] Define every specialization you declare; §13.5.
[8] Consider if a template needs specializations for C-style strings and arrays; §13.5.2.
[9] Parameterize with a policy object; §13.4.
[10] Use specialization and overloading to provide a single interface to implementations of the same concept for different types; §13.5.
[11] Provide a simple interface for simple cases and use overloading and default arguments to express less common cases; §13.5, §13.4.
[12] Debug concrete examples before generalizing to a template; §13.2.1.
[13] Remember to ***export*** template definitions that need to be accessible from other translation units; §13.7.
[14] Separately compile large templates and templates with nontrivial context dependencies; §13.7.
[15] Use templates to express conversions but define those conversions very carefully; §13.6.3.1.
[16] Where necessary, constrain template arguments using a ***constraint***( ) member function; §13.9[16].

[17] Use explicit instantiation to minimize compile time and link time; §C.13.10.
[18] Prefer a template over derived classes when run-time efficiency is at a premium; §13.6.1.
[19] Prefer derived classes over a template if adding new variants without recompilation is important; §13.6.1.
[20] Prefer a template over derived classes when no common base can be defined; §13.6.1.
[21] Prefer a template over derived classes when built-in types and structures with compatibility constraints are important; §13.6.1.

13.9 Exercises [temp.exercises]

1. (∗2) Fix the errors in the definition of ***List*** from §13.2.5 and write out C++ code equivalent to what the compiler must generate for the definition of ***List*** and the function ***f*( )**. Run a small test case using your hand-generated code and the code generated by the compiler from the template version. If possible on your system given your knowledge, compare the generated code.
2. (∗3) Write a singly-linked list class template that accepts elements of any type derived from a class ***Link*** that holds the information necessary to link elements. This is called an *intrusive list*. Using this list, write a singly-linked list that accepts elements of any type (a non-intrusive list). Compare the performance of the two list classes and discuss the tradeoffs between them.
3. (∗2.5) Write intrusive and non-intrusive doubly-linked lists. What operations should be provided in addition to the ones you found necessary to supply for a singly-linked list?
4. (∗2) Complete the ***String*** template from §13.2 based on the ***String*** class from §11.12.
5. (∗2) Define a ***sort*( )** that takes its comparison criterion as a template argument. Define a class ***Record*** with two data members ***count*** and ***price***. Sort a ***vector<Record>*** on each data member.
6. (∗2) Implement a ***qsort*( )** template.
7. (∗2) Write a program that reads **(*key*, *value*)** pairs and prints out the sum of the ***value***s corresponding to each distinct ***key***. Specify what is required for a type to be a ***key*** and a ***value***.
8. (∗2.5) Implement a simple ***Map*** class based on the ***Assoc*** class from §11.8. Make sure ***Map*** works correctly using both C-style strings and ***string***s as keys. Make sure ***Map*** works correctly for types with and without default constructors. Provide a way of iterating over the elements of a ***Map***.
9. (∗3) Compare the performance of the word count program from §11.8 against a program not using an associative array. Use the same style of I/O in both cases.
10. (∗3) Re-implement ***Map*** from §13.9[8] using a more suitable data structure (e.g., a red-black tree or a Splay tree).
11. (∗2.5) Use ***Map*** to implement a topological sort function. Topological sort is described in [Knuth,1968] vol. 1 (second edition), pg 262.
12. (∗1.5) Make the sum program from §13.9[7] work correctly for names containing spaces; for example, ‘‘thumb tack.’’
13. (∗2) Write ***readline*( )** templates for different kinds of lines. For example (item,count,price).
14. (∗2) Use the technique outlined for ***Literate*** in §13.4 to sort strings in reverse lexicographical order. Make sure the technique works both for C++ implementations where ***char*** is ***signed*** and for C++ implementations where it is ***unsigned***. Use a variant of that technique to provide a sort that is not case-sensitive.

15. (∗1.5) Construct an example that demonstrates at least three differences between a function template and a macro (not counting the differences in definition syntax).
16. (∗2) Devise a scheme that ensures that the compiler tests general constraints on the template arguments for every template for which an object is constructed. It is not sufficient just to test constraints of the form ‘‘the argument ***T*** must be a class derived from ***My_base***.’’

14

Exception Handling

Don´t interrupt me
while I´m interrupting.
– Winston S. Churchill

Error handling — grouping of exceptions — catching exceptions — catch all — rethrow — resource management — ***auto_ptr*** — exceptions and ***new*** — resource exhaustion — exceptions in constructors — exceptions in destructors — exceptions that are not errors — exception specifications — unexpected exceptions — uncaught exceptions — exceptions and efficiency — error-handling alternatives — standard exceptions — advice — exercises.

14.1 Error Handling [except.error]

As pointed out in §8.3, the author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors but cannot detect them – or else they would have been handled in the user's code and not left for the library to find. The notion of an *exception* is provided to help deal with such problems. The fundamental idea is that a function that finds a problem it cannot cope with *throws* an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle that kind of problem can indicate that it is willing to *catch* that exception (§2.4.2, §8.3).

This style of error handling compares favorably with more traditional techniques. Consider the alternatives. Upon detecting a problem that cannot be handled locally, the program could:

[1] terminate the program,
[2] return a value representing ''error,''
[3] return a legal value and leave the program in an illegal state, or
[4] call a function supplied to be called in case of ''error.''

Case [1], ''terminate the program,'' is what happens by default when an exception isn't caught. For most errors, we can and must do better. In particular, a library that doesn't know about the purpose and general strategy of the program in which it is embedded cannot simply ***exit***( ) or ***abort***( ). A library that unconditionally terminates cannot be used in a program that cannot afford to crash. One way of viewing exceptions is as a way of giving control to a caller when no meaningful action can be taken locally.

Case [2], ''return an error value,'' isn't always feasible because there is often no acceptable ''error value.'' For example, if a function returns an ***int***, *every* ***int*** might be a plausible result. Even where this approach is feasible, it is often inconvenient because every call must be checked for the error value. This can easily double the size of a program (§14.8). Consequently, this approach is rarely used systematically enough to detect all errors.

Case [3], ''return a legal value and leave the program in an illegal state,'' has the problem that the calling function may not notice that the program has been put in an illegal state. For example, many standard C library functions set the global variable ***errno*** to indicate an error (§20.4.1, §22.3). However, programs typically fail to test ***errno*** consistently enough to avoid consequential errors caused by values returned from failed calls. Furthermore, the use of global variables for recording error conditions doesn't work well in the presence of concurrency.

Exception handling is not meant to handle problems for which case [4], ''call an error-handler function,'' is relevant. However, in the absence of exceptions, an error-handler function has exactly the three other cases as alternatives for how *it* handles the error. For a further discussion of error-handling functions and exceptions, see §14.4.5.

The exception-handling mechanism provides an alternative to the traditional techniques when they are insufficient, inelegant, and error-prone. It provides a way of explicitly separating error-handling code from ''ordinary'' code, thus making the program more readable and more amenable to tools. The exception-handling mechanism provides a more regular style of error handling, thus simplifying cooperation between separately written program fragments.

One aspect of the exception-handling scheme that will appear novel to C and Pascal programmers is that the default response to an error (especially to an error in a library) is to terminate the program. The traditional response has been to muddle through and hope for the best. Thus, exception handling makes programs more ''brittle'' in the sense that more care and effort must be taken to get a program to run acceptably. This seems preferable, though, to getting wrong results later in the development process – or after the development process is considered complete and the program is handed over to innocent users. Where termination is unacceptable, we can catch all exceptions (§14.3.2) or catch all exceptions of a specific kind (§14.6.2). Thus, an exception terminates a program only if a programmer allows it to terminate. This is preferable to the unconditional termination that happens when a traditional incomplete recovery leads to a catastrophic error.

Sometimes people have tried to alleviate the unattractive aspects of ''muddling through'' by writing out error messages, putting up dialog boxes asking the user for help, etc. Such approaches are primarily useful in debugging situations in which the user is a programmer familiar with the structure of the program. In the hands of nondevelopers, a library that asks the (possibly absent) user/operator for help is unacceptable. Also, in many cases error messages have no place to go (say, if the program runs in an environment in which ***cerr*** doesn't connect to anything a user notices); they would be incomprehensible to an end user anyway. At a minimum, the error message might be in the wrong natural language (say, in Finnish to a English user). Worse, the error

message would typically refer to library concepts completely unknown to a user (say, ‘‘bad argument to atan2,’’ caused by bad input to a graphics system). A good library doesn’t ‘‘blabber’’ in this way. Exceptions provide a way for code that detects a problem from which it cannot recover to pass the problem on to some part of the system that might be able to recover. Only a part of the system that has some idea of the context in which the program runs has any chance of composing a meaningful error message.

The exception-handling mechanism can be seen as a run-time analog to the compile-time type checking and ambiguity control mechanisms. It makes the design process more important and can increase the work needed to get an initial and buggy version of a program running. However, the result is code that has a much better chance to run as expected, to run as an acceptable part of a larger program, to be comprehensible to other programmers, and to be amenable to manipulation by tools. Similarly, exception handling provides specific language features to support ‘‘good style’’ in the same way other C++ features support ‘‘good style’’ that can be practiced only informally and incompletely in languages such as C and Pascal.

It should be recognized that error handling will remain a difficult task and that the exception-handling mechanism – although more formalized than the techniques it replaces – is still relatively unstructured compared with language features involving only local control flow. The C++ exception-handling mechanism provides the programmer with a way of handling errors where they are most naturally handled, given the structure of a system. Exceptions make the complexity of error handling visible. However, exceptions are not the cause of that complexity. Be careful not to blame the messenger for bad news.

This may be a good time to review §8.3, where the basic syntax, semantics, and style-of-use aspects of exception handling are presented.

14.1.1 Alternative Views on Exceptions [except.views]

‘‘Exception’’ is one of those words that means different things to different people. The C++ exception-handling mechanism is designed to support handling of errors and other exceptional conditions (hence the name). In particular, it is intended to support error handling in programs composed of independently developed components.

The mechanism is designed to handle only synchronous exceptions, such as array range checks and I/O errors. Asynchronous events, such as keyboard interrupts and certain arithmetic errors, are not necessarily exceptional and are not handled directly by this mechanism. Asynchronous events require mechanisms fundamentally different from exceptions (as defined here) to handle them cleanly and efficiently. Many systems offer mechanisms, such as signals, to deal with asynchrony, but because these tend to be system-dependent, they are not described here.

The exception-handling mechanism is a nonlocal control structure based on stack unwinding (§14.4) that can be seen as an alternative return mechanism. There are therefore legitimate uses of exceptions that have nothing to do with errors (§14.5). However, the primary aim of the exception-handling mechanism and the focus of this chapter is error handling and the support of fault tolerance.

Standard C++ doesn’t have the notion of a thread or a process. Consequently, exceptional circumstances relating to concurrency are not discussed here. The concurrency facilities available on your system are described in its documentation. Here, I’ll just note that the C++ exception-

handling mechanism was designed to be effective in a concurrent program as long as the programmer (or system) enforces basic concurrency rules, such as properly locking a shared data structure while using it.

The C++ exception-handling mechanisms are provided to report and handle errors and exceptional events. However, the programmer must decide what it means to be exceptional in a given program. This is not always easy (§14.5). Can an event that happens most times a program is run be considered exceptional? Can an event that is planned for and handled be considered an error? The answer to both questions is yes. ''Exceptional'' does not mean ''almost never happens'' or ''disastrous.'' It is better to think of an exception as meaning ''some part of the system couldn't do what it was asked to do.'' Usually, we can then try something else. Exception ***throw***s should be infrequent compared to function calls or the structure of the system has been obscured. However, we should expect most large programs to ***throw*** and ***catch*** at least some exceptions in the course of a normal and successful run.

14.2 Grouping of Exceptions [except.grouping]

An exception is an object of some class representing an exceptional occurrence. Code that detects an error (often a library) ***throw***s an object (§8.3). A piece of code expresses desire to handle an exception by a ***catch*** clause. The effect of a ***throw*** is to unwind the stack until a suitable ***catch*** is found (in a function that directly or indirectly invoked the function that threw the exception).

Often, exceptions fall naturally into families. This implies that inheritance can be useful to structure exceptions and to help exception handling. For example, the exceptions for a mathematical library might be organized like this:

```
class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...
```

This allows us to handle any ***Matherr*** without caring precisely which kind it is. For example:

```
void f()
{
    try {
        // ...
    }
    catch (Overflow) {
        // handle Overflow or anything derived from Overflow
    }
    catch (Matherr) {
        // handle any Matherr that is not Overflow
    }
}
```

Here, an ***Overflow*** is handled specifically. All other ***Matherr*** exceptions will be handled by the general case.

Organizing exceptions into hierarchies can be important for robustness of code. For example, consider how you would handle all exceptions from a library of mathematical functions without such a grouping mechanism. This would have to be done by exhaustively listing the exceptions:

```
void g()
{
    try {
        // ...
    }
    catch (Overflow) { /* ... */ }
    catch (Underflow) { /* ... */ }
    catch (Zerodivide) { /* ... */ }
}
```

This is not only tedious, but a programmer can easily forget to add an exception to the list. Consider what would be needed if we didn't group math exceptions. When we added a new exception to the math library, every piece of code that tried to handle every math exception would have to be modified. In general, such universal update is not feasible after the initial release of the library. Often, there is no way of finding every relevant piece of code. Even when there is, we cannot in general assume that every piece of source code is available or that we would be willing to make changes if it were. These recompilation and maintenance problems would lead to a policy that no new exceptions can be added to a library after its first release; *that* would be unacceptable for almost all libraries. This reasoning leads exceptions to be defined as per-library or per-subsystem class hierarchies (§14.6.2).

Please note that neither the built-in mathematical operations nor the basic math library (shared with C) reports arithmetic errors as exceptions. One reason for this is that detection of some arithmetic errors, such as divide-by-zero, are asynchronous on many pipelined machine architectures. The ***Matherr*** hierarchy described here is only an illustration. The standard library exceptions are described in §14.10.

14.2.1 Derived Exceptions [except.derived]

The use of class hierarchies for exception handling naturally leads to handlers that are interested only in a subset of the information carried by exceptions. In other words, an exception is typically caught by a handler for its base class rather than by a handler for its exact class. The semantics for catching and naming an exception are identical to those of a function accepting an argument. That is, the formal argument is initialized with the argument value (§7.2). This implies that the exception thrown is ''sliced'' to the exception caught (§12.2.3). For example:

```
class Matherr {
    // ...
    virtual void debug_print() const { cerr << "Math error"; }
};

class Int_overflow: public Matherr {
    const char* op;
    int a1, a2;
```

```
public:
    Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
    virtual void debug_print() const { cerr << op << '(' << a1 << ',' << a2 << ')'; }
    // ...
};

void f()
{
    try {
        g();
    }
    catch (Matherr m) {
        // ...
    }
}
```

When the ***Matherr*** handler is entered, ***m*** is a ***Matherr*** object – even if the call to ***g*()** threw ***Int_overflow***. This implies that the extra information found in an ***Int_overflow*** is inaccessible.

As always, pointers or references can be used to avoid losing information permanently. For example, we might write:

```
int add(int x, int y)
{
    if ((x>0 && y>0 && x>INT_MAX-y) || (x<0 && y<0 && x<INT_MIN-y))
        throw Int_overflow("+",x,y);

    return x+y;     // x+y will not overflow
}

void f()
{
    try {
        int i1 = add(1,2);
        int i2 = add(INT_MAX,-2);
        int i3 = add(INT_MAX,2);     // here we go!
    }
    catch (Matherr& m) {
        // ...
        m.debug_print();
    }
}
```

The last call of ***add*()** triggers an exception that causes ***Int_overflow*::*debug_print*()** to be invoked. Had the exception been caught by value rather than by reference, ***Matherr*::*debug_print*()** would have been invoked instead.

14.2.2 Composite Exceptions [except.composite]

Not every grouping of exceptions is a tree structure. Often, an exception belongs to two groups. For example:

```
class Netfile_err : public Network_err, public File_system_err { /* ... */ };
```

Such a ***Netfile_err*** can be caught by functions dealing with network exceptions:

```
void f()
{
    try {
        // something
    }
    catch(Network_err& e) {
        // ...
    }
}
```

and also by functions dealing with file system exceptions:

```
void g()
{
    try {
        // something else
    }
    catch(File_system_err& e) {
        // ...
    }
}
```

This nonhierarchical organization of error handling is important where services, such as networking, are transparent to users. In this case, the writer of ***g*()** might not even be aware that a network is involved (see also §14.6).

14.3 Catching Exceptions [except.catch]

Consider:

```
void f()
{
    try {
        throw E();
    }
    catch(H) {
        // when do we get here?
    }
}
```

The handler is invoked:

[1] If ***H*** is the same type as ***E***.
[2] If ***H*** is an unambiguous public base of ***E***.
[3] If ***H*** and ***E*** are pointer types and [1] or [2] holds for the types to which they refer.
[4] If ***H*** is a reference and [1] or [2] holds for the type to which ***H*** refers.

In addition, we can add ***const*** to the type used to catch an exception in the same way that we can add it to a function parameter. This doesn't change the set of exceptions we can catch; it only restricts us from modifying the exception caught.

In principle, an exception is copied when it is thrown, so the handler gets hold of a copy of the original exception. In fact, an exception may be copied several times before it is caught. Consequently, we cannot throw an exception that cannot be copied. The implementation may apply a wide variety of strategies for storing and transmitting exceptions. It is guaranteed, however, that there is sufficient memory to allow ***new*** to throw the standard out-of-memory exception, ***bad_alloc*** (§14.4.5).

14.3.1 Re-Throw [except.rethrow]

Having caught an exception, it is common for a handler to decide that it can't completely handle the error. In that case, the handler typically does what can be done locally and then throws the exception again. Thus, an error can be handled where it is most appropriate. This is the case even when the information needed to best handle the error is not available in a single place, so that the recovery action is best distributed over several handlers. For example:

```
void h()
{
    try {
        // code that might throw Math errors
    }
    catch (Matherr) {
        if (can_handle_it_completely) {
            // handle the Matherr

            return;
        }
        else {
            // do what can be done here

            throw;      // re-throw the exception
        }
    }
}
```

A re-throw is indicated by a ***throw*** without an operand. If a re-throw is attempted when there is no exception to re-throw, ***terminate*()** (§14.7) will be called. A compiler can detect and warn about some, but not all, such cases.

The exception re-thrown is the original exception caught and not just the part of it that was accessible as a ***Matherr***. In other words, had an ***Int_overflow*** been thrown, a caller of ***h*()** could still catch an ***Int_overflow*** that ***h*()** had caught as a ***Matherr*** and decided to re-throw.

14.3.2 Catch Every Exception [except.every]

A degenerate version of this catch-and-rethrow technique can be important. As for functions, the ellipsis **...** indicates ''any argument'' (§7.6), so ***catch*(...)** means ''catch any exception.'' For example:

```
void m()
{
    try {
        // something
    }
    catch ( . . . ) {        // handle every exception
        // cleanup
        throw;
    }
}
```

That is, if any exception occurs as the result of executing the main part of **m( )**, the cleanup action in the handler is invoked. Once the local cleanup is done, the exception that caused the cleanup is re-thrown to trigger further error handling. See §14.6.3.2 for a technique to gain information about an exception caught by a **. . .** handler.

One important aspect of error handling in general and exception handling in particular is to maintain invariants assumed by the program (§24.3.7.1). For example, if **m( )** is supposed to leave certain pointers in the state in which it found them, then we can write code in the handler to give them acceptable values. Thus, a ''catch every exception'' handler can be used to maintain arbitrary invariants. However, for many important cases such a handler is not the most elegant solution to this problem (see §14.4).

14.3.2.1 Order of Handlers [except.order]

Because a derived exception can be caught by handlers for more than one exception type, the order in which the handlers are written in a ***try*** statement is significant. The handlers are tried in order. For example:

```
void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // handle any stream io error (§14.10)
    }
    catch (std::exception& e) {
        // handle any standard library exception (§14.10)
    }
    catch ( . . . ) {
        // handle any other exception (§14.3.2)
    }
}
```

Because the compiler knows the class hierarchy, it can catch many logical mistakes. For example:

```
void g()
{
    try {
        // ...
    }
    catch (...) {
        // handle every exception (§14.3.2)
    }
    catch (std::exception& e) {
        // handle any standard library exception (§14.10)
    }
    catch (std::bad_cast) {
        // handle dynamic_cast failure (§15.4.2)
    }
}
```

Here, the ***exception*** will never be considered. Even if we removed the ‘‘catch-all’’ handler, ***bad_cast*** wouldn’t be considered because it is derived from ***exception***.

14.4 Resource Management [except.resource]

When a function acquires a resource – that is, it opens a file, allocates some memory from the free store, sets an access control lock, etc., – it is often essential for the future running of the system that the resource be properly released. Often that ‘‘proper release’’ is achieved by having the function that acquired it release it before returning to its caller. For example:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "w");
    // use f
    fclose(f);
}
```

This looks plausible until you realize that if something goes wrong after the call of ***fopen***( ) and before the call of ***fclose***( ), an exception may cause ***use_file***( ) to be exited without ***fclose***( ) being called. Exactly the same problem can occur in languages that do not support exception handling. For example, the standard C library function ***longjmp***( ) can cause the same problem. Even an ordinary ***return***-statement could exit ***use_file*** without closing ***f***.

A first attempt to make ***use_file***( ) to be fault-tolerant looks like this:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r");
    try {
        // use f
    }
```

```
        catch ( . . . ) {
            fclose(f);
            throw;
        }
        fclose(f);
    }
```

The code using the file is enclosed in a *try* block that catches every exception, closes the file, and re-throws the exception.

The problem with this solution is that it is verbose, tedious, and potentially expensive. Furthermore, any verbose and tedious solution is error-prone because programmers get bored. Fortunately, there is a more elegant solution. The general form of the problem looks like this:

```
void acquire( )
{
    // acquire resource 1
    // ...
    // acquire resource n

    // use resources

    // release resource n
    // ...
    // release resource 1
}
```

It is typically important that resources are released in the reverse order of their acquisition. This strongly resembles the behavior of local objects created by constructors and destroyed by destructors. Thus, we can handle such resource acquisition and release problems by a suitable use of objects of classes with constructors and destructors. For example, we can define a class ***File_ptr*** that acts like a ***FILE****:

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a) { p = fopen(n,a); }
    File_ptr(FILE* pp) { p = pp; }
    ~File_ptr( ) { fclose(p); }

    operator FILE*( ) { return p; }
};
```

We can construct a ***File_ptr*** given either a ***FILE**** or the arguments required for ***fopen( )***. In either case, a ***File_ptr*** will be destroyed at the end of its scope and its destructor will close the file. Our program now shrinks to this minimum:

```
void use_file(const char* fn)
{
    File_ptr f(fn, "r");
    // use f
}
```

The destructor will be called independently of whether the function is exited normally or exited because an exception is thrown. That is, the exception-handling mechanisms enable us to remove the error-handling code from the main algorithm. The resulting code is simpler and less error-prone than its traditional counterpart.

The process of searching ‘‘up through the stack’’ to find a handler for an exception is commonly called ‘‘stack unwinding.’’ As the call stack is unwound, the destructors for constructed local objects are invoked.

14.4.1 Using Constructors and Destructors [except.using]

The technique for managing resources using local objects is usually referred to as ‘‘resource acquisition is initialization.’’ This is a general technique that relies on the properties of constructors and destructors and their interaction with exception handling.

An object is not considered constructed until its constructor has completed. Then and only then will stack unwinding call the destructor for the object. An object composed of sub-objects is constructed to the extent that its sub-objects have been constructed. An array is constructed to the extent that its elements have been constructed (and only fully constructed elements are destroyed during unwinding).

A constructor tries to ensure that its object is completely and correctly constructed. When that cannot be achieved, a well-written constructor restores – as far as possible – the state of the system to what it was before creation. Ideally, naively written constructors always achieve one of these alternatives and don’t leave their objects in some ‘‘half-constructed’’ state. This can be achieved by applying the ‘‘resource acquisition is initialization’’ technique to the members.

Consider a class ***X*** for which a constructor needs to acquire two resources: a file ***x*** and a lock ***y***. This acquisition might fail and throw an exception. Class ***X***’s constructor must never return having acquired the file but not the lock. Furthermore, this should be achieved without imposing a burden of complexity on the programmer. We use objects of two classes, ***File_ptr*** and ***Lock_ptr***, to represent the acquired resources. The acquisition of a resource is represented by the initialization of the local object that represents the resource:

```
class X {
    File_ptr aa;
    Lock_ptr bb;
public:
    X(const char* x, const char* y)
        : aa(x,"rw"), // acquire ‘x’
          bb(y)       // acquire ‘y’
    {}
    // ...
};
```

Now, as in the local object case, the implementation can take care of all of the bookkeeping. The user doesn’t have to keep track at all. For example, if an exception occurs after ***aa*** has been constructed but before ***bb*** has been, then the destructor for ***aa*** but not for ***bb*** will be invoked.

This implies that where this simple model for acquisition of resources is adhered to, the author of the constructor need not write explicit exception-handling code.

The most common resource acquired in an ad-hoc manner is memory. For example:

```
class Y {
        int* p;
        void init();
public:
        Y(int s) { p = new int[s]; init(); }
        ~Y() { delete[] p; }
        // ...
};
```

This practice is common and can lead to ''memory leaks.'' If an exception is thrown by ***init*()**, then the store acquired will not be freed; the destructor will not be called because the object wasn't completely constructed. A safe variant is:

```
class Z {
        vector<int> p;
        void init();
public:
        Z(int s) : p(s) { init(); }
        // ...
};
```

The memory used by ***p*** is now managed by ***vector***. If ***init*()** throws an exception, the memory acquired will be freed when the destructor for ***p*** is (implicitly) invoked.

14.4.2 Auto_ptr [except.autoptr]

The standard library provides the template class ***auto_ptr***, which supports the ''resource acquisition is initialization'' technique. Basically, an ***auto_ptr*** is initialized by a pointer and can be dereferenced in the way that a pointer can. Also, the object pointed to will be implicitly deleted at the end of the ***auto_ptr***'s scope. For example:

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb) // remember to delete pb on exit
{
        auto_ptr<Shape> p(new Rectangle(p1,p2));     // p points to a rectangle
        auto_ptr<Shape> pbox(pb);

        p->rotate(45); // use auto_ptr<Shape> exactly as a Shape*
        // ...
        if (in_a_mess) throw Mess();
        // ...
}
```

Here the ***Rectangle***, the ***Shape*** pointed to by ***pb***, and the ***Circle*** pointed to by ***pc*** are deleted whether or not an exception is thrown.

To achieve this *ownership semantics* (also called *destructive copy semantics*), ***auto_ptr***s have a copy semantics that differs radically from that of ordinary pointers: When one ***auto_ptr*** is copied into another, the source no longer points to anything. Because copying an ***auto_ptr*** modifies it, a ***const auto_ptr*** cannot be copied.

The ***auto_ptr*** template is declared in ***<memory>***. It can be described by an implementation:

```
template<class X> class std::auto_ptr {
    template <class Y> struct auto_ptr_ref { /* ... */ }; // helper class
    X* ptr;
public:
    typedef X element_type;

    explicit auto_ptr(X* p =0) throw() { ptr=0; }
    auto_ptr(auto_ptr& a) throw() { ptr=a.ptr; a.ptr=0; }     // note: not const auto_ptr&
    template<class Y> auto_ptr(auto_ptr<Y>& a) throw() { ptr=a.ptr; a.ptr=0; }
    auto_ptr& operator=(auto_ptr& a) throw() { ptr=a.ptr; a.ptr=0; }
    template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw() { ptr=a.ptr; a.ptr=0; }
    ~auto_ptr() throw() { delete ptr; }

    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
    X* get() const throw() { return ptr; }                    // extract pointer
    X* release() throw() { X* t = ptr; ptr=0; return t; }     // relinquish ownership
    void reset(X* p =0) throw() { if (p!=ptr) { delete ptr; ptr=p; } }

    auto_ptr(auto_ptr_ref<X>) throw();                        // copy from auto_ptr_ref
    template<class Y> operator auto_ptr_ref<Y>() throw();     // copy from auto_ptr_ref
    template<class Y> operator auto_ptr<Y>() throw();         // destructive copy from auto_ptr
};
```

The purpose of ***auto_ptr_ref*** is to implement the destructive copy semantics for ordinary ***auto_ptr***s while making it impossible to copy a ***const auto_ptr***. The template constructor and template assignment ensures that an ***auto_ptr<D>*** can be implicitly converted to a ***auto_ptr<B>*** if a ***D**** can be converted to a ***B****. For example:

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2 = pc; // now p2 is responsible for deletion
    auto_ptr<Circle> p3 = p2; // now p3 is responsible for deletion (and p2 isn't)
    p2->m = 7;                // programmer error: p2.get()==0
    Shape* ps = p3.get();     // extract the pointer from an auto_ptr
    auto_ptr<Shape> aps = p3; // transfer of ownership and convert type
    auto_ptr<Circle> p4 = p;  // programmer error: now p4 is also responsible for deletion
}
```

The effect of having more than one ***auto_ptr*** own an object is undefined; most likely the object will be deleted twice (with bad effects).

Note that ***auto_ptr***'s destructive copy semantics means that it does not meet the requirements for elements of a standard container or for standard algorithms such as ***sort()***. For example:

```
void h(vector< auto_ptr<Shape*> >& v)     // dangerous: use of auto_ptr in container
{
    sort(v.begin(),v.end());        // Don't do this: The sort will probably mess up v
}
```

Clearly, ***auto_ptr*** isn't a general smart pointer. However, it provides the service for which it was designed – exception safety for automatic pointers – with essentially no overhead.

14.4.3 Caveat [except.caveat]

Not all programs need to be resilient against all forms of failure, and not all resources are critical enough to warrant the effort to protect them using ''resource acquisition is initialization,'' *auto_ptr*, and *catch*(. . .). For example, for many programs that simply read an input and run to completion, the most suitable response to a serious run-time error is to abort the process (after producing a suitable diagnostic). That is, let the system release all acquired resources and let the user re-run the program with a more suitable input. The strategy discussed here is intended for applications for which such a simplistic response to a run-time error is unacceptable. In particular, a library designer usually cannot make assumptions about the fault tolerance requirements of a program using the library and is thus forced to avoid all unconditional run-time failures and to release all resources before a library function returns to the calling program. The ''resource acquisition is initialization'' strategy, together with the use of exceptions to signal failure, is suitable for many such libraries.

14.4.4 Exceptions and New [except.new]

Consider:

```
void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];

    X* p3 = new(buffer[10]) X;          // place X in buffer (no deallocation needed)
    X* p4 = new(buffer[11]) X[10];

    X* p5 = new(a) X;                   // allocation from Arena a (deallocate from a)
    X* p6 = new(a) X[10];
}
```

What happens if *X´s* constructor throws an exception? Is the memory allocated by the *operator new*( ) freed? For the ordinary case, the answer is yes, so the initializations of *p1* and *p2* don't cause memory leaks.

When the placement syntax (§10.4.11) is used, the answer cannot be that simple. Some uses of that syntax allocate memory, which then ought to be released; however, some don't. Furthermore, the point of using the placement syntax is to achieve nonstandard allocation, so nonstandard freeing is typically required. Consequently, the action taken depends on the allocator used. If an allocator *Z*::*operator new*( ) is used, *Z*::*operator delete*( ) is invoked if it exists; otherwise, no deallocation is attempted. Arrays are handled equivalently (§15.6.1). This strategy correctly handles the standard library placement *new* operator (§10.4.11), as well as any case in which the programmer has provided a matching pair of allocation and deallocation functions.

14.4.5 Resource Exhaustion [except.exhaust]

A recurring programming problem is what to do when an attempt to acquire a resource fails. For example, previously we blithely opened files (using *fopen*( )) and requested memory from the free store (using operator *new*) without worrying about what happened if the file wasn't there or if we

had run out of free store. When confronted with such problems, programmers come up with two styles of solutions:

Resumption: Ask some caller to fix the problem and carry on.

Termination: Abandon the computation and return to some caller.

In the former case, a caller must be prepared to help out with resource acquisition problems in unknown pieces of code. In the latter, a caller must be prepared to cope with failure of the attempt to acquire the resource. The latter is in most cases far simpler and allows a system to maintain a better separation of levels of abstraction. Note that it is not the program that terminates when one uses the termination strategy; only an individual computation terminates. ''Termination'' is the traditional term for a strategy that returns from a ''failed'' computation to an error handler associated with a caller (which may re-try the failed computation), rather than trying to repair a bad situation and resume from the point at which the problem was detected.

In C++, the resumption model is supported by the function-call mechanism and the termination model is supported by the exception-handling mechanism. Both can be illustrated by a simple implementation and use of the standard library ***operator new*( )**:

```
void* operator new(size_t size)
{
    for (;;) {
        if (void* p = malloc(size)) return p;          // try to find memory
        if (_new_handler == 0) throw bad_alloc();      // no handler: give up
        _new_handler();                                // ask for help
    }
}
```

Here, I use the standard C library ***malloc*( )** to do the real search for memory; other implementations of ***operator new*( )** may choose other ways. If memory is found, ***operator new*( )** can return a pointer to it. Otherwise, ***operator new*( )** calls the ***_new_handler***. If the ***_new_handler*** can find more memory for ***malloc*( )** to allocate, all is fine. If it can't, the handler cannot return to ***operator new*( )** without causing an infinite loop. The ***_new_handler*( )** might then choose to throw an exception, thus leaving the mess for some caller to handle:

```
void my_new_handler()
{
    int no_of_bytes_found = find_some_memory();
    if (no_of_bytes_found < min_allocation) throw bad_alloc();      // give up
}
```

Somewhere, there ought to be a *try_block* with a suitable handler:

```
try {
    // ...
}
catch (bad_alloc) {
    // somehow respond to memory exhaustion
}
```

The ***_new_handler*** used in the implementation of ***operator new*( )** is a pointer to a function maintained by the standard function ***set_new_handler*( )**. If I want ***my_new_handler*( )** to be used as

the *_new_handler*, I say:

```
set_new_handler(&my_new_handler);
```

If I also want to catch *bad_alloc*, I might say:

```
void f()
{
    void(*oldnh)() = set_new_handler(&my_new_handler);

    try {
        // ...
    }
    catch (bad_alloc) {
        // ...
    }
    catch (...) {
        set_new_handler(oldnh);   // re-set handler
        throw;     // re-throw
    }

    set_new_handler(oldnh);       // re-set handler
}
```

Even better, avoid the *catch*(...) handler by applying the ''resource acquisition is initialization'' technique described in §14.4 to the *_new_handler* (§14.12[1]).

With the *_new_handler*, no extra information is passed along from where the error is detected to the helper function. It is easy to pass more information. However, the more information that is passed between the code detecting a run-time error and a function helping correct that error, the more the two pieces of code become dependent on each other. This implies that changes to the one piece of code require understanding of and maybe even changes to the other. To keep separate pieces of software separate, it is usually a good idea to minimize such dependencies. The exception-handling mechanism supports such separation better than do function calls to helper routines provided by a caller.

In general, it is wise to organize resource allocation in layers (levels of abstraction) and avoid having one layer depend on help from the layer that called it. Experience with larger systems shows that successful systems evolve in this direction.

Throwing an exception requires an object to throw. A C++ implementation is required to have enough spare memory to be able to throw *bad_alloc* in case of memory exhaustion. However, it is possible that throwing some other exception will cause memory exhaustion.

14.4.6 Exceptions in Constructors [except.ctor]

Exceptions provide a solution to the problem of how to report errors from a constructor. Because a constructor does not return a separate value for a caller to test, the traditional (that is, non-exception-handling) alternatives are:

[1] Return an object in a bad state, and trust the user to test the state.

[2] Set a nonlocal variable (e.g., *errno*) to indicate that the creation failed, and trust the user to test that variable.

[3] Don't do any initialization in the constructor, and rely on the user to call an initialization function before the first use.

[4] Mark the object ''uninitialized'' and have the first member function called for the object do the real initialization, and that function can then report an error if initialization fails.

Exception handling allows the information that a construction failed to be transmitted out of the constructor. For example, a simple *Vector* class might protect itself from excessive demands on memory like this:

```
class Vector {
public:
        class Size { };

        enum { max = 32000 };

        Vector::Vector(int sz)
        {
                if (sz<0 || max<sz) throw Size();
                // ...
        }
        // ...
};
```

Code creating *Vector*s can now catch *Vector::Size* errors, and we can try to do something sensible with them:

```
Vector* f(int i)
{
        try {
                Vector* p = new Vector(i);
                // ...
                return p;
        }
        catch(Vector::Size) {
                // deal with size error
        }
}
```

As always, the error handler itself can use the standard set of fundamental techniques for error reporting and recovery. Each time an exception is passed along to a caller, the view of what went wrong changes. If suitable information is passed along in the exception, the amount of information available to deal with the problem could increase. In other words, the fundamental aim of the error-handling techniques is to pass information about an error from the original point of detection to a point where there is sufficient information available to recover from the problem, and to do so reliably and conveniently.

The ''resource acquisition is initialization'' technique is the safest and most elegant way of handling constructors that acquire more than one resource (§14.4). In essence, the technique reduces the problem of handling many resources to repeated application of the (simple) technique for handling one resource.

14.4.6.1 Exceptions and Member Initialization [except.member]

What happens if a member initializer (directly or indirectly) throws an exception? By default, the exception is passed on to whatever invoked the constructor for the member's class. However, the constructor itself can catch such exceptions by enclosing the complete function body – including the member initializer list – in a *try-block*. For example:

```
class X {
        Vector v;
        // ...
public:
        X(int);
        // ...
};

X::X(int s)
try
        :v(s)        // initialize v by s
{
        // ...
}
catch (Vector::Size) { // exceptions thrown for v are caught here
        // ...
}
```

Copy constructors (§10.4.4.1) are special in that they are invoked implicitly and because they often both acquire and release resources. In particular, the standard library assumes proper – non-exception-throwing – behavior of copy constructors. For these reasons, care should be taken that a copy constructor throws an exception only in truly disastrous circumstances. Complete recovery from an exception in a copy constructor is unlikely to be feasible in every context of its use. To be even potentially safe, a copy constructor must leave behind two objects, each of which fulfills the invariant of its class (§24.3.7.1).

Naturally, copy assignment operators should be treated with as much care as copy constructors.

14.4.7 Exceptions in Destructors [except.dtor]

From the point of view of exception handling, a destructor can be called in one of two ways:

[1] *Normal call*: As the result of a normal exit from a scope (§10.4.3), a ***delete*** (§10.4.5), etc.

[2] *Call during exception handling*: During stack unwinding (§14.4), the exception-handling mechanism exits a scope containing an object with a destructor.

In the latter case, an exception may not escape from the destructor itself. If it does, it is considered a failure of the exception-handling mechanism and ***std::terminate()*** (§14.7) is called. After all, there is no general way for the exception-handling mechanism or the destructor to determine whether it is acceptable to ignore one of the exceptions in favor of handling the other.

If a destructor calls functions that may throw exceptions, it can protect itself. For example:

```
X::~X()
try {
        f(); // might throw
}
catch (...) {
        // do something
}
```

The standard library function ***uncaught_exception*()** returns ***true*** if an exception has been thrown but hasn't yet been caught. This allows the programmer to specify different actions in a destructor depending on whether an object is destroyed normally or as part of stack unwinding.

14.5 Exceptions That Are Not Errors [except.not.error]

If an exception is expected and caught so that it has no bad effects on the behavior of the program, then how can it be an error? Only because the programmer thinks of it as an error and of the exception-handling mechanisms as tools for handling errors. Alternatively, one might think of the exception-handling mechanisms as simply another control structure. For example:

```
void f(Queue<X>& q)
{
        try {
                for (;;) {
                        X m = q.get();          // throws 'Empty' if queue is empty
                        // ...
                }
        }
        catch (Queue<X>::Empty) {
                return;
        }
}
```

This actually has some charm, so it is a case in which it is not entirely clear what should be considered an error and what should not.

Exception handling is a less structured mechanism than local control structures such as ***if*** and ***for*** and is often less efficient when an exception is actually thrown. Therefore, exceptions should be used only where the more traditional control structures are inelegant or impossible to use. Note that the standard library offers a ***queue*** of arbitrary elements without using exceptions (§17.3.2).

Using exceptions as alternate returns can be an elegant technique for terminating search functions – especially highly recursive search functions such as a lookup in a tree. For example:

```
void fnd(Tree* p, const string& s)
{
        if (s == p->str) throw p;          // found s
        if (p->left) fnd(p->left, s);
        if (p->right) fnd(p->right, s);
}
```

```
Tree* find(Tree* p, const string& s)
{
    try {
        fnd(p,s);
    }
    catch (Tree* q) {    // q->str==s
        return q;
    }
    return 0;
}
```

However, such use of exceptions can easily be overused and lead to obscure code. Whenever reasonable, one should stick to the ''exception handling is error handling'' view. When this is done, code is clearly separated into two categories: ordinary code and error-handling code. This makes code more comprehensible. Unfortunately, the real world isn't so clear cut. Program organization will (and to some extent should) reflect that.

Error handling is inherently difficult. Anything that helps preserve a clear model of what is an error and how it is handled should be treasured.

14.6 Exception Specifications [except.spec]

Throwing or catching an exception affects the way a function relates to other functions. It can therefore be worthwhile to specify the set of exceptions that might be thrown as part of the function declaration. For example:

```
void f(int a) throw (x2, x3);
```

This specifies that *f*( ) may throw only exceptions *x2*, *x3*, and exceptions derived from these types, but no others. When a function specifies what exceptions it might throw, it effectively offers a guarantee to its callers. If during execution that function does something that tries to abrogate the guarantee, the attempt will be transformed into a call of *std*::*unexpected*( ). The default meaning of *unexpected*( ) is *std*::*terminate*( ), which in turn normally calls *abort*( ); see §9.4.1.1 for details.

In effect,

```
void f() throw (x2, x3)
{
    // stuff
}
```

is equivalent to:

```
void f()
try
{
    // stuff
}
```

```
catch (x2) { throw; }        // re-throw
catch (x3) { throw; }        // re-throw
catch ( . . . ) {
        std::unexpected();   // unexpected() will not return
}
```

The most important advantage is that the function *declaration* belongs to an interface that is visible to its callers. Function *definitions*, on the other hand, are not universally available. Even when we do have access to the source code of all our libraries, we strongly prefer not to have to look at it very often. In addition, a function with an *exception-specification* is shorter and clearer than the equivalent hand-written version.

A function declared without an *exception-specification* is assumed to throw every exception. For example:

```
int f();            // can throw any exception
```

A function that will throw no exceptions can be declared with an empty list:

```
int g() throw ();     // no exception thrown
```

One might think that the default should be that a function throws no exceptions. However, that would require exception specifications for essentially every function, would be a significant cause for recompilation, and would inhibit cooperation with software written in other languages. This would encourage programmers to subvert the exception-handling mechanisms and to write spurious code to suppress exceptions. It would provide a false sense of security to people who failed to notice the subversion.

14.6.1 Checking Exception Specifications [except.check.spec]

It is not possible to catch every violation of an interface specification at compile time. However, much compile-time checking is done. The way to think about *exception-specification*s is to assume that a function *will* throw any exception it can. The rules for compile-time checking *exception-specification*s outlaw easily detected absurdities.

If any declaration of a function has an *exception-specification*, every declaration of that function (including the definition) must have an *exception-specification* with exactly the same set of exception types. For example:

```
int f() throw (std::bad_alloc);

int f()     // error: exception-specification missing
{
        // ...
}
```

Importantly, *exception-specification*s are not required to be checked exactly across compilation-unit boundaries. Naturally, an implementation can check. However, for many large and long-lived systems, it is important that the implementation does not – or, if it does, that it carefully gives hard errors only where violations will not be caught at run time.

The point is to ensure that adding an exception somewhere doesn't force a complete update of related exception specifications and a recompilation of all potentially affected code. A system can

then function in a partially updated state relying on the dynamic (run-time) detection of unexpected exceptions. This is essential for the maintenance of large systems in which major updates are expensive and not all source code is accessible.

A virtual function may be overridden only by a function that has an *exception-specification* at least as restrictive as its own (explicit or implicit) *exception-specification*. For example:

```
class B {
public:
    virtual void f();                    // can throw anything
    virtual void g() throw(X,Y);
    virtual void h() throw(X);
};

class D : public B {
public:
    void f() throw(X);                   // ok
    void g() throw(X);                   // ok: D::g() is more restrictive than B::g()
    void h() throw(X,Y);                 // error: D::h() is less restrictive than B::h()
};
```

This rule is really only common sense. If a derived class threw an exception that the original function didn't advertise, a caller couldn't be expected to catch it. On the other hand, an overriding function that throws fewer exceptions clearly obeys the rule set out by the overridden function's *exception-specification*.

Similarly, you can assign a pointer to function that has a more restrictive *exception-specification* to a pointer to function that has a less restrictive *exception-specification*, but not vice versa. For example:

```
void f() throw(X);
void (*pf1)() throw(X,Y) = &f;       // ok
void (*pf2)() throw() = &f;          // error: f() is less restrictive than pf2
```

In particular, you cannot assign a pointer to a function without an *exception-specification* to a pointer to function that has one:

```
void g(); // might throw anything

void (*pf3)() throw(X) = &g;         // error: g() less restrictive than pf3
```

An *exception-specification* is not part of the type of a function and a ***typedef*** may not contain one. For example:

```
typedef void (*PF)() throw(X);       // error
```

14.6.2 Unexpected Exceptions [except.unexpected]

An *exception-specification* can lead to calls to ***unexpected*()**. Such calls are typically undesirable except during testing. Such calls can be avoided through careful organization of exceptions and specification of interfaces. Alternatively, calls to ***unexpected*()** can be intercepted and rendered harmless.

A well-defined subsystem Y will often have all its exceptions derived from a class ***Yerr***. For example, given

```
class Some_Yerr : public Yerr { /* ... */ };
```

a function declared

```
void f() throw (Xerr, Yerr, exception);
```

will pass any ***Yerr*** on to its caller. In particular, ***f*( )** would handle a ***Some_Yerr*** by passing it on to its caller. Thus, no ***Yerr*** in ***f*( )** will trigger ***unexpected*( )**.

All exceptions thrown by the standard library are derived from class ***exception*** (§14.10).

14.6.3 Mapping Exceptions [except.mapping]

Occasionally, the policy of terminating a program upon encountering an unexpected exception is too Draconian. In such cases, the behavior of ***unexpected*( )** must be modified into something acceptable.

The simplest way of achieving that is to add the standard library exception ***std*::*bad_exception*** to an *exception-specification*. In that case, ***unexpected*( )** will simply throw ***bad_exception*** instead of invoking a function to try to cope. For example:

```
class X { };
class Y { };

void f() throw(X, std::bad_exception)
{
        // ...
        throw Y();      // throw ''bad'' exception
}
```

The *exception-specification* will catch the unacceptable exception ***Y*** and throw an exception of type ***bad_exception*** instead.

There is actually nothing particularly bad about ***bad_exception***; it simply provides a mechanism that is less drastic than calling ***terminate*( )**. However, it is still rather crude. In particular, information about which exception caused the problem is lost.

14.6.3.1 User Mapping of Exceptions [except.user.mapping]

Consider a function ***g*( )** written for a non-networked environment. Assume further that ***g*( )** has been declared with an *exception-specification* so that it will throw only exceptions related to its ''subsystem Y:''

```
void g() throw(Yerr);
```

Now assume that we need to call ***g*( )** in a networked environment.

Naturally, ***g*( )** will not know about network exceptions and will invoke ***unexpected*( )** when it encounters one. To use ***g*( )** in a distributed environment, we must either provide code that handles network exceptions or rewrite ***g*( )**. Assuming a rewrite is infeasible or undesirable, we can handle the problem by redefining the meaning of ***unexpected*( )**.

Memory exhaustion is dealt with by the *_new_handler* determined by *set_new_handler*( ). Similarly, the response to an unexpected exception is determined by an *_unexpected_handler* set by *std*::*set_unexpected*( ) from <*exception*>:

```
typedef void(*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
```

To handle unexpected exceptions well, we first define a class to allow us to use the ''resource acquisition is initialization'' technique for *unexpected*( ) functions:

```
class STC {      // store and reset class
        unexpected_handler old;
public:
        STC(unexpected_handler f) { old = set_unexpected(f); }
        ~STC() { set_unexpected(old); }
};
```

Then, we define a function with the meaning we want for *unexpected*( ) in this case:

```
class Yunexpected : Yerr { };

void throwY() throw(Yunexpected) { throw Yunexpected(); }
```

Used as an *unexpected*( ) function, *throwY*( ) maps any unexpected exception into *Yunexpected*.

Finally, we provide a version of *g*( ) to be used in the networked environment:

```
void networked_g() throw(Yerr)
{
        STC xx(&throwY);    // now unexpected() throws Yunexpected
        g();
}
```

Because *Yunexpected* is derived from *Yerr*, the *exception-specification* is not violated. Had *throwY*( ) thrown an exception that did violate the *exception-specification*, *terminate*( ) would have been called.

By saving and restoring the *_unexpected_handler*, we make it possible for several subsystems to control the handling of unexpected exceptions without interfering with each other. Basically, this technique for mapping an unexpected exception into an expected one is a more flexible variant of what the system offers in the form of *bad_exception*.

14.6.3.2 Recovering the Type of an Exception [except.recover]

Mapping unexpected exceptions to *Yunexpected* would allow a user of *networked_g*( ) to know that an unexpected exception had been mapped into *Yunexpected*. However, such a user wouldn't know which exception had been mapped. That information was lost in *throwY*( ). A simple technique allows that information to be recorded and passed on:

```
class Yunexpected : public Yerr {
public:
    Network_exception* pe;

    Yunexpected(Network_exception* p) :pe(p) { }
};

void throwY() throw(Yunexpected)
{
    try {
        throw; // re-throw to be caught immediately!
    }
    catch(Network_exception& p) {
        throw Yunexpected(&p);   // throw mapped exception
    }
    catch(...) {
        throw Yunexpected(0);
    }
}
```

Re-throwing an exception and catching it allows us to get a handle on any exception of a type we can name. The ***throwY*()** function is called from ***unexpected*()**, which is conceptually called from a ***catch*(...)** handler. There therefore is definitely an exception to re-throw. It is not possible for an ***unexpected*()** function to ignore the exception and return. If it tries to, ***unexpected*()** itself will throw a ***bad_exception*** (§14.6.3).

14.7 Uncaught Exceptions [except.uncaught]

If an exception is thrown but not caught, the function ***std*::*terminate*()** will be called. The ***terminate*()** function will also be called when the exception-handling mechanism finds the stack corrupted and when a destructor called during stack unwinding caused by an exception tries to exit using an exception.

An unexpected exception is dealt with by the ***_unexpected_handler*** determined by ***set_unexpected*()**. Similarly, the response to an uncaught exception is determined by an ***_uncaught_handler*** set by ***std*::*set_terminate*()** from ***<exception>***:

```
typedef void(*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

The return value is the previous function given to ***set_terminate*()**.

The reason for ***terminate*()** is that exception handling must occasionally be abandoned for less subtle error-handling techniques. For example, ***terminate*()** could be used to abort a process or maybe to re-initialize a system. The intent is for ***terminate*()** to be a drastic measure to be applied when the error-recovery strategy implemented by the exception-handling mechanism has failed and it is time to go to another level of a fault tolerance strategy.

By default, ***terminate*()** will call ***abort*()** (§9.4.1.1). This default is the correct choice for most users – especially during debugging.

An *_uncaught_handler* is assumed not to return to its caller. If it tries to, *terminate*( ) will call *abort*( ).

Note that *abort*( ) indicates abnormal exit from the program. The function *exit*( ) can be used to exit a program with a return value that indicates to the surrounding system whether the exit is normal or abnormal (§9.4.1.1).

It is implementation-defined whether destructors are invoked when a program is terminated because of an uncaught exception. On some systems, it is essential that the destructors are not called so that the program can be resumed from the debugger. On other systems, it is architecturally close to impossible *not* to invoke the destructors while searching for a handler.

If you want to ensure cleanup when an uncaught exception happens, you can add a catch-all handler (§14.3.2) to *main*( ) in addition to handlers for exceptions you really care about. For example:

```
int main()
try {
    // ...
}
catch (std::range_error)
{
    cerr << "range error: Not again!\n";
}
catch (std::bad_alloc)
{
    cerr << "new ran out of memory\n";
}
catch (...) {
    // ...
}
```

This will catch every exception, except those thrown by construction and destruction of global variables. There is no way of catching exceptions thrown during initialization of global variables. The only way of gaining control in case of *throw* from an initializer of a nonlocal static object is *set_unexpected*( ) (§14.6.2). This is another reason to avoid global variables whenever possible.

When an exception is caught, the exact point where it was thrown is generally not known. This represents a loss of information compared to what a debugger might know about the state of a program. In some C++ development environments, for some programs, and for some people, it might therefore be preferable *not* to catch exceptions from which the program isn't designed to recover.

14.8 Exceptions and Efficiency [except.efficiency]

In principle, exception handling can be implemented so that there is no run-time overhead when no exception is thrown. In addition, this can be done so that throwing an exception isn't all that expensive compared to calling a function. Doing so without adding significant memory overhead while maintaining compatibility with C calling sequences, debugger conventions, etc., is possible, but hard. However, please remember that the alternatives to exceptions are not free either. It is not unusual to find traditional systems in which half of the code is devoted to error handling.

Consider a simple function *f*( ) that appears to have nothing to do with exception handling:

```
void g(int);

void f()
{
    string s;
    // ...
    g(1);
    g(2);
}
```

However, *g*( ) may throw an exception, so *f*( ) must contain code ensuring that *s* is destroyed correctly in case of an exception. However, had *g*( ) not thrown an exception it would have had to report its error some other way. Consequently, the comparable code using ordinary code to handle errors instead of exceptions isn't the plain code above, but something like:

```
bool g(int);

bool f()
{
    string s;
    // ...
    if (g(1))
        if (g(2))
            return true;
        else
            return false;
    else
        return false;
}
```

People don't usually handle errors this systematically, though, and it is not always critical to do so. However, when careful and systematic handling of errors is necessary, such housekeeping is best left to a computer, that is, to the exception-handling mechanisms.

Exception-specifications (§14.6) can be most helpful in improving generated code. Had we stated that *g*( ) didn't throw an exception:

```
void g(int) throw();
```

the code generation for *f*( ) could have been improved. It is worth observing that no traditional C function throws an exception, so in most programs every C function can be declared with the empty throw specification *throw*( ). In particular, an implementation knows that only a few standard C library functions (such as *atexit*( ) and *qsort*( )) can throw exceptions, and it can take advantage of that fact to generate better code.

Before giving a ''C function'' an empty *exception-specification*, *throw*( ), take a minute to consider if it could possibly throw an exception. For example, it might have been converted to use the C++ operator *new*, which can throw *bad_alloc*, or it might call a C++ library that throws an exception.

14.9 Error-Handling Alternatives [except.alternatives]

The purpose of the exception-handling mechanisms is to provide a means for one part of a program to inform another part of a program that an ''exceptional circumstance'' has been detected. The assumption is that the two parts of the program are written independently and that the part of the program that handles the exception often can do something sensible about the error.

To use handlers effectively in a program, we need an overall strategy. That is, the various parts of the program must agree on how exceptions are used and where errors are dealt with. The exception-handling mechanisms are inherently nonlocal, so adherence to an overall strategy is essential. This implies that the error-handling strategy is best considered in the earliest phases of a design. It also implies that the strategy must be simple (relative to the complexity of the total program) and explicit. Something complicated would not be consistently adhered to in an area as inherently tricky as error recovery.

First of all, the idea that a single mechanism or technique can handle all errors must be dispelled; it would lead to complexity. Successful fault-tolerant systems are multilevel. Each level copes with as many errors as it can without getting too contorted and leaves the rest to higher levels. The notion of ***terminate*( )** is intended to support this view by providing an escape if the exception-handling mechanism itself is corrupted or if it has been incompletely used, thus leaving exceptions uncaught. Similarly, the notion of ***unexpected*( )** is intended to provide an escape when the strategy using *exception-specification*s to provide firewalls fails.

Not every function should be a firewall. In most systems, it is not feasible to write every function to do sufficient checking to ensure that it either completes successfully or fails in a well-defined manner. The reasons that this will not work varies from program to program and from programmer to programmer. However, for larger programs:

[1] The amount of work needed to ensure this notion of ''reliability'' is too great to be done consistently.

[2] The overheads in time and space are too great for the system to run acceptably (there will be a tendency to check for the same errors, such as invalid arguments, over and over again).

[3] Functions written in other languages won't obey the rules.

[4] This purely local notion of ''reliability'' leads to complexities that actually become a burden to overall system reliability.

However, separating the program into distinct subsystems that either complete successfully or fail in well-defined ways is essential, feasible, and economical. Thus, a major library, subsystem, or key function should be designed in this way. Exception specifications are intended for interfaces to such libraries and subsystems.

Usually, we don't have the luxury of designing all of the code of a system from scratch. Therefore, to impose a general error-handling strategy on all parts of a program, we must take into account program fragments implemented using strategies different from ours. To do this we must address a variety of concerns relating to the way a program fragment manages resources and the state in which it leaves the system after an error. The aim is to have the program fragment appear to follow the general error-handling strategy even if it internally follows a different strategy.

Occasionally, it is necessary to convert from one style of error reporting to another. For example, we might check ***errno*** and possibly throw an exception after a call to a C library or, conversely, catch an exception and set ***errno*** before returning to a C program from a C++ library:

```
void callC() throw(C_blewit)
{
    errno = 0;
    c_function();
    if (errno) {
        // cleanup, if possible and necessary
        throw C_blewit(errno);
    }
}

extern "C" void call_from_C() throw()
{
    try {
        c_plus_plus_function();
    }
    catch (...) {
        // cleanup, if possible and necessary
        errno = E_CPLPLFCTBLEWIT;
    }
}
```

In such cases, it is important to be systematic enough to ensure that the conversion of error reporting styles is complete.

Error handling should be – as far as possible – hierarchical. If a function detects a run-time error, it should not ask its caller for help with recovery or resource acquisition. Such requests set up cycles in the system dependencies. That in turn makes the program hard to understand and introduces the possibility of infinite loops in the error-handling and recovery code.

Simplifying techniques such as ‘‘resource acquisition is initialization’’ and simplifying assumptions such as ‘‘exceptions represent errors’’ should be used to make the error-handling code more regular. See also §24.3.7.1 for ideas about how to use invariants and assertions to make the triggering of exceptions more regular.

14.10 Standard Exceptions [except.std]

Here is a table of standard exceptions and the functions, operators, and general facilities that throw them:

Standard Exceptions (thrown by the language)			
Name	Thrown by	Reference	Header
bad_alloc	*new*	§6.2.6.2, §19.4.5	<new>
bad_cast	*dynamic_cast*	§15.4.1.1	<typeinfo>
bad_typeid	*typeid*	§15.4.4	<typeinfo>
bad_exception	*exception specification*	§14.6.3	<exception>

Standard Exceptions (thrown by the standard library)			
Name	Thrown by	Reference	Header
out_of_range	*at()*	§3.7.2, §16.3.3, §20.3.3	<stdexcept>
	bitset<>::operator[]()	§17.5.3	<stdexcept>
invalid_argument	*bitset constructor*	§17.5.3.1	<stdexcept>
overflow_error	*bitset<>::to_ulong()*	§17.5.3.3	<stdexcept>
ios_base::failure	*ios_base::clear()*	§21.3.6	<ios>

The library exceptions are part of a class hierarchy rooted in the standard library exception class ***exception*** presented in ***<exception>***:

```
class exception {
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();

    virtual const char* what() const throw();
private:
    // ...
};
```

The hierarchy looks like this:

This seems rather elaborate for organizing the eight standard exceptions. This hierarchy attempts to provide a framework for exceptions beyond the ones defined by the standard library. Logic errors are errors that in principle could be caught either before the program starts executing or by tests of arguments to functions and constructors. Run-time errors are all other errors. Some people view this as a useful framework for all errors and exceptions; I don't.

The standard library exception classes don't add functions to the set provided by ***exception***; they simply define the required virtual functions appropriately. Thus, we can write:

```
void f()
try {
    // use standard library
}
```

```
catch (exception& e) {
        cout << "standard library exception " << e.what() << '\n';     // well, maybe
        // ...
}
catch (...) {
        cout << "other exception\n";
        // ...
}
```

The standard exceptions are derived from ***exception***. However, not every exception is, so it would be a mistake to try to catch every exception by catching ***exception***. Similarly, it would be a mistake to assume that every exception derived from ***exception*** is a standard library exception: programmers can add their own exceptions to the ***exception*** hierarchy .

Note that ***exception*** operations do not themselves throw exceptions. In particular, this implies that throwing a standard library exception doesn't cause a ***bad_alloc*** exception. The exception-handling mechanism keeps a bit of memory to itself for holding exceptions (possibly on the stack). Naturally, it is possible to write code that eventually consumes all memory in the system, thus forcing a failure.

Here is a function that – if called – tests whether the function call or the exception-handling mechanism runs out of memory first:

```
void perverted()
{
        try {
                throw exception();    // recursive exception throw
        }
        catch (exception& e) {
                perverted();          // recursive function call
                cout << e.what();
        }
}
```

The purpose of the output statement is simply to prevent the compiler from re-using the memory occupied by the exception named ***e***.

14.11 Advice [except.advice]

[1] Use exceptions for error handling; §14.1, §14.5, §14.9.
[2] Don't use exceptions where more local control structures will suffice; §14.1.
[3] Use the ''resource allocation is initialization'' technique to manage resources; §14.4.
[4] Not every program needs to be exception safe; §14.4.3.
[5] Use ''resource allocation is initialization'' and exception handlers to maintain invariants; §14.3.2.
[6] Minimize the use of *try-block*s. Use ''resource acquisition is initialization'' instead of explicit handler code; §14.4.
[7] Not every function needs to handle every possible error; §14.9.

[8] Throw an exception to indicate failure in a constructor; §14.4.6.
[9] Avoid throwing exceptions from copy constructors; §14.4.6.1.
[10] Avoid throwing exceptions from destructors; §14.4.7.
[11] Have ***main***( ) catch and report all exceptions; §14.7.
[12] Keep ordinary code and error-handling code separate; §14.4.5, §14.5.
[13] Be sure that every resource acquired in a constructor is released when throwing an exception in that constructor; §14.4.
[14] Keep resource management hierarchical; §14.4.
[15] Use *exception-specification*s for major interfaces; §14.9.
[16] Beware of memory leaks caused by memory allocated by ***new*** not being released in case of an exception; §14.4.1, §14.4.2, §14.4.4.
[17] Assume that every exception that can be thrown by a function will be thrown; §14.6.
[18] Don't assume that every exception is derived from class ***exception***; §14.10.
[19] A library shouldn't unilaterally terminate a program. Instead, throw an exception and let a caller decide; §14.1.
[20] A library shouldn't produce diagnostic output aimed at an end user. Instead, throw an exception and let a caller decide; §14.1.
[21] Develop an error-handling strategy early in a design; §14.9.

14.12 Exercises [except.exercises]

1. (∗2) Generalize the ***STC*** class (§14.6.3.1) to a template that can use the ''resource acquisition is initialization'' technique to store and reset functions of a variety of types.
2. (∗3) Complete the ***Ptr_to_T*** class from §11.11 as a template that uses exceptions to signal run-time errors.
3. (∗3) Write a function that searches a binary tree of nodes based on a ***char***** field for a match. If a node containing ***hello*** is found, ***find***("***hello***") will return a pointer to that node. Use an exception to indicate ''not found.''
4. (∗3) Define a class ***Int*** that acts exactly like the built-in type ***int***, except that it throws exceptions rather than overflowing or underflowing.
5. (∗2.5) Take the basic operations for opening, closing, reading, and writing from the C interface to your operating system and provide equivalent C++ functions that call the C functions but throw exceptions in case of errors.
6. (∗2.5) Write a complete ***Vector*** template with ***Range*** and ***Size*** exceptions.
7. (∗1) Write a loop that computes the sum of a ***Vector*** as defined in §14.12[6] without examining the size of the ***Vector***. Why is this a bad idea?
8. (∗2.5) Consider using a class ***Exception*** as the base of all classes used as exceptions. What should it look like? How should it be used? What good might it do? What disadvantages might result from a requirement to use such a class?
9. (∗1) Given a

```
int main() { /* ... */ }
```

change it so that it catches all exceptions, turns them into error messages, and ***abort***( )s. Hint:

call_from_C( ) in §14.9 doesn't quite handle all cases.

10. (∗2) Write a class or template suitable for implementing callbacks.
11. (∗2.5) Write a ***Lock*** class for some system supporting concurrency.

15

Class Hierarchies

Abstraction is selective ignorance.
– Andrew Koenig

Multiple inheritance — ambiguity resolution — inheritance and *using-declarations* — replicated base classes — virtual base classes — uses of multiple inheritance — access control — protected — access to base classes — run-time type information — ***dynamic_cast*** — static and dynamic casts — casting from virtual bases — ***typeid*** — extended type information — uses and misuses of run-time type information — pointers to members — free store — virtual constructors — advice — exercises.

15.1 Introduction and Overview [hier.intro]

This chapter discusses how derived classes and virtual functions interact with other language facilities such as access control, name lookup, free store management, constructors, pointers, and type conversions. It has five main parts:

§15.2 Multiple Inheritance
§15.3 Access Control
§15.4 Run-time Type Identification
§15.5 Pointers to Members
§15.6 Free Store Use

In general, a class is constructed from a lattice of base classes. Because most such lattices historically have been trees, a *class lattice* is often called a *class hierarchy*. We try to design classes so that users need not be unduly concerned about the way a class is composed out of other classes. In particular, the virtual call mechanism ensures that when we call a function *f*( ) on an object, the same function is called whichever class in the hierarchy provided the declaration of *f*( ) used for the call. This chapter focuses on ways to compose class lattices and to control access to parts of classes and on facilities for navigating class lattices at compile time and run time.

15.2 Multiple Inheritance [hier.mi]

As shown in §2.5.4 and §12.3, a class can have more than one direct base class, that is, more than one class specified after the : in the class declaration. Consider a simulation in which concurrent activities are represented by a class ***Task*** and data gathering and display is achieved through a class ***Displayed***. We can then define a class of simulated entities, class ***Satellite***:

```
class Satellite : public Task, public Displayed {
    // ...
};
```

The use of more than one immediate base class is usually called *multiple inheritance*. In contrast, having just one direct base class is called *single inheritance*.

In addition to whatever operations are defined specifically for a ***Satellite***, the union of operations on ***Task***s and ***Displayed***s can be applied. For example:

```
void f(Satellite& s)
{
    s.draw();        // Displayed::draw()
    s.delay(10);     // Task::delay()
    s.transmit();    // Satellite::transmit()
}
```

Similarly, a ***Satellite*** can be passed to functions that expect a ***Task*** or a ***Displayed***. For example:

```
void highlight(Displayed*);
void suspend(Task*);

void g(Satellite* p)
{
    highlight(p);    // pass a pointer to the Displayed part of the Satellite
    suspend(p);      // pass a pointer to the Task part of the Satellite
}
```

The implementation of this clearly involves some (simple) compiler technique to ensure that functions expecting a ***Task*** see a different part of a ***Satellite*** than do functions expecting a ***Displayed***. Virtual functions work as usual. For example:

```
class Task {
    // ...
    virtual void pending() = 0;
};

class Displayed {
    // ...
    virtual void draw() = 0;
};
```

```
class Satellite : public Task, public Displayed {
    // ...
    void pending();        // override Task::pending()
    void draw();           // override Displayed::draw()
};
```

This ensures that ***Satellite*::*draw*()** and ***Satellite*::*pending*()** will be called for a ***Satellite*** treated as a ***Displayed*** and a ***Task***, respectively.

Note that with single inheritance (only), the programmer's choices for implementing the classes ***Displayed***, ***Task***, and ***Satellite*** would be limited. A ***Satellite*** could be a ***Task*** or a ***Displayed***, but not both (unless ***Task*** was derived from ***Displayed*** or vice versa). Either alternative involves a loss of flexibility.

Why would anyone want a class ***Satellite***? Contrary to some people's conjectures, the ***Satellite*** example is real. There really was – and maybe there still is – a program constructed along the lines used to describe multiple inheritance here. It was used to study the design of communication systems involving satellites, ground stations, etc. Given such a simulation, we can answer questions about traffic flow, determine proper responses to a ground station that is being blocked by a rainstorm, consider tradeoffs between satellite connections and Earth-bound connections, etc. Such simulations do involve a variety of display and debugging operations. Also, we do need to store the state of objects such as ***Satellite***s and their subcomponents for analysis, debugging, and error recovery.

15.2.1 Ambiguity Resolution [hier.ambig]

Two base classes may have member functions with the same name. For example:

```
class Task {
    // ...
    virtual debug_info* get_debug();
};

class Displayed {
    // ...
    virtual debug_info* get_debug();
};
```

When a ***Satellite*** is used, these functions must be disambiguated:

```
void f(Satellite* sp)
{
    debug_info* dip = sp->get_debug();  // error: ambiguous
    dip = sp->Task::get_debug();        // ok
    dip = sp->Displayed::get_debug();   // ok
}
```

However, explicit disambiguation is messy, so it is usually best to resolve such problems by defining a new function in the derived class:

```
class Satellite : public Task, public Displayed {
    // ...

    debug_info* get_debug() // override Task::get_debug() and Displayed::get_debug()
    {
        debug_info* dip1 = Task::get_debug();
        debug_info* dip2 = Displayed::get_debug();
        return dip1->merge(dip2);
    }
};
```

This localizes the information about ***Satellite***'s base classes. Because ***Satellite*****::*****get_debug*****()** overrides the ***get_debug*****()** functions from both of its base classes, ***Satellite*****::*****get_debug*****()** is called wherever ***get_debug*****()** is called for a ***Satellite*** object.

A qualified name ***Telstar*****::*****draw*** can refer to a ***draw*** declared either in ***Telstar*** or in one of its base classes. For example:

```
class Telstar : public Satellite {
    // ...
    void draw()
    {
        draw();                          // oops!: recursive call
        Satellite::draw();               // finds Displayed::draw
        Displayed::draw();
        Satellite::Displayed::draw();    // redundant double qualification
    }
};
```

In other words, if a ***Satellite*****::*****draw*** doesn't resolve to a ***draw*** declared in ***Satellite***, the compiler recursively looks in its base classes; that is, it looks for ***Task*****::*****draw*** and ***Displayed*****::*****draw***. If exactly one match is found, that name will be used. Otherwise, ***Satellite*****::*****draw*** is either not found or is ambiguous.

15.2.2 Inheritance and Using-Declarations [hier.using]

Overload resolution is not applied across different class scopes (§7.4). In particular, ambiguities between functions from different base classes are not resolved based on argument types.

When combining essentially unrelated classes, such as ***Task*** and ***Displayed*** in the ***Satellite*** example, similarity in naming typically does not indicate a common purpose. When such name clashes occur, they often come as quite a surprise to the programmer. For example:

```
class Task {
    // ...
    void debug(double p);    // print info only if priority is lower than p
};
```

```
class Displayed {
    // ...
    void debug(int v);   // the higher the 'v,' the more debug information is printed
};

class Satellite : public Task, public Displayed {
    // ...
};

void g(Satellite* p)
{
    p->debug(1);          // error: ambiguous. Displayed::debug(int) or Task::debug(double) ?
    p->Task::debug(1);        // ok
    p->Displayed::debug(1); // ok
}
```

What if the use of the same name in different base classes was the result of a deliberate design decision and the user wanted selection based on the argument types? In that case, a *using-declaration* (§8.2.2) can bring the functions into a common scope. For example:

```
class A {
public:
    int f(int);
    char f(char);
    // ...
};

class B {
public:
    double f(double);
    // ...
};

class AB: public A, public B {
public:
    using A::f;
    using B::f;
    char f(char);   // hides A::f(char)
    AB f(AB);
};

void g(AB& ab)
{
    ab.f(1);          // A::f(int)
    ab.f('a');        // AB::f(char)
    ab.f(2.0);        // B::f(double)
    ab.f(ab);         // AB::f(AB)
}
```

Using-declarations allow a programmer to compose a set of overloaded functions from base classes and the derived class. Functions declared in the derived class hide functions that would otherwise be available from a base. Virtual functions from bases can be overridden as ever (§15.2.3.1).

A *using-declaration* (§8.2.2) in a class definition must refer to members of a base class. A *using-declaration* may not be used for a member of a class from outside that class, its derived classes, and their member functions. A *using-directive* (§8.2.3) may not appear in a class definition and may not be used for a class.

A *using-declaration* cannot be used to gain access to additional information. It is simply a mechanism for making accessible information more convenient to use (§15.3.2.2).

15.2.3 Replicated Base Classes [hier.replicated]

With the ability of specifying more than one base class comes the possibility of having a class as a base twice. For example, had ***Task*** and ***Displayed*** each been derived from a ***Link*** class, a ***Satellite*** would have two ***Link***s:

```
struct Link {
        Link* next;
};

class Task : public Link {
        // the Link is used to maintain a list of all Tasks (the scheduler list)
        // ...
};

class Displayed : public Link {
        // the Link is used to maintain a list of all Displayed objects (the display list)
        // ...
};
```

This causes no problems. Two separate ***Link*** objects are used to represent the links, and the two lists do not interfere with each other. Naturally, one cannot refer to members of the ***Link*** class without risking an ambiguity (§15.2.3.1). A ***Satellite*** object could be drawn like this:

Examples of where the common base class shouldn't be represented by two separate objects can be handled using a virtual base class (§15.2.4).

Usually, a base class that is replicated the way ***Link*** is here is an implementation detail that shouldn't be used from outside its immediate derived class. If such a base must be referred to from a point where more than one copy of the base is visible, the reference must be explicitly qualified to resolve the ambiguity. For example:

```
void mess_with_links(Satellite* p)
{
        p->next = 0;                    // error: ambiguous (which Link?)
        p->Link::next = 0;              // error: ambiguous (which Link?)
```

```
        p->Task::Link::next = 0;          // ok
        p->Displayed::Link::next = 0;  // ok
        // ...
    }
```

This is exactly the mechanism used to resolve ambiguous references to members (§15.2.1).

15.2.3.1 Overriding [hier.override]

A virtual function of a replicated base class can be overridden by a (single) function in a derived class. For example, one might represent the ability of an object to read itself from a file and write itself back to a file like this:

```
class Storable {
public:
        virtual const char* get_file() = 0;
        virtual void read() = 0;
        virtual void write() = 0;
        virtual ~Storable() { write(); } // to be called from overriding destructors (see §15.2.4.1)
};
```

Naturally, several programmers might rely on this to develop classes that can be used independently or in combination to build more elaborate classes. For example, one way of stopping and restarting a simulation is to store components of a simulation and then restore them later. That idea might be implemented like this:

```
class Transmitter : public Storable {
public:
        void write();
        // ...
};

class Receiver : public Storable {
public:
        void write();
        // ...
};

class Radio : public Transmitter, public Receiver {
public:
        const char* get_file();
        void read();
        void write();
        // ...
};
```

Typically, an overriding function calls its base class versions and then does the work specific to the derived class:

```
void Radio::write()
{
    Transmitter::write();
    Receiver::write();
    // write radio-specific information
}
```

Casting from a replicated base class to a derived class is discussed in §15.4.2. For a technique for overriding each of the ***write()*** functions with separate functions from derived classes, see §25.6.

15.2.4 Virtual Base Classes [hier.vbase]

The ***Radio*** example in the previous subsection works because class ***Storable*** can be safely, conveniently, and efficiently replicated. Often, that is not the case for the kind of class that makes a good building block for other classes. For example, we might define ***Storable*** to hold the name of the file to be used for storing the object:

```
class Storable {
public:
    Storable(const char* s);
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
private:
    const char* store;

    Storable(const Storable&);
    Storable& operator=(const Storable&);
};
```

Given this apparently minor change to ***Storable***, we must must change the design of ***Radio***. All parts of an object must share a single copy of ***Storable***; otherwise, it becomes unnecessarily hard to avoid storing multiple copies of the object. One mechanism for specifying such sharing is a virtual base class. Every ***virtual*** base of a derived class is represented by the same (shared) object. For example:

```
class Transmitter : public virtual Storable {
public:
    void write();
    // ...
};

class Receiver : public virtual Storable {
public:
    void write();
    // ...
};
```

```
class Radio : public Transmitter, public Receiver {
public:
    void write();
    // ...
};
```

Or graphically:

Compare this diagram with the drawing of the ***Satellite*** object in §15.2.3 to see the difference between ordinary inheritance and virtual inheritance. In an inheritance graph, every base class of a given name that is specified to be virtual will be represented by a single object of that class. On the other hand, each base class not specified ***virtual*** will have its own sub-object representing it.

15.2.4.1 Programming Virtual Bases [hier.vbase.prog]

When defining the functions for a class with a virtual base, the programmer in general cannot know whether the base will be shared with other derived classes. This can be a problem when implementing a service that requires a base class function to be called exactly once. For example, the language ensures that a constructor of a virtual base is called exactly once. The constructor of a virtual base is invoked (implicitly or explicitly) from the constructor for the complete object (the constructor for the most derived class). For example:

```
class A {  // no constructor
    // ...
};

class B {
public:
    B(); // default constructor
    // ...
};

class C {
public:
    C(int);    // no default constructor
};

class D : virtual public A, virtual public B, virtual public C
{
    D() { /* ... */ }                  // error: no default constructor for C
    D(int i) : C(i) { /* ... */ };     // ok
    // ...
};
```

The constructor for a virtual base is called before the constructors for its derived classes.

Where needed, the programmer can simulate this scheme by calling a virtual base class function only from the most derived class. For example, assume we have a basic ***Window*** class that knows how to draw its contents:

```
class Window {
    // basic stuff
    virtual void draw();
};
```

In addition, we have various ways of decorating a window and adding facilities:

```
class Window_with_border : public virtual Window {
    // border stuff
    void own_draw();    // display the border
    void draw();
};
class Window_with_menu : public virtual Window {
    // menu stuff
    void own_draw();    // display the menu
    void draw();
};
```

The ***own_draw()*** functions need not be virtual because they are meant to be called from within a virtual ***draw()*** function that ''knows'' the type of the object for which it was called.

From this, we can compose a plausible ***Clock*** class:

```
class Clock : public Window_with_border, public Window_with_menu {
    // clock stuff
    void own_draw();    // display the clock face and hands
    void draw();
};
```

Or graphically:

The ***draw()*** functions can now be written using the ***own_draw()*** functions so that a caller of any ***draw()*** gets ***Window::draw()*** invoked exactly once. This is done independently of the kind of ***Window*** for which ***draw()*** is invoked:

```
void Window_with_border::draw()
{
    Window::draw();
    own_draw();    // display the border
}

void Window_with_menu::draw()
{
    Window::draw();
    own_draw();    // display the menu
}

void Clock::draw()
{
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw();    // display the clock face and hands
}
```

Casting from a ***virtual*** base class to a derived class is discussed in §15.4.2.

15.2.5 Using Multiple Inheritance [hier.using.mi]

The simplest and most obvious use of multiple inheritance is to ‘‘glue’’ two otherwise unrelated classes together as part of the implementation of a third class. The ***Satellite*** class built out of the ***Task*** and ***Displayed*** classes in §15.2 is an example of this. This use of multiple inheritance is crude, effective, and important, but not very interesting. Basically, it saves the programmer from writing a lot of forwarding functions. This technique does not affect the overall design of a program significantly and can occasionally clash with the wish to keep implementation details hidden. However, a technique doesn’t have to be clever to be useful.

Using multiple inheritance to provide implementations for abstract classes is more fundamental in that it affects the way a program is designed. Class ***BB_ival_slider*** (§12.3) is an example:

```
class BB_ival_slider
    : public Ival_slider    // interface
    , protected BBslider    // implementation
{
    // implementation of functions required by ‘Ival_slider’ and ‘BBslider’
    // using the facilities provided by ‘BBslider’
};
```

In this example, the two base classes play logically distinct roles. One base is a public abstract class providing the interface and the other is a protected concrete class providing implementation ‘‘details.’’ These roles are reflected in both the style of the classes and in the access control provided. The use of multiple inheritance is close to essential here because the derived class needs to override virtual functions from both the interface and the implementation.

Multiple inheritance allows sibling classes to share information without introducing a dependence on a unique common base class in a program. This is the case in which the so-called

diamond-shaped inheritance occurs (for example, the ***Radio*** (§15.2.4) and ***Clock*** (§15.2.4.1)). A virtual base class, as opposed to an ordinary base class, is needed if the base class cannot be replicated.

I find that a diamond-shaped inheritance lattice is most manageable if either the virtual base class or the classes directly derived from it are abstract classes. For example, consider again the ***Ival_box*** classes from §12.4. In the end, I made all the ***Ival_box*** classes abstract to reflect their role as pure interfaces. Doing that allowed me to place all implementation details in specific implementation classes. Also, all sharing of implementation details was done in the classical hierarchy of the windows system used for the implementation.

It would make sense for the class implementing a ***Popup_ival_slider*** to share most of the implementation of the class implementing a plain ***Ival_slider***. After all, these implementation classes would share everything except the handling of prompts. However, it would then seem natural to avoid replication of ***Ival_slider*** objects within the resulting slider implementation objects. Therefore, we could make ***Ival_slider*** a virtual base:

```
class BB_ival_slider : public virtual Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public virtual Ival_slider { /* ... */ };
class BB_popup_ival_slider
        : public virtual Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

or graphically:

It is easy to imagine further interfaces derived from ***Popup_ival_slider*** and further implementation classes derived from such classes and ***BB_popup_slider***.

If we take this idea to its logical conclusion, all of the derivations from the abstract classes that constitute our application's interfaces would become virtual. This does indeed seem to be the most logical, general, and flexible approach. The reason I didn't do that was partly historical and partly because the most obvious and common techniques for implementing virtual bases impose time and space overhead that make their extensive use within a class unattractive. Should this overhead become an issue for an otherwise attractive design, note that an object representing an ***Ival_slider*** usually holds only a virtual table pointer. As noted in §15.2.4, such an abstract class holding no variable data can be replicated without ill effects. Thus, we can eliminate the virtual base in favor of ordinary ones:

```
class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
class BB_popup_ival_slider
        : public Popup_ival_slider, protected BB_ival_slider { /* ... */ };
```

or graphically:

This is most likely a viable optimization to the admittedly cleaner alternative presented previously.

15.2.5.1 Overriding Virtual Base Functions [hier.dominance]

A derived class can override a virtual function of its direct or indirect virtual base class. In particular, two different classes might override different virtual functions from the virtual base. In that way, several derived classes can contribute implementations to the interface presented by a virtual base class. For example, the ***Window*** class might have functions ***set_color***( ) and ***prompt***( ). In that case, ***Window_with_border*** might override ***set_color***( ) as part of controlling the color scheme and ***Window_with_menu*** might override ***prompt***( ) as part of its control of user interactions:

```
class Window {
    // ...
    virtual set_color(Color) = 0;    // set background color
    virtual void prompt( ) = 0;
};

class Window_with_border : public virtual Window {
    // ...
    set_color(Color);    // control background color
};

class Window_with_menu : public virtual Window {
    // ...
    void prompt( ); // control user interactions
};

class My_window : public Window_with_menu, public Window_with_border {
    // ...
};
```

What if different derived classes override the same function? This is allowed if and only if some overriding class is derived from every other class that overrides the function. That is, one function must override all others. For example, ***My_window*** could override ***prompt***( ) to improve on what ***Window_with_menu*** provides:

```
class My_window : public Window_with_menu, public Window_with_border {
    // ...
    void prompt( ); // don't leave user interactions to base
};
```

or graphically:

If two classes override a base class function, but neither overrides the other, the class hierarchy is an error. No virtual function table can be constructed because a call to that function on the complete object would have been ambiguous. For example, had ***Radio*** in §15.2.4 not declared ***write*( )**, the declarations of ***write*( )** in ***Receiver*** and ***Transmitter*** would have caused an error when defining ***Radio***. As with ***Radio***, such a conflict is resolved by adding an overriding function to the most derived class.

A class that provides some – but not all – of the implementation for a virtual base class is often called a ''mixin.''

15.3 Access Control [hier.access]

A member of a class can be ***private***, ***protected***, or ***public***:

- If it is ***private***, its name can be used only by member functions and friends of the class in which it is declared.
- If it is ***protected***, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class (see §11.5).
- If it is ***public***, its name can be used by any function.

This reflects the view that there are three kinds of functions accessing a class: functions implementing the class (its friends and members), functions implementing a derived class (the derived class' friends and members), and other functions. This can be presented graphically:

The access control is applied uniformly to names. What a name refers to does not affect the control of its use. This means that we can have private member functions, types, constants, etc., as well as private data members. For example, an efficient non-intrusive (§16.2.1) list class often requires data structures to keep track of elements. Such information is best kept private:

```
template<class T> class List {
private:
    struct Link { T val; Link* next; };
    struct Chunk {
        enum { chunk_size = 15 };
        Link v[chunk_size];
        Chunk* next;
    };
    class Underflow { };

    Chunk* allocated;
    Link* free;
    Link* get_free();
    Link* head;
public:
    void insert(T);
    T get();
    // ...
};

template<class T> void List<T>::insert(T val)
{
    Link* lnk = get_free();
    lnk->val = val;
    lnk->next = head;
    head = lnk;
}

template<class T> List<T>::Link* List<T>::get_free()
{
    if (free == 0) {
        // allocate a new chunk and place its Links on the free list
    }
    Link* p = free;
    free = free->next;
    return p;
}

template<class T> T List<T>::get()
{
    if (head == 0) throw Underflow();

    Link* p= head;
    head = p->next;
    p->next = free;
    free = p;
    return p->val;
}
```

The ***List<T>*** scope is entered by saying ***List<T>::*** in a member function definition. Because the return type of ***get_free()*** is mentioned before the name ***List<T>::get_free()*** is mentioned, the full name ***List<T>::Link*** must be used instead of the abbreviation ***Link<T>***.

Nonmember functions (except friends) do not have such access:

```
void would_be_meddler(List<T>* p)
{
    List<T>::Link* q = 0;                    // error: List<T>::Link is private
    // ...
    q = p->free;                             // error: List<T>::free is private
    // ...
    if (List<T>::Chunk::chunk_size > 31) {   // error: List<T>::Chunk::chunk_size is private
        // ...
    }
}
```

In a ***class***, a member is by default private; in a ***struct***, a member is by default public (§10.2.8).

15.3.1 Protected Members [hier.protected]

As an example of how to use ***protected*** members, consider the ***Window*** example from §15.2.4.1. The ***own_draw()*** functions were (deliberately) incomplete in the service they provided. They were designed as building blocks for use by derived classes (only) and are not safe or convenient for general use. The ***draw()*** operations, on the other hand, were designed for general use. This distinction can be expressed by separating the interface of the ***Window*** classes in two, the ***protected*** interface and the ***public*** interface:

```
class Window_with_border {
public:
    virtual void draw();
    // ...
protected:
    void own_draw();
    // other tool-building stuff
private:
    // representation, etc.
};
```

A derived class can access a base class' protected members only for objects of its own type:

```
class Buffer {
protected:
    char a[128];
    // ...
};

class Linked_buffer : public Buffer { /* ... */ };

class Cyclic_buffer : public Buffer {
    // ...
    void f(Linked_buffer* p) {
        a[0] = 0;        // ok: access to cyclic_buffer's own protected member
        p->a[0] = 0;     // error: access to protected member of different type
    }
};
```

This prevents subtle errors that would otherwise occur when one derived class corrupts data belonging to other derived classes.

15.3.1.1 Use of Protected Members [hier.protected.use]

The simple private/public model of data hiding serves the notion of concrete types (§10.3) well. However, when derived classes are used, there are two kinds of users of a class: derived classes and ''the general public.'' The members and friends that implement the operations on the class operate on the class objects on behalf of these users. The private/public model allows the programmer to distinguish clearly between the implementers and the general public, but it does not provide a way of catering specifically to derived classes.

Members declared ***protected*** are far more open to abuse than members declared ***private***. In particular, declaring data members protected is usually a design error. Placing significant amounts of data in a common class for all derived classes to use leaves that data open to corruption. Worse, protected data, like public data, cannot easily be restructured because there is no good way of finding every use. Thus, protected data becomes a software maintenance problem.

Fortunately, you don't have to use protected data; ***private*** is the default in classes and is usually the better choice. In my experience, there have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly.

Note that none of these objections are significant for protected member *functions*; ***protected*** is a fine way of specifying operations for use in derived classes. The ***Ival_slider*** in §12.4.2 is an example of this. Had the implementation class been ***private*** in this example, further derivation would have been infeasible.

Technical examples illustrating access to members can be found in §C.11.1.

15.3.2 Access to Base Classes [hier.base.access]

Like a member, a base class can be declared ***private***, ***protected***, or ***public***. For example:

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

Public derivation makes the derived class a subtype of its base; this is the most common form of derivation. Protected and private derivation are used to represent implementation details. Protected bases are useful in class hierarchies in which further derivation is the norm; the ***Ival_slider*** from §12.4.2 is a good example of that. Private bases are most useful when defining a class by restricting the interface to a base so that stronger guarantees can be provided. For example, ***Vec*** adds range checking to its private base ***vector*** (§3.7.1) and the ***list*** of pointers template adds type checking to its ***list<void*>*** base (§13.5).

The access specifier for a base class can be left out. In that case, the base defaults to a private base for a ***class*** and a public base for a ***struct***. For example:

```
class XX : B { /* ... */ };        // B is a private base
struct YY : B { /* ... */ };       // B is a public base
```

For readability, it is best always to use an explicit access specifier.

The access specifier for a base class controls the access to members of the base class and the conversion of pointers and references from the derived class type to the base class type. Consider a class ***D*** derived from a base class ***B***:

- If ***B*** is a ***private*** base, its public and protected members can be used only by member functions and friends of ***D***. Only friends and members of ***D*** can convert a ***D**** to a ***B****.
- If ***B*** is a ***protected*** base, its public and protected members can be used only by member functions and friends of ***D*** and by member functions and friends of classes derived from ***D***. Only friends and members of ***D*** and friends and members of classes derived from ***D*** can convert a ***D**** to a ***B****.
- If ***B*** is a ***public*** base, its public members can be used by any function. In addition, its protected members can be used by members and friends of ***D*** and members and friends of classes derived from ***D***. Any function can convert a ***D**** to a ***B****.

This basically restates the rules for member access (§15.3). We choose access for bases in the same way as for members. For example, I chose to make ***BBwindow*** a ***protected*** base of ***Ival_slider*** (§12.4.2) because ***BBwindow*** was part of the implementation of ***Ival_slider*** rather than part of its interface. However, I couldn't completely hide ***BBwindow*** by making it a private base because I wanted to be able to derive further classes from ***Ival_slider***, and those derived classes would need access to the implementation.

Technical examples illustrating access to bases can be found in §C.11.2.

15.3.2.1 Multiple Inheritance and Access Control [hier.mi.access]

If a name or a base class can be reached through multiple paths in a multiple inheritance lattice, it is accessible if it is accessible through any path. For example:

```
struct B {
    int m;
    static int sm;
    // ...
};

class D1 : public virtual B { /* ... */ } ;
class D2 : public virtual B { /* ... */ } ;
class DD : public D1, private D2 { /* ... */ };

DD* pd = new DD;
B* pb = pd;                 // ok: accessible through D1
int i1 = pd->m;             // ok: accessible through D1
```

If a single entity is reachable through several paths, we can still refer to it without ambiguity. For example:

```
class X1 : public B { /* ... */ } ;
class X2 : public B { /* ... */ } ;
class XX : public X1, public X2 { /* ... */ };

XX* pxx = new XX;
int i1 = pxx->m;      // error, ambiguous: XX::X1::B::m or XX::X2::B::m
int i2 = pxx->sm;     // ok: there is only one B::sm in an XX
```

15.3.2.2 Using-Declarations and Access Control [hier.access.using]

A *using-declaration* cannot be used to gain access to additional information. It is simply a mechanism for making accessible information more convenient to use. On the other hand, once access is available, it can be granted to other users. For example:

```
class B {
private:
    int a;
protected:
    int b;
public:
    int c;
};

class D : public B {
public:
    using B::a;     // error: B::a is private
    using B::b;     // make B::b publically available through D
};
```

When a *using-declaration* is combined with private or protected derivation, it can be used to specify interfaces to some, but not all, of the facilities usually offered by a class. For example:

```
class BB : private B {     // give access to B::b and B::c, but not B::a
    using B::b;
    using B::c;
};
```

See also §15.2.2.

15.4 Run-Time Type Information [hier.rtti]

A plausible use of the ***Ival_box***es defined in §12.4 would be to hand them to a system that controlled a screen and have that system hand objects back to the application program whenever some activity had occurred. This is how many user-interfaces work. However, a user-interface system will not know about our ***Ival_box***es. The system's interfaces will be specified in terms of the system's own classes and objects rather than our application's classes. This is necessary and proper. However, it does have the unpleasant effect that we lose information about the type of objects passed to the system and later returned to us.

Recovering the ''lost'' type of an object requires us to somehow ask the object to reveal its type. Any operation on an object requires us to have a pointer or reference of a suitable type for the object. Consequently, the most obvious and useful operation for inspecting the type of an object at run time is a type conversion operation that returns a valid pointer if the object is of the expected type and a null pointer if it isn't. The ***dynamic_cast*** operator does exactly that. For example, assume that ''the system'' invokes ***my_event_handler*( )** with a pointer to a ***BBwindow***, where an activity has occurred. I then might invoke my application code using ***Ival_box***'s ***do_something*( )**:

```
void my_event_handler(BBwindow* pw)
{
    if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))    // does pw point to an Ival_box?
        pb->do_something();
    else {
        // Oops! unexpected event
    }
}
```

One way of explaining what is going on is that ***dynamic_cast*** translates from the implementation-oriented language of the user-interface system to the language of the application. It is important to note what is *not* mentioned in this example: the actual type of the object. The object will be a particular kind of ***Ival_box***, say an ***Ival_slider***, implemented by a particular kind of ***BBwindow***, say a ***BBslider***. It is neither necessary nor desirable to make the actual type of the object explicit in this interaction between ''the system'' and the application. An interface exists to represent the essentials of an interaction. In particular, a well-designed interface hides inessential details.

Graphically, the action of

```
pb = dynamic_cast<Ival_box*>(pw)
```

can be represented like this:

The arrows from ***pw*** and ***pb*** represent the pointers into the object passed, whereas the rest of the arrows represent the inheritance relationships between the different parts of the object passed.

The use of type information at run time is conventionally referred to as ''run-time type information'' and often abbreviated to RTTI.

Casting from a base class to a derived class is often called a *downcast* because of the convention of drawing inheritance trees growing from the root down. Similarly, a cast from a derived class to a base is called an *upcast*. A cast that goes from a base to a sibling class, like the cast from ***BBwindow*** to ***Ival_box***, is called a *crosscast*.

15.4.1 Dynamic_cast [hier.dynamic.cast]

The ***dynamic_cast*** operator takes two operands, a type bracketed by **<** and **>**, and a pointer or reference bracketed by **(** and **)**.

Consider first the pointer case:

```
dynamic_cast<T*>(p)
```

If ***p*** is of type ***T***** or an accessible base class of ***T***, the result is exactly as if we had simply assigned ***p*** to a ***T*****. For example:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
        // ...
};

void f(BB_ival_slider* p)
{
        Ival_slider* pi1 = p;           // ok
        Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);           // ok

        BBslider* pbb1 = p;             // error: BBslider is a protected base
        BBslider* pbb2 = dynamic_cast<BBslider*>(p);                // ok: pbb2 becomes 0
}
```

That is the uninteresting case. However, it is reassuring to know that ***dynamic_cast*** doesn't allow accidental violation of the protection of private and protected base classes.

The purpose of ***dynamic_cast*** is to deal with the case in which the correctness of the conversion cannot be determined by the compiler. In that case,

```
dynamic_cast<T*>(p)
```

looks at the object pointed to by ***p*** (if any). If that object is of class ***T*** or has a unique base class of type ***T***, then ***dynamic_cast*** returns a pointer of type ***T**** to that object; otherwise, ***0*** is returned. If the value of ***p*** is ***0***, ***dynamic_cast<T*>(p)*** returns ***0***. Note the requirement that the conversion must be to a uniquely identified object. It is possible to construct examples where the conversion fails and ***0*** is returned because the object pointed to by ***p*** has more than one sub-object representing bases of type ***T*** (see §15.4.2).

A ***dynamic_cast*** requires a pointer or a reference to a polymorphic type to do a downcast or a crosscast. For example:

```
class My_slider: public Ival_slider { // polymorphic base (Ival_slider has virtual functions)
        // ...
};

class My_date : public Date {   // base not polymorphic (Date has no virtual functions)
        // ...
};

void g(Ival_box* pb, Date* pd)
{
        My_slider* pd1 = dynamic_cast<My_slider*>(pb);      // ok
        My_date* pd2 = dynamic_cast<My_date*>(pd);          // error: Date not polymorphic
}
```

Requiring the pointer's type to be polymorphic simplifies the implementation of ***dynamic_cast*** because it makes it easy to find a place to hold the necessary information about the object's type. A typical implementation will attach a ''type information object'' to an object by placing a pointer to the type information in the object's virtual function table (§2.5.5). For example:

The dashed arrow represents an offset that allows the start of the complete object to be found given only a pointer to a polymorphic sub-object. It is clear that ***dynamic_cast*** can be efficiently implemented. All that is involved are a few comparisons of ***type_info*** objects representing base classes; no expensive lookups or string comparisons are needed.

Restricting ***dynamic_cast*** to polymorphic types also makes sense from a logical point of view. This is, if an object has no virtual functions, it cannot safely be manipulated without knowledge of its exact type. Consequently, care should be taken not to get such an object into a context in which its type isn't known. If its type *is* known, we don't need to use ***dynamic_cast***.

The target type of ***dynamic_cast*** need not be polymorphic. This allows us to wrap a concrete type in a polymorphic type, say for transmission through an object I/O system (see §25.4.1), and then ''unwrap'' the concrete type later. For example:

```
class Io_obj {                  // base class for object I/O system
    virtual Io_obj* clone() = 0;
};

class Io_date : public Date, public Io_obj { };

void f(Io_obj* pio)
{
    Date* pd = dynamic_cast<Date*>(pio);
    // ...
}
```

A ***dynamic_cast*** to ***void**** can be used to determine the address of the beginning of an object of polymorphic type. For example:

```
void g(Ival_box* pb, Date* pd)
{
    void* pd1 = dynamic_cast<void*>(pb);     // ok
    void* pd2 = dynamic_cast<void*>(pd);     // error: Date not polymorphic
}
```

This is only useful for interaction with very low-level functions.

15.4.1.1 Dynamic_cast of References [hier.re.cast]

To get polymorphic behavior, an object must be manipulated through a pointer or a reference. When a ***dynamic_cast*** is used for a pointer type, a ***0*** indicates failure. That is neither feasible nor desirable for references.

Given a pointer result, we must consider the possibility that the result is *0*; that is, that the pointer doesn't point to an object. Consequently, the result of a ***dynamic_cast*** of a pointer should always be explicitly tested. For a pointer ***p***, ***dynamic_cast<T*>(p)*** can be seen as the question, ''Is the object pointed to by ***p*** of type ***T***?''

On the other hand, we may legitimately assume that a reference refers to an object. Consequently, ***dynamic_cast<T&>(r)*** of a reference ***r*** is not a question but an assertion: ''The object referred to by ***r*** is of type ***T***.'' The result of a ***dynamic_cast*** for a reference is implicitly tested by the implementation of ***dynamic_cast*** itself. If the operand of a ***dynamic_cast*** to a reference isn't of the expected type, a ***bad_cast*** exception is thrown. For example:

```
void f(Ival_box* p, Ival_box& r)
{
    if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {      // does p point to an Ival_slider?
        // use 'is'
    }
    else {
        // *p not a slider
    }

    Ival_slider& is = dynamic_cast<Ival_slider&>(r);        // r references an Ival_slider!
    // use 'is'
}
```

The difference in results of a failed dynamic pointer cast and a failed dynamic reference cast reflects a fundamental difference between references and pointers. If a user wants to protect against bad casts to references, a suitable handler must be provided. For example:

```
void g()
{
    try {
        f(new BB_ival_slider, *new BB_ival_slider);    // arguments passed as Ival_boxs
        f(new BBdial, *new BBdial);                    // arguments passed as Ival_boxs
    }
    catch (bad_cast) {    // §14.10
        // ...
    }
}
```

The first call to ***f()*** will return normally, while the second will cause a ***bad_cast*** exception that will be caught by ***g()***.

Explicit tests against *0* can be – and therefore occasionally will be – accidentally omitted. If that worries you, you can write a conversion function that throws an exception instead returning *0* (§15.8[1]) in case of failure.

15.4.2 Navigating Class Hierarchies [hier.navigate]

When only single inheritance is used, a class and its base classes constitute a tree rooted in a single base class. This is simple but often constraining. When multiple inheritance is used, there is no single root. This in itself doesn't complicate matters much. However, if a class appears more than

once in a hierarchy, we must be a bit careful when we refer to the object or objects that represent that class.

Naturally, we try to keep hierarchies as simple as our application allows (and no simpler). However, once a nontrivial hierarchy has been made we soon need to navigate it to find an appropriate class to use as an interface. This need occurs in two variants. That is, sometimes, we want to explicitly name an object of a base class or a member of a base class; §15.2.3 and §15.2.4.1 are examples of this. At other times, we want to get a pointer to the object representing a base or derived class of an object given a pointer to a complete object or some sub-object; §15.4 and §15.4.1 are examples of this.

Here, we consider how to navigate a class hierarchy using type conversions (casts) to gain a pointer of the desired type. To illustrate the mechanisms available and the rules that guide them, consider a lattice containing both a replicated base and a virtual base:

```
class Component : public virtual Storable { /* ... */ };
class Receiver : public Component { /* ... */ };
class Transmitter : public Component { /* ... */ };
class Radio : public Receiver, public Transmitter { /* ... */ };
```

Or graphically:

Here, a ***Radio*** object has two sub-objects of class ***Component***. Consequently, a ***dynamic_cast*** from ***Storable*** to ***Component*** within a ***Radio*** will be ambiguous and return a ***0***. There is simply no way of knowing which ***Component*** the programmer wanted:

```
void h1(Radio& r)
{
    Storable* ps = &r;
    // ...
    Component* pc = dynamic_cast<Component*>(ps); // pc = 0
}
```

This ambiguity is not in general detectable at compile time:

```
void h2(Storable* ps)    // ps might or might not point to a Component
{
    Component* pc = dynamic_cast<Component*>(ps);
    // ...
}
```

This kind of run-time ambiguity detection is needed only for virtual bases. For ordinary bases,

there is always a unique sub-object of a given cast (or none) when downcasting (that is, towards a derived class; §15.4). The equivalent ambiguity occurs when upcasting (that is, towards a base; §15.4) and such ambiguities are caught at compile time.

15.4.2.1 Static and Dynamic Casts [hier.static.cast]

A ***dynamic_cast*** can cast from a polymorphic virtual base class to a derived class or a sibling class (§15.4.1). A ***static_cast*** (§6.2.7) does not examine the object it casts from, so it cannot:

```
void g(Radio& r)
{
    Receiver* prec = &r;                        // Receiver is ordinary base of Radio
    Radio* pr = static_cast<Radio*>(prec);      // ok, unchecked
    pr = dynamic_cast<Radio*>(prec);            // ok, run-time checked

    Storable* ps = &r;                          // Storable is virtual base of Radio
    pr = static_cast<Radio*>(ps);               // error: cannot cast from virtual base
    pr = dynamic_cast<Radio*>(ps);              // ok, run-time checked
}
```

The ***dynamic_cast*** requires a polymorphic operand because there is no information stored in a non-polymorphic object that can be used to find the objects for which it represents a base. In particular, an object of a type with layout constraints determined by some other language – such as Fortran or C – may be used as a virtual base class. For objects of such types, only static type information will be available. However, the information needed to provide run-time type identification includes the information needed to implement the ***dynamic_cast***.

Why would anyone want to use a ***static_cast*** for class hierarchy navigation? There is a small run-time cost associated with the use of a ***dynamic_cast*** (§15.4.1). More significantly, there are millions of lines of code that were written before ***dynamic_cast*** became available. This code relies on alternative ways of making sure that a cast is valid, so the checking done by ***dynamic_cast*** is seen as redundant. However, such code is typically written using the C-style cast (§6.2.7); often obscure errors remain. Where possible, use the safer ***dynamic_cast***.

The compiler cannot assume anything about the memory pointed to by a ***void****. This implies that ***dynamic_cast*** – which must look into an object to determine its type – cannot cast from a ***void****. For that, a ***static_cast*** is needed. For example:

```
Radio* f(void* p)
{
    Storable* ps = static_cast<Storable*>(p);  // trust the programmer
    return dynamic_cast<Radio*>(ps);
}
```

Both ***dynamic_cast*** and ***static_cast*** respect ***const*** and access controls. For example:

```
class Users : private set<Person> { /* ... */ };
```

```
void f(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person>*>(pu);        // error: access violation
    dynamic_cast<set<Person>*>(pu);       // error: access violation

    static_cast<Receiver*>(pcr);          // error: can't cast away const
    dynamic_cast<Receiver*>(pcr);         // error: can't cast away const

    Receiver* pr = const_cast<Receiver*>(pcr);      // ok
    // ...
}
```

It is not possible to cast to a private base class, and ''casting away ***const***'' requires a ***const_cast*** (§6.2.7). Even then, using the result is safe only provided the object wasn't originally declared ***const*** (§10.2.7.1) .

15.4.3 Class Object Construction and Destruction [hier.class.obj]

A class object is more than simply a region of memory (§4.9.6). A class object is built from ''raw memory'' by its constructors and it reverts to ''raw memory'' as its destructors are executed. Construction is bottom up, destruction is top down, and a class object is an object to the extent that it has been constructed or destroyed. This is reflected in the rules for RTTI, exception handling (§14.4.7), and virtual functions.

It is extremely unwise to rely on details of the order of construction and destruction, but that order can be observed by calling virtual functions, ***dynamic_cast***, or ***typeid*** (§15.4.4) at a point where the object isn't complete. For example, if the constructor for ***Component*** in the hierarchy from §15.4.2 calls a virtual function, it will invoke a version defined for ***Storable*** or ***Component***, but not one from ***Receiver***, ***Transmitter***, or ***Radio***. At that point of construction, the object isn't yet a ***Radio***; it is merely a partially constructed object. It is best to avoid calling virtual functions during construction and destruction.

15.4.4 Typeid and Extended Type Information [hier.typeid]

The ***dynamic_cast*** operator serves most needs for information about the type of an object at run time. Importantly, it ensures that code written using it works correctly with classes derived from those explicitly mentioned by the programmer. Thus, ***dynamic_cast*** preserves flexibility and extensibility in a manner similar to virtual functions.

However, it is occasionally essential to know the exact type of an object. For example, we might like to know the name of the object's class or its layout. The ***typeid*** operator serves this purpose by yielding an object representing the type of its operand. Had ***typeid*( )** been a function, its declaration would have looked something like this:

```
class type_info;
const type_info& typeid(type_name) throw(bad_typeid);     // pseudo declaration
const type_info& typeid(expression);                      // pseudo declaration
```

That is, ***typeid*( )** returns a reference to a standard library type called ***type_info*** defined in ***<type-info>***. Given a *type-name* as its operand, ***typeid*( )** returns a reference to a ***type_info*** that represents the *type-name*. Given an *expression* as its operand, ***typeid*( )** returns a reference to a

type_info that represents the type of the object denoted by the *expression*. A *typeid*( ) is most commonly used to find the type of an object referred to by a reference or a pointer:

```
void f(Shape& r, Shape* p)
{
    typeid(r);      // type of object referred to by r
    typeid(*p);     // type of object pointed to by p
    typeid(p);      // type of pointer, that is, Shape* (uncommon, except as a mistake)
}
```

If the value of a pointer or a reference operand is *0*, *typeid*( ) throws a *bad_typeid* exception.

The implementation-independent part of *type_info* looks like this:

```
class type_info {
public:
    virtual ~type_info();                          // is polymorphic

    bool operator==(const type_info&) const;       // can be compared
    bool operator!=(const type_info&) const;
    bool before(const type_info&) const;           // ordering

    const char* name() const;                      // name of type
private:
    type_info(const type_info&);                   // prevent copying
    type_info& operator=(const type_info&);        // prevent copying
    // ...
};
```

The *before*( ) function allows *type_info*s to be sorted. There is no relation between the relationships defined by *before* and inheritance relationships.

It is *not* guaranteed that there is only one *type_info* object for each type in the system. In fact, where dynamically linked libraries are used it can be hard for an implementation to avoid duplicate *type_info* objects. Consequently, we should use == on *type_info* objects to test equality, rather than == on pointers to such objects.

We sometimes want to know the exact type of an object so as to perform some standard service on the whole object (and not just on some base of the object). Ideally, such services are presented as virtual functions so that the exact type needn't be known. In some cases, no common interface can be assumed for every object manipulated, so the detour through the exact type becomes necessary (§15.4.4.1). Another, much simpler, use has been to obtain the name of a class for diagnostic output:

```
#include<typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

The character representation of a class' name is implementation-defined. This C-style string resides in memory owned by the system, so the programmer should not attempt to *delete*[] it.

15.4.4.1 Extended Type Information [hier.extended]

Typically, finding the exact type of an object is simply the first step to acquiring and using more-detailed information about that type.

Consider how an implementation or a tool could make information about types available to users at run time. Suppose I have a tool that generates descriptions of object layouts for each class used. I can put these descriptors into a ***map*** to allow user code to find the layout information:

```
map<const char*, Layout> layout_table;

void f(B* p)
{
    Layout& x = layout_table[typeid(*p).name()];
    // use x
}
```

Someone else might provide a completely different kind of information:

```
struct TI_eq {
    bool operator()(const type_info* p, const type_info* q) { return *p==*q; }
};

struct TI_hash {
    int operator()(const type_info* p); // compute hash value (§17.6.2.2)
};

hash_map<type_info*, Icon, hash_fct, TI_hash, TI_eq> icon_table;    // §17.6

void g(B* p)
{
    Icon& i = icon_table[&typeid(*p)];
    // use i
}
```

This way of associating ***typeid***s with information allows several people or tools to associate different information with types totally independently of each other:

This is most important because the likelihood is zero that someone can come up with a single set of information that satisfies every user.

15.4.5 Uses and Misuses of RTTI [hier.misuse]

One should use explicit run-time type information only when necessary. Static (compile-time) checking is safer, implies less overhead, and – where applicable – leads to better-structured programs. For example, RTTI can be used to write thinly disguised *switch-statement*s:

```
// misuse of run-time type information:

void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // do nothing
    }
    else if (typeid(r) == typeid(Triangle)) {
        // rotate triangle
    }
    else if (typeid(r) == typeid(Square)) {
        // rotate square
    }
    // ...
}
```

Using ***dynamic_cast*** rather than ***typeid*** would improve this code only marginally.

Unfortunately, this is not a strawman example; such code really does get written. For many people trained in languages such as C, Pascal, Modula-2, and Ada, there is an almost irresistible urge to organize software as a set of *switch-statement*s. This urge should usually be resisted. Use virtual functions (§2.5.5, §12.2.6) rather than RTTI to handle most cases when run-time discrimination based on type is needed.

Many examples of proper use of RTTI arise when some service code is expressed in terms of one class and a user wants to add functionality through derivation. The use of ***Ival_box*** in §15.4 is an example of this. If the user is willing and able to modify the definitions of the library classes, say ***BBwindow***, then the use of RTTI can be avoided; otherwise, it is needed. Even if the user is willing to modify the base classes, such modification may cause its own problems. For example, it may be necessary to introduce dummy implementations of virtual functions in classes for which those functions are not needed or not meaningful. This problem is discussed in some detail in §24.4.3. A use of RTTI to implement a simple object I/O system can be found in §25.4.1.

For people with a background in languages that rely heavily on dynamic type checking, such as Smalltalk or Lisp, it is tempting to use RTTI in conjunction with overly general types. Consider:

```
// misuse of run-time type information:

class Object { /* ... */ }; // polymorphic

class Container : public Object {
public:
    void put(Object*);
    Object* get();
    // ...
};
```

```
class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
    c->put(ps);
    // ...
    Object* p = c->get();
    if (Ship* q = dynamic_cast<Ship*>(p)) { // run-time check
        return q;
    }
    else {
        // do something else (typically, error handling)
    }
}
```

Here, class ***Object*** is an unnecessary implementation artifact. It is overly general because it does not correspond to an abstraction in the application domain and forces the application programmer to use an implementation-level abstraction. Problems of this kind are often better solved by using container templates that hold only a single kind of pointer:

```
Ship* f(Ship* ps, list<Ship*>& c)
{
    c.push_front(ps);
    // ...
    return c.pop_front();
}
```

Combined with the use of virtual functions, this technique handles most cases.

15.5 Pointers to Members [hier.ptom]

Many classes provide simple, very general interfaces intended to be invoked in several different ways. For example, many ''object-oriented'' user-interfaces define a set of requests to which every object represented on the screen should be prepared to respond. In addition, such requests can be presented directly or indirectly from programs. Consider a simple variant of this idea:

```
class Std_interface {
public:
    virtual void start() = 0;
    virtual void suspend() = 0;
    virtual void resume() = 0;
    virtual void quit() = 0;
    virtual void full_size() = 0;
    virtual void small() = 0;

    virtual ~Std_interface() {}
};
```

The exact meaning of each operation is defined by the object on which it is invoked. Often, there is a layer of software between the person or program issuing the request and the object receiving it.

Ideally, such intermediate layers of software should not have to know anything about the individual operations such as ***resume***( ) and ***full_size***( ). If they did, the intermediate layers would have to be updated each time the set of operations changed. Consequently, such intermediate layers simply transmit some data representing the operation to be invoked from the source of the request to its recipient.

One simple way of doing that is to send a ***string*** representing the operation to be invoked. For example, to invoke ***suspend***( ) we could send the string *"suspend"*. However, someone has to create that string and someone has to decode it to determine to which operation it corresponds – if any. Often, that seems indirect and tedious. Instead, we might simply send an integer representing the operation. For example, *2* might be used to mean ***suspend***( ). However, while an integer may be convenient for machines to deal with, it can get pretty obscure for people. We still have to write code to determine that *2* means ***suspend***( ) and to invoke ***suspend***( ).

C++ offers a facility for indirectly referring to a member of a class. A pointer to a member is a value that identifies a member of a class. You can think of it as the position of the member in an object of the class, but of course an implementation takes into account the differences between data members, virtual functions, non-virtual functions, etc.

Consider ***Std_interface***. If I want to invoke ***suspend***( ) for some object without mentioning ***suspend***( ) directly, I need a pointer to member referring to ***Std_interface***::***suspend***( ). I also need a pointer or reference to the object I want to suspend. Consider a trivial example:

```
typedef void (Std_interface::* Pstd_mem)();    // pointer to member type

void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;

    p->suspend();          // direct call

    (p->*s)();             // call through pointer to member
}
```

A *pointer to member* can be obtained by applying the address-of operator & to a fully qualified class member name, for example, &***Std_interface***::***suspend***. A variable of type ‘‘pointer to member of class ***X***’’ is declared using a declarator of the form ***X***::*.

The use of ***typedef*** to compensate for the lack of readability of the C declarator syntax is typical. However, please note how the ***X***::* declarator matches the traditional * declarator exactly.

A pointer to member ***m*** can be used in combination with an object. The operators ->* and .* allow the programmer to express such combinations. For example, ***p***->****m*** binds ***m*** to the object pointed to by ***p***, and ***obj***.****m*** binds ***m*** to the object ***obj***. The result can be used in accordance with ***m***’s type. It is not possible to store the result of a ->* or a .* operation for later use.

Naturally, if we knew which member we wanted to call we would invoke it directly rather than mess with pointers to members. Just like ordinary pointers to functions, pointers to member functions are used when we need to refer to a function without having to know its name. However, a pointer to member isn’t a pointer to a piece of memory the way a pointer to a variable or a pointer to a function is. It is more like an offset into a structure or an index into an array. When a pointer to member is combined with a pointer to an object of the right type, it yields something that identifies a particular member of a particular object.

This can be represented graphically like this:

Because a pointer to a virtual member (*s* in this example) is a kind of offset, it does not depend on an object's location in memory. A pointer to a virtual member can therefore safely be passed between different address spaces as long as the same object layout is used in both. Like pointers to ordinary functions, pointers to non-virtual member functions cannot be exchanged between address spaces.

Note that the function invoked through the pointer to function can be ***virtual***. For example, when we call ***suspend***( ) through a pointer to function, we get the right ***suspend***( ) for the object to which the pointer to function is applied. This is an essential aspect of pointers to functions.

An interpreter might use pointers to members to invoke functions presented as strings:

```
map<string, Std_interface*> variable;
map<string, Pstd_mem> operation;

void call_member(string var, string oper)
{
        (variable[var]->*operation[oper])();     // var.oper()
}
```

A critical use of pointers to member functions is found in ***mem_fun***( ) (§3.8.5, §18.4).

A static member isn't associated with a particular object, so a pointer to a static member is simply an ordinary pointer. For example:

```
class Task {
        // ...
        static void schedule();
};

void (*p)() = &Task::schedule;              // ok
void (Task::* pm)() = &Task::schedule;      // error: ordinary pointer assigned
                                            // to pointer to member
```

Pointers to data members are described in §C.12.

15.5.1 Base and Derived Classes [hier.contravariance]

A derived class has at least the members that it inherits from its base classes. Often it has more. This implies that we can safely assign a pointer to a member of a base class to a pointer to a member of a derived class, but not the other way around. This property is often called *contravariance*. For example:

```
class text : public Std_interface {
public:
    void start();
    void suspend();
    // ...
    virtual void print();
private:
    vector s;
};

void (Std_interface::* pmi)() = &text::print;    // error
void (text::*pmt)() = &Std_interface::start;     // ok
```

This contravariance rule appears to be the opposite of the rule that says we can assign a pointer to a derived class to a pointer to its base class. In fact, both rules exist to preserve the fundamental guarantee that a pointer may never point to an object that doesn't at least have the properties that the pointer promises. In this case, ***Std_interface*::*** can be applied to any ***Std_interface***, and most such objects presumably are not of type ***text***. Consequently, they do not have the member ***text*::*print*** with which we tried to initialize ***pmi***. By refusing the initialization, the compiler saves us from a run-time error.

15.6 Free Store [hier.free]

It is possible to take over memory management for a class by defining ***operator new*( )** and ***operator delete*( )** (§6.2.6.2). However, replacing the global ***operator new*( )** and ***operator delete*( )** is not for the fainthearted. After all, someone else might rely on some aspect of the default behavior or might even have supplied other versions of these functions.

A more selective, and often better, approach is to supply these operations for a specific class. This class might be the base for many derived classes. For example, we might like to have the ***Employee*** class from §12.2.6 provide a specialized allocator and deallocator for itself and all of its derived classes:

```
class Employee {
    // ...
public:
    // ...
    void* operator new(size_t);
    void operator delete(void*, size_t);
};
```

Member ***operator new*( )**s and ***operator delete*( )**s are implicitly ***static*** members. Consequently, they don't have a ***this*** pointer and do not modify an object. They provide storage that a constructor can initialize and a destructor can clean up.

```
void* Employee::operator new(size_t s)
{
    // allocate 's' bytes of memory and return a pointer to it
}
```

```
void Employee::operator delete(void* p, size_t s)
{
    // assume 'p' points to 's' bytes of memory allocated by Employee::operator new()
    // and free that memory for reuse
}
```

The use of the hitherto mysterious ***size_t*** argument now becomes obvious. It is the size of the object actually deleted. Deleting a ''plain'' ***Employee*** gives an argument value of ***sizeof(Employee)***; deleting a ***Manager*** gives an argument value of ***sizeof(Manager)***. This allows a class-specific allocator to avoid storing size information with each allocation. Naturally, a class-specific allocator can store such information (like a general-purpose allocator must) and ignore the ***size_t*** argument to ***operator delete()***. However, that makes it harder to improve significantly on the speed and memory consumption of a general-purpose allocator.

How does a compiler know how to supply the right size to ***operator delete()***? As long as the type specified in the ***delete*** operation matches the actual type of the object, this is easy. However, that is not always the case:

```
class Manager : public Employee {
    int level;
    // ...
};

void f()
{
    Employee* p = new Manager;    // trouble (the exact type is lost)
    delete p;
}
```

In this case, the compiler will not get the size right. As when an array is deleted, the user must help. This is done by adding a virtual destructor to the base class, ***Employee***:

```
class Employee {
public:
    void* operator new(size_t);
    void operator delete(void*, size_t);
    virtual ~Employee();
    // ...
};
```

Even an empty destructor will do:

```
Employee::~Employee() { }
```

In principle, deallocation is then done from within the destructor (which knows the size). Furthermore, the presence of a destructor in ***Employee*** ensures that every class derived from it will be supplied with a destructor (thus getting the size right), even if the derived class doesn't have a user-defined destructor. For example:

```
void f()
{
    Employee* p = new Manager;
    delete p;        // now fine (Employee is polymorphic)
}
```

Allocation is done by a (compiler-generated) call:

```
Employee::operator new(sizeof(Manager))
```

and deallocation by a (compiler-generated) call:

```
Employee::operator delete(p,sizeof(Manager))
```

In other words, if you want to supply an allocator/deallocator pair that works correctly for derived classes, you must either supply a virtual destructor in the base class or refrain from using the ***size_t*** argument in the deallocator. Naturally, the language could have been designed to save you from such concerns. However, that can be done only by also ‘‘saving’’ you from the benefits of the optimizations possible in the less safe system.

15.6.1 Array Allocation [hier.array]

The ***operator new***( ) and ***operator delete***( ) functions allow a user to take over allocation and deallocation of individual objects; ***operator new***[ ]( ) and ***operator delete***[ ]( ) serve exactly the same role for the allocation and deallocation of arrays. For example:

```
class Employee {
public:
    void* operator new[](size_t);
    void operator delete[](void*, size_t);
    // ...
};

void f(int s)
{
    Employee* p = new Employee[s];
    // ...
    delete[] p;
}
```

Here, the memory needed will be obtained by a call,

```
Employee::operator new[](sizeof(Employee)*s+delta)
```

where ***delta*** is some minimal implementation-defined overhead, and released by a call:

```
Employee::operator delete[](p,s*sizeof(Employee)+delta)
```

The number of elements (***s***) is ‘‘remembered’’ by the system.

15.6.2 "Virtual Constructors" [hier.vctor]

After hearing about virtual destructors, the obvious question is, ''Can constructors be virtual?'' The short answer is no; a slightly longer one is, no, but you can easily get the effect you are looking for.

To construct an object, a constructor needs the exact type of the object it is to create. Consequently, a constructor cannot be virtual. Furthermore, a constructor is not quite an ordinary function. In particular, it interacts with memory management routines in ways ordinary member functions don't. Consequently, you cannot have a pointer to a constructor.

Both of these restrictions can be circumvented by defining a function that calls a constructor and returns a constructed object. This is fortunate because creating a new object without knowing its exact type is often useful. The *Ival_box_maker* (§12.4.4) is an example of a class designed specifically to do that. Here, I present a different variant of that idea, where objects of a class can provide users with a clone (copy) of themselves or a new object of their type. Consider:

```
class Expr {
public:
    Expr();                          // default constructor
    Expr(const Expr&);               // copy constructor

    virtual Expr* new_expr() { return new Expr(); }
    virtual Expr* clone() { return new Expr(*this); }
    // ...
};
```

Because functions such as *new_expr()* and *clone()* are virtual and they (indirectly) construct objects, they are often called ''virtual constructors'' – by a strange misuse of the English language. Each simply uses a constructor to create a suitable object.

A derived class can override *new_expr()* and/or *clone()* to return an object of its own type:

```
class Cond : public Expr {
public:
    Cond();
    Cond(const Cond&);

    Cond* new_expr() { return new Cond(); }
    Cond* clone() { return new Cond(*this); }
    // ...
};
```

This means that given an object of class *Expr*, a user can create a new object of ''just the same type.'' For example:

```
void user(Expr* p)
{
    Expr* p2 = p->new_expr();
    // ...
}
```

The pointer assigned to *p2* is of an appropriate, but unknown, type.

The return type of *Cond::new_expr()* and *Cond::clone()* was *Cond** rather than *Expr**.

This allows a ***Cond*** to be cloned without loss of type information. For example:

```
void user2(Cond* pc, Expr* pe)
{
    Cond* p2 = pc->clone();
    Cond* p3 = pe->clone();  // error
    // ...
}
```

The type of an overriding function must be the same as the type of the virtual function it overrides, except that the return type may be relaxed. That is, if the original return type was ***B****, then the return type of the overriding function may be ***D****, provided ***B*** is a public base of ***D***. Similarly, a return type of ***B***& may be relaxed to ***D***&.

Note that a similar relaxation of the rules for argument types would lead to type violations (see §15.8 [12]).

15.7 Advice [hier.advice]

[1] Use ordinary multiple inheritance to express a union of features; §15.2, §15.2.5.
[2] Use multiple inheritance to separate implementation details from an interface; §15.2.5.
[3] Use a ***virtual*** base to represent something common to some, but not all, classes in a hierarchy; §15.2.5.
[4] Avoid explicit type conversion (casts); §15.4.5.
[5] Use ***dynamic_cast*** where class hierarchy navigation is unavoidable; §15.4.1.
[6] Prefer ***dynamic_cast*** over ***typeid***; §15.4.4.
[7] Prefer ***private*** to ***protected***; §15.3.1.1.
[8] Don't declare data members ***protected***; §15.3.1.1.
[9] If a class defines ***operator delete*( )**, it should have a virtual destructor; §15.6.
[10] Don't call virtual functions during construction or destruction; §15.4.3.
[11] Use explicit qualification for resolution of member names sparingly and preferably use it in overriding functions; §15.2.1

15.8 Exercises [hier.exercises]

1. (∗1) Write a template ***ptr_cast*** that works like ***dynamic_cast***, except that it throws ***bad_cast*** rather than returning ***0***.
2. (∗2) Write a program that illustrates the sequence of constructor calls at the state of an object relative to RTTI during construction. Similarly illustrate destruction.
3. (∗3.5) Implement a version of a Reversi/Othello board game. Each player can be either a human or the computer. Focus on getting the program correct and (then) getting the computer player ''smart'' enough to be worth playing against.
4. (∗3) Improve the user interface of the game from §15.8[3].
5. (∗3) Define a graphical object class with a plausible set of operations to serve as a common base class for a library of graphical objects; look at a graphics library to see what operations were

supplied there. Define a database object class with a plausible set of operations to serve as a common base class for objects stored as sequences of fields in a database; look at a database library to see what operations were supplied there. Define a graphical database object with and without the use of multiple inheritance and discuss the relative merits of the two solutions.

6. (*2) Write a version of the ***clone*( )** operation from §15.6.2 that can place its cloned object in an ***Arena*** (see §10.4.11) passed as an argument. Implement a simple ***Arena*** as a class derived from ***Arena***.
7. (*2) Without looking in the book, write down as many C++ keywords you can.
8. (*2) Write a standards-conforming C++ program containing a sequence of at least ten consecutive keywords not separated by identifiers, operators, punctuation characters, etc.
9. (*2.5) Draw a plausible memory layout for a ***Radio*** as defined in §15.2.3.1. Explain how a virtual function call could be implemented.
10. (*2) Draw a plausible memory layout for a ***Radio*** as defined in §15.2.4. Explain how a virtual function call could be implemented.
11. (*3) Consider how ***dynamic_cast*** might be implemented. Define and implement a ***dcast*** template that behaves like ***dynamic_cast*** but relies on functions and data you define only. Make sure that you can add new classes to the system without having to change the definitions of ***dcast*** or previously-written classes.
12. (*2) Assume that the type-checking rules for arguments were relaxed in a way similar to the relaxation for return types so that a function taking a ***Derived**** could overwrite a ***Base****. Then write a program that would corrupt an object of class ***Derived*** without using a cast. Describe a safe relaxation of the overriding rules for argument types.

Part III

The Standard Library

This part describes the C++ standard library. It presents the design of the library and key techniques used in its implementation. The aim is to provide understanding of how to use the library, to demonstrate generally useful design and programming techniques, and to show how to extend the library in the ways in which it was intended to be extended.

Chapters

16 Library Organization and Containers
17 Standard Containers
18 Algorithms and Function Objects
19 Iterators and Allocators
20 Strings
21 Streams
22 Numerics

16

Library Organization and Containers

It was new. It was singular.
It was simple. It must succeed!
– H. Nelson

Design criteria for the standard library — library organization — standard headers — language support — container design — iterators — based containers — STL containers — ***vector*** — iterators — element access — constructors — modifiers — list operations — size and capacity — ***vector<bool>***— advice — exercises.

16.1 Standard Library Design [org.intro]

What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not, ''What ought to be in *some* library?'' but ''What ought to be in the *standard* library?'' The answer ''Everything!'' is a reasonable first approximation to an answer to the former question but not to the latter. A standard library is something that every implementer must supply so that every programmer can rely on it.

The C++ standard library:

[1] Provides support for language features, such as memory management (§6.2.6) and run-time type information (§15.4).

[2] Supplies information about implementation-defined aspects of the language, such as the largest ***float*** value (§22.2).

[3] Supplies functions that cannot be implemented optimally in the language itself for every system, such as ***sqrt*( )** (§22.3) and ***memmove*( )** (§19.4.6).

[4] Supplies nonprimitive facilities that a programmer can rely on for portability, such as lists (§17.2.2), maps (§17.4.1), sort functions (§18.7.1), and I/O streams (Chapter 21).

[5] Provides a framework for extending the facilities it provides, such as conventions and

support facilities that allow a user to provide I/O of a user-defined type in the style of I/O for built-in types.

[6] Provides the common foundation for other libraries.

In addition, a few facilities – such as random-number generators (§22.7) – are provided by the standard library simply because it is conventional and useful to do so.

The design of the library is primarily determined by the last three roles. These roles are closely related. For example, portability is commonly an important design criterion for a specialized library, and common container types such as lists and maps are essential for convenient communication between separately developed libraries.

The last role is especially important from a design perspective because it helps limit the scope of the standard library and places constraints on its facilities. For example, string and list facilities are provided in the standard library. If they were not, separately developed libraries could communicate only by using built-in types. However, pattern matching and graphics facilities are not provided. Such facilities are obviously widely useful, but they are rarely directly involved in communication between separately developed libraries.

Unless a facility is somehow needed to support these roles, it can be left to some library outside the standard. For good and bad, leaving something out of the standard library opens the opportunity for different libraries to offer competing realizations of an idea.

16.1.1 Design Constraints [org.constraints]

The roles of a standard library impose several constraints on its design. The facilities offered by the C++ standard library are designed to be:

[1] Invaluable and affordable to essentially every student and professional programmer, including the builders of other libraries.

[2] Used directly or indirectly by every programmer for everything within the scope of the library.

[3] Efficient enough to provide genuine alternatives to hand-coded functions, classes, and templates in the implementation of further libraries.

[4] Either policy-free or give the user the option to supply policies as arguments.

[5] Primitive in the mathematical sense. That is, a component that serves two weakly related roles will almost certainly suffer overheads compared to individual components designed to perform only a single role.

[6] Convenient, efficient, and reasonably safe for common uses.

[7] Complete at what they do. The standard library may leave major functions to other libraries, but if it takes on a task, it must provide enough functionality so that individual users or implementers need not replace it to get the basic job done.

[8] Blend well with and augment built-in types and operations.

[9] Type safe by default.

[10] Supportive of commonly accepted programming styles.

[11] Extensible to deal with user-defined types in ways similar to the way built-in types and standard-library types are handled.

For example, building the comparison criteria into a sort function is unacceptable because the same data can be sorted according to different criteria. This is why the C standard library ***qsort*( )** takes

a comparison function as an argument rather than relying on something fixed, say, the < operator (§7.7). On the other hand, the overhead imposed by a function call for each comparison compromises *qsort*( ) as a building block for further library building. For almost every data type, it is easy to do a comparison without imposing the overhead of a function call.

Is that overhead serious? In most cases, probably not. However, the function call overhead can dominate the execution time for some algorithms and cause users to seek alternatives. The technique of supplying comparison criteria through a template argument described in §13.4 solves that problem. The example illustrates the tension between efficiency and generality. A standard library is not just required to perform its tasks. It must also perform them efficiently enough not to tempt users to supply their own mechanisms. Otherwise, implementers of more advanced features are forced to bypass the standard library in order to remain competitive. This would add a burden to the library developer and seriously complicate the lives of users wanting to stay platform-independent or to use several separately developed libraries.

The requirements of ''primitiveness'' and ''convenience of common uses'' appear to conflict. The former requirement precludes exclusively optimizing the standard library for common cases. However, components serving common, but nonprimitive, needs can be included in the standard library in addition to the primitive facilities, rather than as replacements. The cult of orthogonality must not prevent us from making life convenient for the novice and the casual user. Nor should it cause us to leave the default behavior of a component obscure or dangerous.

16.1.2 Standard Library Organization [org.org]

The facilities of the standard library are defined in the ***std*** namespace and presented as a set of headers. The headers identify the major parts of the library. Thus, listing them gives an overview of the library and provides a guide to the description of the library in this and subsequent chapters.

The rest of this subsection is a list of headers grouped by function, accompanied by brief explanations and annotated by references to where they are discussed. The grouping is chosen to match the organization of the standard. A reference to the standard (such as §s.18.1) means that the facility is not discussed here.

A standard header with a name starting with the letter ***c*** is equivalent to a header in the C standard library. For every header ***<cX>*** defining names in the ***std*** namespace, there is a header ***<X.h>*** defining the same names in the global namespace (see §9.2.2).

Containers		
<vector>	*one-dimensional array of **T***	§16.3
<list>	*doubly-linked list of **T***	§17.2.2
<deque>	*double-ended queue of **T***	§17.2.3
<queue>	*queue of **T***	§17.3.2
<stack>	*stack of **T***	§17.3.1
<map>	*associative array of **T***	§17.4.1
<set>	*set of **T***	§17.4.3
<bitset>	*array of booleans*	§17.5.3

The associative containers ***multimap*** and ***multiset*** can be found in ***<map>*** and ***<set>***, respectively. The ***priority_queue*** is declared in ***<queue>***.

General Utilities		
<utility>	*operators and pairs*	§17.1.4, §17.4.1.2
<functional>	*function objects*	§18.4
<memory>	*allocators for containers*	§19.4.4
<ctime>	*C-style date and time*	§s.20.5

The *<memory>* header also contains the ***auto_ptr*** template that is primarily used to smooth the interaction between pointers and exceptions (§14.4.2).

Iterators		
<iterator>	*iterators and iterator support*	Chapter 19

Iterators provide the mechanism to make standard algorithms generic over the standard containers and similar types (§2.7.2, §19.2.1).

Algorithms		
<algorithm>	*general algorithms*	Chapter 18
<cstdlib>	***bsearch*( ) *qsort*( )**	§18.11

A typical general algorithm can be applied to any sequence (§3.8, §18.3) of any type of elements. The C standard library functions ***bsearch*( )** and ***qsort*( )** apply to built-in arrays with elements of types without user-defined copy constructors and destructors only (§7.7).

Diagnostics		
<exception>	*exception class*	§14.10
<stdexcept>	*standard exceptions*	§14.10
<cassert>	*assert macro*	§24.3.7.2
<cerrno>	*C-style error handling*	§20.4.1

Assertions relying on exceptions are described in §24.3.7.1.

Strings		
<string>	*string of **T***	Chapter 20
<cctype>	*character classification*	§20.4.2
<cwtype>	*wide-character classification*	§20.4.2
<cstring>	*C-style string functions*	§20.4.1
<cwchar>	*C-style wide-character string functions*	§20.4
<cstdlib>	*C-style string functions*	§20.4.1

The *<cstring>* header declares the ***strlen*( )**, ***strcpy*( )**, etc., family of functions. The *<cstdlib>* declares ***atof*( )** and ***atoi*( )** that convert C-style strings to numeric values.

Input/Output		
<iosfwd>	*forward declarations of I/O facilities*	§21.1
<iostream>	*standard iostream objects and operations*	§21.2.1
<ios>	*iostream bases*	§21.2.1
<streambuf>	*stream buffers*	§21.6
<istream>	*input stream template*	§21.3.1
<ostream>	*output stream template*	§21.2.1
<iomanip>	*manipulators*	§21.4.6.2
<sstream>	*streams to/from strings*	§21.5.3
<cstdlib>	*character classification functions*	§20.4.2
<fstream>	*streams to/from files*	§21.5.1
<cstdio>	***printf*( ) *family of I/O*	§21.8
<cwchar>	***printf*( )*-style I/O of wide characters*	§21.8

Manipulators are objects used to manipulate the state of a stream (e.g., changing the format of floating-point output) by applying them to the stream (§21.4.6).

Localization		
<locale>	*represent cultural differences*	§21.7
<clocale>	*represent cultural differences C-style*	§21.7

A ***locale*** localizes differences such as the output format for dates, the symbol used to represent currency, and string collation criteria that vary among different natural languages and cultures.

Language Support		
<limits>	*numeric limits*	§22.2
<climits>	*C-style numeric scalar-limit macros*	§22.2.1
<cfloat>	*C-style numeric floating-point limit macros*	§22.2.1
<new>	*dynamic memory management*	§16.1.3
<typeinfo>	*run-time type identification support*	§15.4.1
<exception>	*exception-handling support*	§14.10
<cstddef>	*C library language support*	§6.2.1
<cstdarg>	*variable-length function argument lists*	§7.6
<csetjmp>	*C-style stack unwinding*	§s.18.7
<cstdlib>	*program termination*	§9.4.1.1
<ctime>	*system clock*	§s.18.7
<csignal>	*C-style signal handling*	§s.18.7

The ***<cstddef>*** header defines the type of values returned by ***sizeof*( )**, ***size_t***, the type of the result of pointer subtraction, ***ptrdiff_t*** (§6.2.1), and the infamous ***NULL*** macro (§5.1.1).

Numerics		
<complex>	*complex numbers and operations*	§22.5
<valarray>	*numeric vectors and operations*	§22.4
<numeric>	*generalized numeric operations*	§22.6
<cmath>	*standard mathematical functions*	§22.3
<cstdlib>	*C-style random numbers*	§22.7

For historical reasons, ***abs*( )**, ***fabs*( )**, and ***div*( )** are found in ***<cstdlib>*** rather than in ***<cmath>*** with the rest of the mathematical functions (§22.3).

A user or a library implementer is not allowed to add or subtract declarations from the standard headers. Nor is it acceptable to try to change the contents of headers by defining macros before they are included or to try to change the meaning of the declarations in the headers by declarations in their context (§9.2.3). Any program or implementation that plays such games does not conform to the standard, and programs that rely on such tricks are not portable. Even if they work today, the next release of any part of an implementation may break them. Avoid such trickery.

For a standard library facility to be used its header must be included. Writing out the relevant declarations yourself is *not* a standards-conforming alternative. The reason is that some implementations optimize compilation based on standard header inclusion and others provide optimized implementations of standard library facilities triggered by the headers. In general, implementers use standard headers in ways programmers cannot predict and shouldn't have to know about.

A programmer can, however, specialize utility templates, such as ***swap*( )** (§16.3.9), for nonstandard-library, user-defined types.

16.1.3 Language Support [org.lang]

A small part of the standard library is language support; that is, facilities that must be present for a program to run because language features depend on them.

The library functions supporting operators ***new*** and ***delete*** are discussed in §6.2.6, §10.4.11, §14.4.4, and §15.6; they are presented in ***<new>***.

Run-time type identification relies on class ***type_info***, which is described in §15.4.4 and presented in ***<typeinfo>***.

The standard exception classes are discussed in §14.10 and presented in ***<new>***, ***<typeinfo>***, ***<ios>***, ***<exception>***, and ***<stdexcept>***.

Program start and termination are discussed in §3.2, §9.4, and §10.4.9.

16.2 Container Design [org.cont]

A container is an object that holds other objects. Examples are lists, vectors, and associative arrays. In general, you can add objects to a container and remove objects from it.

Naturally, this idea can be presented to users in many different ways. The C++ standard library containers were designed to meet two criteria: to provide the maximum freedom in the design of an individual container, while at the same time allowing containers to present a common interface to users. This allows optimal efficiency in the implementation of containers and enables users to write code that is independent of the particular container used.

Container designs typically meet just one or the other of these two design criteria. The container and algorithms part of the standard library (often called the STL) can be seen as a solution to the problem of simultaneously providing generality and efficiency. The following sections present the strengths and weaknesses of two traditional styles of containers as a way of approaching the design of the standard containers.

16.2.1 Specialized Containers and Iterators [org.specialized]

The obvious approach to providing a vector and a list is to define each in the way that makes the most sense for its intended use:

```
template<class T> class Vector {        // optimal
public:
    explicit Vector(size_t n);  // initialize to hold n objects with value T()

    T& operator[](size_t);      // subscripting
    // ...
};

template<class T> class List {   // optimal
public:
    class Link { /* ... */ };

    List();             // initially empty
    void put(T*);       // put before current element
    T* get();           // get current element

    // ...
};
```

Each class provides operations that are close to ideal for their use, and for each class we can choose a suitable representation without worrying about other kinds of containers. This allows the implementations of operations to be close to optimal. In particular, the most common operations such as ***put***( ) for a ***List*** and ***operator***[ ]( ) for a ***Vector*** are small and easily inlined.

A common use of most kinds of containers is to iterate through the container looking at the elements one after the other. This is typically done by defining an iterator class appropriate to the kind of container (see §11.5 and §11.14[7]).

However, a user iterating over a container often doesn't care whether data is stored in a ***List*** or a ***Vector***. In that case, the code iterating should not depend on whether a ***List*** or a ***Vector*** was used. Ideally, the same piece of code should work in both cases.

A solution is to define an iterator class that provides a get-next-element operation that can be implemented for any container. For example:

```
template<class T> class Itor {   // common interface (abstract class §2.5.4, §12.3)
public:
    // return 0 to indicate no-more-elements

    virtual T* first() = 0;     // pointer to first element
    virtual T* next() = 0;      // pointer to next element
};
```

We can now provide implementations for ***Vector***s and ***List***s:

```
template<class T> class Vector_itor : public Itor<T> {      // Vector implementation
        Vector<T>& v;
        size_t index;     // index of current element
public:
        Vector_itor(Vector<T>& vv) :v(vv), index(0) { }
        T* first() { return (v.size()) ? &v[index=0] : 0; }
        T* next() { return (++index<v.size()) ? &v[index] : 0; }
};

template<class T> class List_itor : public Itor<T> {          // List implementation
        List<T>& lst;
        List<T>::Link p;      // points to current element
public:
        List_itor(List<T>&);
        T* first();
        T* next();
};
```

Or graphically, using dashed lines to represent ‘‘implemented using:’’

The internal structure of the two iterators is quite different, but that doesn’t matter to users. We can now write code that iterates over anything for which we can implement an ***Itor***. For example:

```
int count(Itor<char>& ii, char term)
{
        int c = 0;
        for (char* p = ii.first(); p; p=ii.next()) if (*p==term) c++;
        return c;
}
```

There is a snag, however. The operations on an ***Itor*** iterator are simple, yet they incur the overhead of a (virtual) function call. In many situations, this overhead is minor compared to what else is being done. However, iterating through a simple container is the critical operation in many high-performance systems and a function call is many times more expensive than the integer addition or pointer dereferencing that implements ***next*( )** for a ***vector*** and a ***list***. Consequently, this model is unsuitable, or at least not ideal, for a standard library.

However, this container-and-iterator model has been successfully used in many systems. For years, it was my favorite for most applications. Its strengths and weaknesses can be summarized like this:

+ Individual containers are simple and efficient.
+ Little commonality is required of containers. Iterators and wrapper classes (§25.7.1) can be used to fit independently developed containers into a common framework.

+ Commonality of use is provided through iterators (rather than through a general container type; §16.2.2).
+ Different iterators can be defined to serve different needs for the same container.
+ Containers are by default type safe and homogeneous (that is, all elements in a container are of the same type). A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
+ The containers are non-intrusive (that is, an object need not have a special base class or link field to be a member of a container). Non-intrusive containers work well with built-in types and with ***struct***s with externally-imposed layouts.
– Each iterator access incurs the overhead of a virtual function call. The time overhead can be serious compared to simple inlined access functions.
– A hierarchy of iterator classes tends to get complicated.
– There is nothing in common for every container and nothing in common for every object in every container. This complicates the provision of universal services such as persistence and object I/O.

A + indicates an advantage and a - indicates a disadvantage.

I consider the flexibility provided by iterators especially important. A common interface, such as ***Itor***, can be provided long after the design and implementation of containers (here, ***Vector*** and ***List***). When we design, we typically first invent something fairly concrete. For example, we design an array and invent a list. Only later do we discover an abstraction that covers both arrays and lists in a given context.

As a matter of fact, we can do this ''late abstraction'' several times. Suppose we want to represent a set. A set is a very different abstraction from ***Itor***, yet we can provide a ***Set*** interface to ***Vector*** and ***List*** in much the same way that I provided ***Itor*** as an interface to ***Vector*** and ***List***:

Thus, late abstraction using abstract classes allows us to provide different implementations of a concept even when there is no significant similarity between the implementations. For example, lists and vectors have some obvious commonality, but we could easily implement an ***Itor*** for an ***istream***.

Logically, the last two points on the list are the main weaknesses of the approach. That is, even if the function call overhead for iterators and similar interfaces to containers were eliminated (as is possible in some contexts), this approach would not be ideal for a standard library.

Non-intrusive containers incur a small overhead in time and space for some containers compared with intrusive containers. I have not found this a problem. Should it become a problem, an iterator such as ***Itor*** can be provided for an intrusive container (§16.5[11]).

16.2.2 Based Containers [org.based]

One can define an intrusive container without relying on templates or any other way of parameterizing a type declaration. For example:

```
struct Link {
    Link* pre;
    Link* suc;
    // ...
};

class List {
    Link* head;
    Link* curr;             // current element
public:
    Link* get();            // remove and return current element
    void put(Link*);        // insert before current element
    // ...
};
```

A ***List*** is now a list of ***Link***s, and it can hold objects of any type derived from ***Link***. For example:

```
class Ship : public Link { /* ... */ };

void f(List* lst)
{
    while (Link* po = lst->get()) {
        if (Ship* ps = dynamic_cast<Ship*>(po)) {      // Ship must be polymorphic (§15.4.1)
            // use ship
        }
        else {
            // Oops, do something else
        }
    }
}
```

Simula defined its standard containers in this style, so this approach can be considered the original for languages supporting object-oriented programming. These days, a common class for all objects is usually called ***Object*** or something similar. An ***Object*** class typically provides other common services in addition to serving as a link for containers.

Often, but not necessarily, this approach is extended to provide a common container type:

```
class Container : public Object {
public:
    virtual Object* get();                  // remove and return current element
    virtual void put(Object*);              // insert before current element
    virtual Object*& operator[](size_t);    // subscripting
    // ...
};
```

Note that the operations provided by ***Container*** are virtual so that individual containers can override them appropriately:

```
class List : public Container {
public:
    Object* get();
    void put(Object*);
    // ...
};

class Vector : public Container {
public:
    Object*& operator[](size_t);
    // ...
};
```

One problem arises immediately. What operations do we want ***Container*** to provide? We could provide only the operations that every container can support. However, the intersection of the sets of operations on all containers is a ridiculously narrow interface. In fact, in many interesting cases that intersection is empty. So, realistically, we must provide the union of essential operations on the variety of containers we intend to support. Such a union of interfaces to a set of concepts is called a *fat interface* (§24.4.3).

We can either provide default implementations of the functions in the fat interface or force every derived class to implement every function by making them pure virtual functions. In either case, we end up with a lot of functions that simply report a run-time error. For example:

```
class Container : public Object {
public:
    struct Bad_op { // exception class
        const char* p;
        Bad_op(const char* pp) :p(pp) { }
    };

    virtual void put(Object*) { throw Bad_op("put"); }
    virtual Object* get() { throw Bad_op("get"); }
    virtual Object*& operator[](int) { throw Bad_op("[]"); }
    // ...
};
```

If we want to protect against the possibility of a container that does not support ***get()***, we must catch ***Container::Bad_op*** somewhere. We could now write the ***Ship*** example like this:

```
class Ship : public Object { /* ... */ };

void f1(Container* pc)
{
    try {
        while (Object* po = pc->get()) {
            if (Ship* ps = dynamic_cast<Ship*>(po)) {
                // use ship
            }
```

```
                else {
                        // Oops, do something else
                }
            }
        }
        catch (Container::Bad_op& bad) {
                // Oops, do something else
        }
    }
```

This is tedious, so the checking for ***Bad_op*** will typically be elsewhere. By relying on exceptions caught elsewhere, we can reduce the example to:

```
void f2(Container* pc)
{
    while (Object* po = pc->get()) {
        Ship& s = dynamic_cast<Ship&>(*po);
        // use ship
    }
}
```

However, I find unnecessary reliance on run-time checking distasteful and inefficient. In this kind of case, I prefer the statically-checked alternative:

```
void f3(Itor<Ship>* i)
{
    while (Ship* ps = i->next()) {
        // use ship
    }
}
```

The strengths and weakness of the ‘‘based object’’ approach to container design can be summarized like this (see also §16.5[10]):

– Operations on individual containers incur virtual function overhead.
– All containers must be derived from ***Container***. This implies the use of fat interfaces, requires a large degree of foresight, and relies on run-time type checking. Fitting an independently developed container into the common framework is awkward at best (see §16.5[12]).
+ The common base ***Container*** makes it easy to use containers that supply similar sets of operations interchangeably.
– Containers are heterogeneous and not type safe by default (all we can rely on is that elements are of type ***Object****). When desired, type-safe and homogeneous containers can be defined using templates.
– The containers are intrusive (that is, every element must be of a type derived from ***Object***). Objects of built-in types and structs with externally imposed layouts cannot be placed directly in containers.
– An element retrieved from a container must be given a proper type using explicit type conversion before it can be used.
+ Class ***Container*** and class ***Object*** are handles for implementing services for every object or

every container. This greatly eases the provision of universal services such as persistence and object I/O.

As before (§16.2.1), + indicates an advantage and - indicates a disadvantage.

Compared to the approach using unrelated containers and iterators, the based-object approach unnecessarily pushes complexity onto the user, imposes significant run-time overheads, and restricts the kinds of objects that can be placed in a container. In addition, for many classes, to derive from ***Object*** is to expose an implementation detail. Thus, this approach is far from ideal for a standard library.

However, the generality and flexibility of this approach should not be underestimated. Like its alternatives, it has been used successfully in many applications. Its strengths lie in areas in which efficiency is less important than the simplicity afforded by a single ***Container*** interface and services such as object I/O.

16.2.3 STL Containers [org.stl]

The standard library containers and iterators (often called the STL framework, §3.10) can be understood as an approach to gain the best of the two traditional models described previously. That wasn't the way the STL was designed, though. The STL was the result of a single-minded search for uncompromisingly efficient and generic algorithms.

The aim of efficiency rules out hard-to-inline virtual functions for small, frequently-used access functions. Therefore, we cannot present a standard interface to containers or a standard iterator interface as an abstract class. Instead, each kind of container supports a standard set of basic operations. To avoid the problems of fat interfaces (§16.2.2, §24.4.3), operations that cannot be efficiently implemented for all containers are not included in the set of common operations. For example, subscripting is provided for ***vector*** but not for ***list***. In addition, each kind of container provides its own iterators that support a standard set of iterator operations.

The standard containers are not derived from a common base. Instead, every container implements all of the standard container interface. Similarly, there is no common iterator base class. No explicit or implicit run-time type checking is involved in using the standard containers and iterators.

The important and difficult issue of providing common services for all containers is handled through ''allocators'' passed as template arguments (§19.4.3) rather than through a common base.

Before I go into details and code examples, the strengths and weaknesses of the STL approach can be summarized:

+ Individual containers are simple and efficient (not quite as simple as truly independent containers can be, but just as efficient).
+ Each container provides a set of standard operations with standard names and semantics. Additional operations are provided for a particular container type as needed. Furthermore, wrapper classes (§25.7.1) can be used to fit independently developed containers into a common framework (§16.5[14]).
+ Additional commonality of use is provided through standard iterators. Each container provides iterators that support a set of standard operations with standard names and semantics. An iterator type is defined for each particular container type so that these iterators are as simple and efficient as possible.

+ To serve different needs for containers, different iterators and other generalized interfaces can be defined in addition to the standard iterators.
+ Containers are by default type-safe and homogeneous (that is, all elements in a container are of the same type). A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
+ The containers are non-intrusive (that is, an object need not have a special base class or link field to be a member of a container). Non-intrusive containers work well with built-in types and with ***struct***s with externally imposed layouts.
+ Intrusive containers can be fitted into the general framework. Naturally, an intrusive container will impose constraints on its element types.
+ Each container takes an argument, called an ***allocator***, which can be used as a handle for implementing services for every container. This greatly eases the provision of universal services such as persistence and object I/O (§19.4.3).
– There is no standard run-time representation of containers or iterators that can be passed as a function argument (although it is easy to define such representations for the standard containers and iterators where needed for a particular application; §19.3).

As before (§16.2.1), + indicates an advantage and - indicates a disadvantage.

In other words, containers and iterators do not have fixed standard representations. Instead, each container provides a standard interface in the form of a set of operations so that containers can be used interchangeably. Iterators are handled similarly. This implies minimal overheads in time and space while allowing users to exploit commonality both at the level of containers (as with the based-object approach) and at the level of iterators (as with the specialized container approach).

The STL approach relies heavily on templates. To avoid excessive code replication, partial specialization to provide shared implementations for containers of pointers is usually required (§13.5).

16.3 Vector [org.vector]

Here, ***vector*** is described as an example of a complete standard container. Unless otherwise stated, what is said about ***vector*** holds for every standard container. Chapter 17 describes features peculiar to ***list***s, ***set***s, ***map***s, etc. The facilities offered by ***vector*** – and similar containers – are described in some detail. The aim is to give an understanding both of the possible uses of ***vector*** and of its role in the overall design of the standard library.

An overview of the standard containers and the facilities they offer can be found in §17.1. Below, ***vector*** is introduced in stages: member types, iterators, element access, constructors, stack operations, list operations, size and capacity, helper functions, and ***vector<bool>***.

16.3.1 Types [org.types]

The standard ***vector*** is a template defined in namespace ***std*** and presented in ***<vector>***. It first defines a set of standard names of types:

```
template <class T, class A = allocator<T> > class std::vector {
public:
    // types:
```

```
    typedef T value_type;                          // type of element
    typedef A allocator_type;                      // type of memory manager
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef implementation_dependent1 iterator;    // T*
    typedef implementation_dependent2 const_iterator;  // const T*
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    typedef typename A::pointer pointer;           // pointer to element
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;       // reference to element
    typedef typename A::const_reference const_reference;

    // ...
};
```

Every standard container defines these typenames as members. Each defines them in the way most appropriate to its implementation.

The type of the container's elements is passed as the first template argument and is known as its ***value_type***. The ***allocator_type***, which is optionally supplied as the second template argument, defines how the ***value_type*** interacts with various memory management mechanisms. In particular, an allocator supplies the functions that a container uses to allocate and deallocate memory for its elements. Allocators are discussed in §19.4. In general, ***size_type*** specifies the type used for indexing into the container, and ***difference_type*** is the type of the result of subtracting two iterators for a container. For most containers, they correspond to ***size_t*** and ***ptrdiff_t*** (§6.2.1).

Iterators were introduced in §2.7.2 and are described in detail in Chapter 19. They can be thought of as pointers to elements of the container. Every container provides a type called ***iterator*** for pointing to elements. It also provides a ***const_iterator*** type for use when elements don't need to be modified. As with pointers, we use the safer ***const*** version unless there is a reason to do otherwise. The actual types of ***vector***'s iterators are implementation-defined. The obvious definitions for a conventionally-defined ***vector*** would be ***T**** and ***const T****, respectively.

The reverse iterator types for ***vector*** are constructed from the standard ***reverse_iterator*** templates (§19.2.5). They present a sequence in the reverse order.

As shown in §3.8.1, these member typenames allow a user to write code using a container without having to know about the actual types involved. In particular, they allow a user to write code that will work for any standard container. For example:

```
template<class C> typename C::value_type sum(const C& c)
{
    typename C::value_type s = 0;
    typename C::const_iterator p = c.begin();      // start at the beginning
    while (p!=c.end()) {                           // continue until the end
        s += *p;          // get value of element
        ++p;              // make p point to next element
    }
    return s;
}
```

Having to add ***typename*** before the names of member types of a template parameter is a nuisance. However, the compiler isn't psychic. There is no general way for it to know whether a member of a template argument type is a typename (§C.13.5).

As for pointers, prefix * means dereference the iterator (§2.7.2, §19.2.1) and ++ means increment the iterator.

16.3.2 Iterators [org.begin]

As shown in the previous subsection, iterators can be used to navigate containers without the programmers having to know the actual type used to identify elements. A few key member functions allow the programmer to get hold of the ends of the sequence of elements:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // iterators:

    iterator begin();                    // points to first element
    const_iterator begin() const;
    iterator end();                      // points to one-past-last element
    const_iterator end() const;

    reverse_iterator rbegin();           // points to first element of reverse sequence
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();             // points to one-past-last element of reverse sequence
    const_reverse_iterator rend() const;

    // ...
};
```

The ***begin*()/*end*()** pair gives the elements of the container in the ordinary element order. That is, element ***0*** is followed by element ***1***, element ***2***, etc. The ***rbegin*()/*rend*()** pair gives the elements in the reverse order. That is, element ***n-1*** is followed by element ***n-2***, element ***n-3***, etc. For example, a sequence seen like this using an ***iterator***:

can be viewed like this using a ***reverse_iterator*** (§19.2.5):

This allows us to use algorithms in a way that views a sequence in the reverse order. For example:

```
template<class C>
typename C::iterator find_last(const C& c, typename C::value_type v)
{
    return find_first(c.rbegin(), c.rend(), v).base();
}
```

The *base*( ) function returns an *iterator* corresponding to the *reverse_iterator* (§19.2.5). Without reverse iterators, we could have had to write something like:

```
template<class C>
typename C::iterator find_last(const C& c, typename C::value_type v)
{
    typename C::iterator p = c.end();   //search backwards from end
    while (p!=c.begin()) {
        --p;
        if (*p==v) return p;
    }
    return p;
}
```

A reverse iterator is a perfectly ordinary iterator, so we could have written:

```
template<class C>
typename C::reverse_iterator find_last(const C& c, typename C::value_type v)
{
    typename C::reverse_iterator p = c.rbegin();   // view sequence in reverse order
    while (p!=c.rend()) {
        if (*p==v) return p;
        ++p;                // note: not decrement (--)
    }
    return p;
}
```

16.3.3 Element Access [org.element]

One important aspect of a *vector* compared with other containers is that one can easily and efficiently access individual elements in any order:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // element access:

    reference operator[](size_type n);          // unchecked access
    const_reference operator[](size_type n) const;

    reference at(size_type n);                  // checked access
    const_reference at(size_type n) const;
```

```
        reference front();                          // first element
        const_reference front() const;
        reference back();                           // last element
        const_reference back() const;

        // ...
};
```

Indexing is done by *operator*[]() and *at*(); *operator*[]() provides unchecked access, whereas *at*() does a range check and throws *out_of_range* if an index is out of range. For example:

```
void f(vector<int>& v, int i1, int i2)
try {
        for(int i = 0; i < v.size(); i++) {
                // range already checked: use unchecked v[i] here
        }

        v.at(i1) = v.at(i2); // check range on access

        // ...
}
catch(out_of_range) {
        // oops: out-of-range error
}
```

This illustrates one idea for use. That is, if the range has already been checked, the unchecked subscripting operator can be used safely; otherwise, it is wise to use the range-checked *at*() function. This distinction is important when efficiency is at a premium. When that is not the case or when it is not perfectly obvious whether a range has been correctly checked, it is safer to use a vector with a checked [] operator (such as *Vec* from §3.7.1) or a checked iterator (§19.3).

The default access is unchecked to match arrays. Also, you can build a safe (checked) facility on top of a fast one but not a faster facility on top of a slower one.

The access operations return values of type *reference* or *const_reference* depending on whether or not they are applied to a *const* object. A reference is some suitable type for accessing elements. For the simple and obvious implementation of *vector<X>*, *reference* is simply *X*& and *const_reference* is simply *const X*&. The effect of trying to create an out-of-range reference is undefined. For example:

```
void f(vector<double>& v)
{
        double d = v[v.size()];    // undefined: bad index

        list<char> lst;
        char c = lst.front();      // undefined: list is empty
}
```

Of the standard sequences, only *vector* and *deque* (§17.2.3) support subscripting. The reason is the desire not to confuse users by providing fundamentally inefficient operations. For example, subscripting could have been provided for *list* (§17.2.2), but doing that would have been dangerously inefficient (that is, *O*(*n*)).

The members *front*() and *back*() return references to the first and last element, respectively.

They are most useful where these elements are known to exist and in code where these elements are of particular interest. A *vector* used as a *stack* (§16.3.5) is an obvious example. Note that *front*( ) returns a reference to the element to which *begin*( ) returns an iterator. I often think of *front*( ) as the first element and *begin*( ) as a pointer to the first element. The correspondence between *back*( ) and *end*( ) is less simple: *back*( ) is the last element and *end*( ) points to the last-plus-one element position.

16.3.4 Constructors [org.ctor]

Naturally, *vector* provides a complete set (§11.7) of constructors, destructor, and copy operations:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // constructors, etc.:

    explicit vector(const A& = A());
    explicit vector(size_type n, const T& val = T(), const A& = A());    // n copies of val
    template <class In>                                  // In must be an input iterator (§19.2.1)
        vector(In first, In last, const A& = A()); // copy from [first:last[
    vector(const vector& x);

    ~vector();

    vector& operator=(const vector& x);

    template <class In>                          // In must be an input iterator (§19.2.1)
        void assign(In first, In last);          // copy from [first:last[
    void assign(size_type n, const T& val);      // n copies of val

    // ...
};
```

A *vector* provides fast access to arbitrary elements, but changing its size is relatively expensive. Consequently, we typically give an initial size when we create a *vector*. For example:

```
vector<Record> vr(10000);

void f(int s1, int s2)
{
    vector<int> vi(s1);

    vector<double>* p = new vector<double>(s2);
}
```

Elements of a vector allocated this way are initialized by the default constructor for the element type. That is, each of *vr*'s *10000* elements is initialized by *Record*( ) and each of *vi*'s *s1* elements is initialized by *int*( ). Note that the default constructor for a built-in type performs initialization to *0* of the appropriate type (§4.9.5, §10.4.2).

If a type does not have a default constructor, it is not possible to create a vector with elements of that type without explicitly providing the value of each element. For example:

```
class Num {     // infinite precision
public:
        Num(long);
        // no default constructor
        // ...
};

vector<Num> v1(1000);                // error: no default Num
vector<Num> v2(1000,Num(0));         // ok
```

Since a **vector** cannot have a negative number of elements, its size must be non-negative. This is reflected in the requirement that **vector**'s **size_type** must be an **unsigned** type. This allows a greater range of vector sizes on some architectures. However, it can also lead to surprises:

```
void f(int i)
{
        vector<char> vc0(-1);       // fairly easy for compiler to warn against
        vector<char> vc1(i);
}

void g()
{
        f(-1);       // trick f() into accepting a large positive number!
}
```

In the call **f(-1)**, **-1** is converted into a (rather large) positive integer (§C.6.3). If we are lucky, the compiler will find a way of complaining.

The size of a **vector** can also be provided implicitly by giving the initial set of elements. This is done by supplying the constructor with a sequence of values from which to construct the **vector**. For example:

```
void f(const list<X>& lst)
{
        vector<X> v1(lst.begin(),lst.end());       // copy elements from list

        char p[] = "despair";
        vector<char> v2(p,&p[sizeof(p)-1]);       // copy characters from C-style string
}
```

In each case, the **vector** constructor adjusts the size of the **vector** as it copies elements from its input sequence.

The **vector** constructors that can be invoked with a single argument are declared **explicit** to prevent accidental conversions (§11.7.1). For example:

```
vector<int> v1(10);                  // ok: vector of 10 ints
vector<int> v2 = vector<int>(10);    // ok: vector of 10 ints
vector<int> v3 = v2;                 // ok: v3 is a copy of v2
vector<int> v4 = 10;                 // error: attempted implicit conversion of 10 to vector<int>
```

The copy constructor and the copy-assignment operators copy the elements of a **vector**. For a **vector** with many elements, that can be an expensive operation, so **vector**s are typically passed by reference. For example:

```
void f1(vector<int>&);          // common style
void f2(const vector<int>&);    // common style
void f3(vector<int>);           // rare style

void h()
{
    vector<int> v(10000);

    // ...

    f1(v);    // pass a reference
    f2(v);    // pass a reference
    f3(v);    // copy the 10000 elements into a new vector for f3() to use
}
```

The ***assign*** functions exist to provide counterparts to the multi-argument constructors. They are needed because = takes a single right-hand operand, so ***assign*()** is used where a default argument value or a range of values is needed. For example:

```
class Book {
    // ...
};

void f(vector<Num>& vn, vector<char>& vc, vector<Book>& vb, list<Book>& lb)
{
    vn.assign(10,Num(0));              // assign vector of 10 copies of Num(0) to vn

    char s[] = "literal";
    vc.assign(s,&s[sizeof(s)-1]);      // assign "literal" to vc

    vb.assign(lb.begin(),lb.end());    // assign list elements

    // ...
}
```

Thus, we can initialize a ***vector*** with any sequence of its element type and similarly assign any such sequence. Importantly, this is done without explicitly introducing a multitude of constructors and conversion functions. Note that assignment completely changes the elements of a vector. Conceptually, all old elements are erased and the new ones are inserted. After assignment, the size of a ***vector*** is the number of elements assigned. For example:

```
void f()
{
    vector<char> v(10,´x´);        // v.size()==10, each element has the value 'x'
    v.assign(5,´a´);               // v.size()==5, each element has the value 'a'
    // ...
}
```

Naturally, what ***assign*()** does could be done indirectly by first creating a suitable ***vector*** and then assigning that. For example:

```
void f2(vector<Book>& vh, list<Book>& lb)
{
    vector<Book> vt(lb.begin(),lb.end());
    vh = vt;
    // ...
}
```

However, this can be both ugly and inefficient.

Constructing a ***vector*** with two arguments of the same type can lead to an apparent ambiguity:

```
vector<int> v(10,50);    // vector(size,value) or vector(iterator1,iterator2)? vector(size,value)!
```

However, an ***int*** isn't an iterator and the implementation must ensure that this actually invokes

```
vector(vector<int>::size_type, const int&, const vector<int>::allocator_type&);
```

rather than

```
vector(vector<int>::iterator, vector<int>::iterator, const vector<int>::allocator_type&);
```

The library achieves this by suitable overloading of the constructors and handles the equivalent ambiguities for ***assign***( ) and ***insert***( ) (§16.3.6) similarly.

16.3.5 Stack Operations [org.stack]

Most often, we think of a ***vector*** as a compact data structure that we can index to access elements. However, we can ignore this concrete notion and view ***vector*** as an example of the more abstract notion of a sequence. Looking at a ***vector*** this way, and observing common uses of arrays and ***vector***s, it becomes obvious that stack operations make sense for a ***vector***:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // stack operations:

    void push_back(const T& x);    // add to end
    void pop_back();               // remove last element
    // ...
};
```

These functions treat a ***vector*** as a stack by manipulating its end. For example:

```
void f(vector<char>& s)
{
    s.push_back('a');
    s.push_back('b');
    s.push_back('c');
    s.pop_back();
    if (s[s.size()-1] != 'b') error("impossible!");
    s.pop_back();
    if (s.back() != 'a') error("should never happen!");
}
```

Each time *push_back*( ) is called, the *vector s* grows by one element and that element is added at the end. So *s*[*s*.*size*( )-*1*], also known as *s*.*back*( ) (§16.3.3), is the element most recently pushed onto the *vector*.

Except for the word *vector* instead of *stack*, there is nothing unusual in this. The suffix *_back* is used to emphasize that elements are added to the end of the *vector* rather than to the beginning. Adding an element to the end of a *vector* could be an expensive operation because extra memory needs to be allocated to hold it. However, an implementation must ensure that repeated stack operations incur growth-related overhead only infrequently.

Note that *pop_back*( ) does not return a value. It just pops, and if we want to know what was on the top of the stack before the pop, we must look. This happens not to be my favorite style of stack (§2.5.3, §2.5.4), but it's arguably more efficient and it's the standard.

Why would one do stack-like operations on a *vector*? An obvious reason is to implement a *stack* (§17.3.1), but a more common reason is to construct a *vector* incrementally. For example, we might want to read a *vector* of points from input. However, we don't know how many points will be read, so we can't allocate a vector of the right size and then read into it. Instead, we might write:

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;

    while (cin >> buf) {
        if (buf == sentinel) return;
        // check new point
        cities.push_back(buf);
    }
}
```

This ensures that the *vector* expands as needed. If all we needed to do with a new point were to put it into the *vector*, we might have initialized *cities* directly from input in a constructor (§16.3.4). However, it is common to do a bit of processing on input and expand a data structure gradually as a program progresses; *push_back*( ) supports that.

In C programs, this is one of the most common uses of the C standard library function *realloc*( ). Thus, *vector* – and, in general, any standard container – provides a more general, more elegant, and no less efficient alternative to *realloc*( ).

The *size*( ) of a *vector* is implicitly increased by *push_back*( ) so the *vector* cannot overflow (as long as there is memory available to acquire; see §19.4.1). However, a *vector* can underflow:

```
void f()
{
    vector<int> v;
    v.pop_back();        // undefined effect: the state of v becomes undefined
    v.push_back(7);      // undefined effect (the state of v is undefined), probably bad
}
```

The effect of underflow is undefined, but the obvious implementation of *pop_back*( ) causes memory not owned by the *vector* to be overwritten. Like overflow, underflow must be avoided.

16.3.6 List Operations [org.list]

The *push_back*( ), *pop_back*( ), and *back*( ) operations (§16.3.5) allow a *vector* to be used effectively as a stack. However, it is sometimes also useful to add elements in the middle of a *vector* and to remove elements from a *vector*:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // list operations:

    iterator insert(iterator pos, const T& x);          // add x before 'pos'
    void insert(iterator pos, size_type n, const T& x);
    template <class In>                  // In must be an input iterator (§19.2.1)
        void insert(iterator pos, In first, In last);   // insert elements from sequence

    iterator erase(iterator pos);                       // remove element at pos
    iterator erase(iterator first, iterator last);      // erase sequence
    void clear();                                       // erase all elements

    // ...
};
```

To see how these operations work, let's do some (nonsensical) manipulation of a *vector* of names of fruit. First, we define the *vector* and populate it with some names:

```
vector<string> fruit;

fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
```

If I take a dislike to fruits whose names start with the letter *p*, I can remove those names like this:

```
sort(fruit.begin(), fruit.end());
vector<string>::iterator p1 = find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::iterator p2 = find_if(p1, fruit.end(), initial_not('p'));
fruit.erase(p1, p2);
```

In other words, sort the *vector*, find the first and the last fruit with a name that starts with the letter *p*, and erase those elements from *fruit*. How to write predicate functions such as *initial*(*x*) (is the initial letter *x*?) and *initial_not*( ) (is the initial letter different from *p*?) is explained in §18.4.2.

The *erase*(*p1*, *p2*) operation removes elements starting from *p1* up to and not including *p2*. This can be illustrated graphically:

```
fruit[]:
                                p1          p2
                                |           |
                                v           v
    apple   grape   kiwifruit   peach   pear   starfruit
```

The *erase*(*p1*, *p2*) removes *peach* and *pear*, yielding:

```
fruit[]:
    apple    grape    kiwifruit    starfruit
```

As usual, the sequence specified by the user is from the beginning to one-past-the-end of the sequence affected by the operation.

It would be tempting to write:

```
vector<string>::iterator  p1 = find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::reverse_iterator  p2 = find_if(fruit.rbegin(), fruit.rend(), initial('p'));
fruit.erase(p1, p2+1);      // oops!: type error
```

However, *vector<fruit>::iterator* and *vector<fruit>::reverse_iterator* need not be the same type, so we couldn't rely on the call of *erase*() to compile. To be used with an *iterator*, a *reverse_iterator* must be explicitly converted:

```
fruit.erase(p1, p2.base());      // extract iterator from reverse_iterator (§19.2.5)
```

Erasing an element from a *vector* changes the size of the *vector*, and the elements after the erased elements are copied into the freed positions. In this example, *fruit.size*() becomes *4* and the *starfruit* that used to be *fruit*[*5*] is now *fruit*[*3*].

Naturally, it is also possible to *erase*() a single element. In that case, only an iterator for that element is needed (rather than a pair of iterators). For example,

```
fruit.erase(find(fruit.begin(), fruit.end(), "starfruit"));
fruit.erase(fruit.begin()+1);
```

gets rid of the *starfruit* and the *grape*, thus leaving *fruit* with two elements:

```
fruit[]:
    apple    kiwifruit
```

It is also possible to insert elements into a vector. For example:

```
fruit.insert(fruit.begin()+1, "cherry");
fruit.insert(fruit.end(), "cranberry");
```

The new element is inserted before the position mentioned, and the elements from there to the end are moved to make space. We get:

```
fruit[]:
    apple    cherry    kiwifruit    cranberry
```

Note that *f.insert*(*f.end*(), *x*) is equivalent to *f.push_back*(*x*).

We can also insert whole sequences:

```
fruit.insert(fruit.begin()+2, citrus.begin(), citrus.end());
```

If *citrus* is a container

```
citrus[]:
    lemon   grapefruit   orange   lime
```

we get:

```
fruit[]:
    apple   cherry   lemon   grapefruit   orange   lime   kiwifruit   cranberry
```

The elements of *citrus* are copied into *fruit* by *insert*( ). The value of *citrus* is unchanged.

Clearly, *insert*( ) and *erase*( ) are more general than are operations that affect only the tail end of a *vector* (§16.3.5). They can also be more expensive. For example, to make room for a new element, *insert*( ) may have to reallocate every element to a new part of memory. If insertions into and deletions from a container are common, maybe that container should be a *list* rather than a *vector*. A *list* is optimized for *insert*( ) and *erase*( ) rather than for subscripting (§16.3.3).

Insertion into and erasure from a *vector* (but not a *list* or an associative container such as *map*) potentially move elements around. Consequently, an iterator pointing to an element of a *vector* may after an *insert*( ) or *erase*( ) point to another element or to no element at all. Never access an element through an invalid iterator; the effect is undefined and quite likely disastrous. In particular, beware of using the iterator that was used to indicate where an insertion took place; *insert*( ) makes its first argument invalid. For example:

```
void duplicate_elements(vector<string>& f)
{
    for(vector<string>::iterator p = f.begin(); p!=f.end(); ++p) f.insert(p, *p); // No!
}
```

Just think of it (§16.5[15]). A *vector* implementation would move all elements – or at least all elements after *p* – to make room for the new element.

The operation *clear*( ) erases all elements of a container. Thus, *c.clear*( ) is a shorthand for *c.erase*(*c.begin*( ), *c.end*( )). After *c.clear*( ), *c.size*( ) is *0*.

16.3.7 Addressing Elements [org.addressing]

Most often, the target of an *erase*( ) or *insert*( ) is a well-known place (such as *begin*( ) or *end*( )), the result of a search operation (such as *find*( )), or a location found during an iteration. In such cases, we have an iterator pointing to the relevant element. However, we often refer to elements of a *vector* by subscripting. How do we get an iterator suitable as an argument for *erase*( ) or *insert*( ) for the element with index *7* of a container *c*? Since that element is the 7th element after the beginning, *c.begin*( )+*7* is a good answer. Other alternatives that may seem plausible by analogy to arrays should be avoided. Consider:

```
template<class C> void f(C& c)
{
    c.erase(c.begin()+7);          // ok
    c.erase(&c[7]);                // not general
```

```
        c.erase(c+7);                        // error: adding 7 to a container makes no sense
        c.erase(c.back());                   // error: c.back() is a reference, not an iterator
        c.erase(c.end()-2);                  // ok (second to last element)
        c.erase(c.rbegin()+2);               // error: vector::reverse_iterator and vector::iterator
                                             // are different types
        c.erase((c.rbegin()+2).base());      // obscure, but ok (see §19.2.5)
}
```

The most tempting alternative, `&c[7]`, actually happens to work with the obvious implementation of *vector*, where `c[7]` refers directly to the element and its address is a valid iterator. However, this is not true for other containers. For example, a *list* or *map* iterator is almost certainly not a simple pointer to an element. Consequently, their iterators do not support `[]`. Therefore, `&c[7]` would be an error that the compiler catches.

The alternatives `c+7` and `c.back()` are simple type errors. A container is not a numeric variable to which we can add *7*, and `c.back()` is an element with a value like `"pear"` that does not identify the pear's location in the container *c*.

16.3.8 Size and Capacity [org.size]

So far, *vector* has been described with minimal reference to memory management. A *vector* grows as needed. Usually, that is all that matters. However, it is possible to ask directly about the way a *vector* uses memory, and occasionally it is worthwhile to affect it directly. The operations are:

```
template <class T, class A = allocator<T> > class vector {
public:
        // ...
        // capacity:

        size_type size() const;                      // number of elements
        bool empty() const { return size()==0; }
        size_type max_size() const;                  // size of the largest possible vector
        void resize(size_type sz, T val = T());      // added elements initialized by val

        size_type capacity() const;      // size of the memory (in number of elements) allocated
        void reserve(size_type n);       // make room for a total of n elements; don't initialize
                                         // throw a length_error if n>max_size()
        // ...
};
```

At any given time, a *vector* holds a number of elements. This number can be obtained by calling `size()` and can be changed using `resize()`. Thus, a user can determine the size of a vector and change it if it seems insufficient or excessive. For example:

```
class Histogram {
        vector<int> count;
public:
        Histogram(int h) : count(max(h,8)) {}
        void record(int i);
        // ...
};
```

```
void Histogram::record(int i)
{
    if (i<0) i = 0;
    if (count.size()<=i) count.resize(i+i);    // make lots of room
    count[i]++;
}
```

Using *resize*( ) on a *vector* is very similar to using the C standard library function *realloc*( ) on a C array allocated on the free store.

When a *vector* is resized to accommodate more (or fewer) elements, all of its elements may be moved to new locations. Consequently, it is a bad idea to keep pointers to elements in a *vector* that might be resized; after *resize*( ), such pointers could point to deallocated memory. Instead, we can keep indices. Note that *push_back*( ), *insert*( ), and *erase*( ) implicitly resize a *vector*.

In addition to the elements held, an application may keep some space for potential expansion. A programmer who knows that expansion is likely can tell the *vector* implementation to *reserve*( ) space for future expansion. For example:

```
struct Link {
    Link* next;
    Link(Link* n =0) : next(n) {}
    // ...
};

vector<Link> v;

void chain(size_t n) // fill v with n Links so that each Link points to its predecessor
{
    v.reserve(n);
    v.push_back(Link(0));
    for (int i = 1; i<n; i++) v.push_back(Link(&v[i-1]));
    // ...
}
```

A call *v.reserve*(*n*) ensures that no allocation will be needed when the size of *v* is increased until *v.size*( ) exceeds *n*.

Reserving space in advance has two advantages. First, even a simple-minded implementation can then allocate sufficient space in one operation rather than slowly acquiring enough memory along the way. However, in many cases there is a logical advantage that outweighs the potential efficiency gain. The elements of a container are potentially relocated when a *vector* grows. Thus, the links built between the elements of *v* in the previous example are guaranteed only because the call of *reserve*( ) ensures that there are no allocations while the vector is being built. That is, in some cases *reserve*( ) provides a guarantee of correctness in addition to whatever efficiency advantages it gives.

That same guarantee can be used to ensure that potential memory exhaustion and potentially expensive reallocation of elements take place at predictable times. For programs with stringent real-time constraints, this can be of great importance.

Note that *reserve*( ) doesn't change the size of a *vector*. Thus, it does not have to initialize any new elements. In both respects, it differs from *resize*( ).

In the same way as *size*( ) gives the current number of elements, *capacity*( ) gives the current number of reserved memory slots; *c*.*capacity*( )-*c*.*size*( ) is the number of elements that can be inserted without causing reallocation.

Decreasing the size of a *vector* doesn't decrease its capacity. It simply leaves room for the *vector* to grow into later. If you want to give memory back to the system, assign a new value to the *vector*. For example:

```
v = vector<int>(4,99);
```

A *vector* gets the memory it needs for its elements by calling member functions of its allocator (supplied as a template parameter). The default allocator, called *allocator* (§19.4.1), uses *new* to obtain storage so that it will throw *bad_alloc* if no more storage is obtainable. Other allocators can use different strategies (see §19.4.2).

The *reserve*( ) and *capacity*( ) functions are unique to *vector* and similar compact containers. Containers such as *list* do not provide equivalents.

16.3.9 Other Member Functions [org.etc]

Many algorithms – including important sort algorithms – involve swapping elements. The obvious way of swapping (§13.5.2) simply copies elements. However, a *vector* is typically implemented with a structure that acts as a handle (§13.5, §17.1.3) to the elements. Thus, two *vector*s can be swapped much more efficiently by interchanging the handles; *vector*::*swap*( ) does that. The time difference between this and the default *swap*( ) is orders of magnitude in important cases:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...

    void swap(vector&);

    allocator_type get_allocator() const;
};
```

The *get_allocator*( ) function gives the programmer a chance to get hold of a *vector*'s allocator (§16.3.1, §16.3.4). Typically, the reason for this is to ensure that data from an application that is related to a *vector* is allocated similarly to the *vector* itself (§19.4.1).

16.3.10 Helper Functions [org.algo]

Two *vector*s can be compared using == and <:

```
template <class T, class A>
bool std::operator==(const vector<T,A>& x, const vector<T,A>& y);

template <class T, class A>
bool std::operator<(const vector<T,A>& x, const vector<T,A>& y);
```

Two *vector*s *v1* and *v2* compare equal if *v1*.*size*( )==*v2*.*size*( ) and *v1*[*n*]==*v2*[*n*] for every valid index *n*. Similarly, < is a lexicographical ordering. In other words, < for *vector*s could be defined like this:

```
template <class T, class A>
inline bool std::operator<(const vector<T,A>& x, const vector<T,A>& y)
{
    return lexicographical_compare(x.begin(),x.end(),y.begin(),y.end()); // see §18.9
}
```

This means that *x* is less than *y* if the first element *x*[*i*] that is not equal to the corresponding element *y*[*i*] is less than *y*[*i*], or *x.size*()<*y.size*() with every *x*[*i*] equal to its corresponding *y*[*i*].

The standard library also provides !=, <=, >, and >=, with definitions that correspond to those of == and <.

Because *swap*() is a member, it is called using the *v1.swap*(*v2*) syntax. However, not every type has a *swap*() member, so generic algorithms use the conventional *swap*(*a*,*b*) syntax. To make that work for *vector*s also, the standard library provides the specialization:

```
template <class T, class A> void std::swap(vector<T,A>& x, vector<T,A>& y)
{
    x.swap(y);
}
```

16.3.11 Vector<bool> [org.vector.bool]

The specialization (§13.5) *vector<bool>* is provided as a compact *vector* of *bool*. A *bool* variable is addressable, so it takes up at least one byte. However, it is easy to implement *vector<bool>* so that each element takes up only a bit.

The usual *vector* operations work for *vector<bool>* and retain their usual meanings. In particular, subscripting and iteration work as expected. For example:

```
void f(vector<bool>& v)
{
    for (int i = 0; i<v.size(); ++i) cin >> v[i];          // iterate using subscripting

    typedef vector<bool>::const_iterator VI;
    for (VI p = v.begin(); p!=v.end(); ++p) cout<<*p;      // iterate using iterators
}
```

To achieve this, an implementation must simulate addressing of a single bit. Since a pointer cannot address a unit of memory smaller than a byte, *vector<bool>::iterator* cannot be a pointer. In particular, one cannot rely on *bool** as an iterator for a *vector<bool>*:

```
void f(vector<bool>& v)
{
    bool* p = v.begin(); // error: type mismatch
    // ...
}
```

A technique for addressing a single bit is outlined in §17.5.3.

The library also provides *bitset* as a set of Boolean values with Boolean set operations (§17.5.3).

16.4 Advice [org.advice]

[1] Use standard library facilities to maintain portability; §16.1.
[2] Don't try to redefine standard library facilities; §16.1.2.
[3] Don't believe that the standard library is best for everything.
[4] When building a new facility, consider whether it can be presented within the framework offered by the standard library; §16.3.
[5] Remember that standard library facilities are defined in namespace ***std***; §16.1.2.
[6] Declare standard library facilities by including its header, not by explicit declaration; §16.1.2.
[7] Take advantage of late abstraction; §16.2.1.
[8] Avoid fat interfaces; §16.2.2.
[9] Prefer algorithms with reverse iterators over explicit loops dealing with reverse order; §16.3.2.
[10] Use ***base***( ) to extract an ***iterator*** from a ***reverse_iterator***; §16.3.2.
[11] Pass containers by reference; §16.3.4.
[12] Use iterator types, such as ***list<char>::iterator***, rather than pointers to refer to elements of a container; §16.3.1.
[13] Use ***const*** iterators where you don't need to modify the elements of a container; §16.3.1.
[14] Use ***at***( ), directly or indirectly, if you want range checking; §16.3.3.
[15] Use ***push_back***( ) or ***resize***( ) on a container rather than ***realloc***( ) on an array; §16.3.5.
[16] Don't use iterators into a resized ***vector***; §16.3.8.
[17] Use ***reserve***( ) to avoid invalidating iterators; §16.3.8.
[18] When necessary, use ***reserve***( ) to make performance predictable; §16.3.8.

16.5 Exercises [org.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

1. (∗1.5) Create a ***vector<char>*** containing the letters of the alphabet in order. Print the elements of that vector in order and in reverse order.
2. (∗1.5) Create a ***vector<string>*** and read a list of names of fruits from ***cin*** into it. Sort the list and print it.
3. (∗1.5) Using the ***vector*** from §16.5[2], write a loop to print the names of all fruits with the initial letter ***a***.
4. (∗1) Using the ***vector*** from §16.5[2], write a loop to delete all fruits with the initial letter ***a***.
5. (∗1) Using the ***vector*** from §16.5[2], write a loop to delete all citrus fruits.
6. (∗1.5) Using the ***vector*** from §16.5[2], write a loop to delete all fruits that you don't like.
7. (∗2) Complete the ***Vector***, ***List***, and ***Itor*** classes from §16.2.1.
8. (∗2.5) Given an ***Itor*** class, consider how to provide iterators for forwards iteration, backwards iteration, iteration over a container that might change during an iteration, and iteration over an immutable container. Organize this set of containers so that a user can interchangeably use iterators that provide sufficient functionality for an algorithm. Minimize replication of effort in the implementation of the containers. What other kinds of iterators might a user need? List the strengths and weaknesses of your approach.

9. (∗2) Complete the ***Container***, ***Vector***, and ***List*** classes from §16.2.2.
10. (∗2.5) Generate 10,000 uniformly distributed random numbers in the range 0 to 1,023 and store them in (a) an standard library ***vector***, (b) a ***Vector*** from §16.5[7], and (3) a ***Vector*** from §16.5[9]. In each case, calculate the arithmetic mean of the elements of the vector (as if you didn't know it already). Time the resulting loops. Estimate, measure, and compare the memory consumption for the three styles of vectors.
11. (∗1.5) Write an iterator to allow ***Vector*** from §16.2.2 to be used as a container in the style of §16.2.1.
12. (∗1.5) Write a class derived from ***Container*** to allow ***Vector*** from §16.2.1 to be used as a container in the style of §16.2.2.
13. (∗2) Write classes to allow ***Vector*** from §16.2.1 and ***Vector*** from §16.2.2 to be used as standard containers.
14. (∗2) Write a template that implements a container with the same member functions and member types as the standard ***vector*** for an existing (nonstandard, non-student-exercise) container type. Do not modify the (pre)existing container type. How would you deal with functionality offered by the nonstandard ***vector*** but not by the standard ***vector***?
15. (∗1.5) Outline the possible behavior of ***duplicate_elements*( )** from §16.3.6 for a ***vector<string>*** with the three elements ***don´t do this***.

17

Standard Containers

Now is a good time to put your work
on a firm theoretical foundation.
– Sam Morgan

Standard containers — container and operation summaries — efficiency — representation — element requirements — sequences — ***vector*** — ***list*** — ***deque*** — adapters — ***stack*** — ***queue*** — ***priority_queue*** — associative containers — ***map*** — comparisons — ***multimap*** — ***set*** — ***multiset*** — ''almost containers'' — ***bitset*** — arrays — hash tables — implementing a ***hash_map*** — advice — exercises.

17.1 Standard Containers [cont.intro]

The standard library defines two kinds of containers: sequences and associative containers. The sequences are all much like ***vector*** (§16.3). Except where otherwise stated, the member types and functions mentioned for ***vector*** can also be used for any other container and produce the same effect. In addition, associative containers provide element access based on keys (§3.7.4).

Built-in arrays (§5.2), ***string***s (Chapter 20), ***valarray***s (§22.4), and ***bitset***s (§17.5.3) hold elements and can therefore be considered containers. However, these types are not fully-developed standard containers. If they were, that would interfere with their primary purpose. For example, a built-in array cannot both hold its own size and remain layout-compatible with C arrays.

A key idea for the standard containers is that they should be logically interchangeable wherever reasonable. The user can then choose between them based on efficiency concerns and the need for specialized operations. For example, if lookup based on a key is common, a ***map*** (§17.4.1) can be used. On the other hand, if general list operations dominate, a ***list*** (§17.2.2) can be used. If many additions and removals of elements occur at the ends of the container, a ***deque*** (double-ended queue, §17.2.3), a ***stack*** (§17.3.1), or a ***queue*** (§17.3.2) should be considered. In addition, a user can design additional containers to fit into the framework provided by the standard containers

(§17.6). By default, a *vector* (§16.3) should be used; it will be implemented to perform well over a wide range of uses.

The idea of treating different kinds of containers – and more generally all kinds of information sources – in uniform ways leads to the notion of generic programming (§2.7.2, §3.8). The standard library provides many generic algorithms to support this idea (Chapter 18). Such algorithms can save the programmer from having to deal directly with details of individual containers.

17.1.1 Operations Summary [cont.operations]

This section lists the common and almost common members of the standard containers. For more details, read your standard headers (*<vector>*, *<list>*, *<map>*, etc.; §16.1.2).

Member Types (§16.3.1)	
value_type	Type of element.
allocator_type	Type of memory manager.
size_type	Type of subscripts, element counts, etc.
difference_type	Type of difference between iterators.
iterator	Behaves like *value_type**.
const_iterator	Behaves like *const value_type**.
reverse_iterator	View container in reverse order; like *value_type**.
const_reverse_iterator	View container in reverse order; like *const value_type**.
reference	Behaves like *value_type&*.
const_reference	Behaves like *const value_type&*.
key_type	Type of key (for associative containers only).
mapped_type	Type of *mapped_value* (for associative containers only).
key_compare	Type of comparison criterion (for associative containers only).

A container can be viewed as a sequence either in the order defined by the container's *iterator* or in reverse order. For an associative container, the order is based on the container's comparison criterion (by default <):

Iterators (§16.3.2)	
begin()	Points to first element.
end()	Points to one-past-last element.
rbegin()	Points to first element of reverse sequence.
rend()	Points to one-past-last element of reverse sequence.

Some elements can be accessed directly:

Element Access (§16.3.3)	
front()	First element.
back()	Last element.
[]	Subscripting, unchecked access (not for list).
at()	Subscripting, checked access (not for list).

Most containers provide efficient operations at the end (back) of their sequence of elements. In addition, lists and deques provide the equivalent operations on the start (front) of their sequences:

Stack and Queue Operations (§16.3.5, §17.2.2.2)	
push_back()	Add to end.
pop_back()	Remove last element.
push_front()	Add new first element (for list and deque only).
pop_front()	Remove first element (for list and deque only).

Containers provide list operations:

List Operations (§16.3.6)	
insert(p,x)	Add *x* before *p*.
insert(p,n,x)	Add *n* copies of *x* before *p*.
insert(p,first,last)	Add elements from [*first*:*last*[before *p*.
erase(p)	Remove element at *p*.
erase(first,last)	Erase [*first*:*last*[.
clear()	Erase all elements.

All containers provide operations related to the number of elements and a few other operations:

Other Operations (§16.3.8, §16.3.9, §16.3.10)	
size()	Number of elements.
empty()	Is the container empty?
max_size()	Size of the largest possible container.
capacity()	Space allocated for *vector* (for vector only).
reserve()	Reserve space for future expansion (for vector only).
resize()	Change size of container (for vector, list, and deque only).
swap()	Swap elements of two containers.
get_allocator()	Get a copy of the container's allocator.
==	Is the content of two containers the same?
!=	Is the content of two containers different?
<	Is one container lexicographically before another?

Containers provide a variety of constructors and assignment operations:

Constructors, etc. (§16.3.4)	
container()	Empty container.
container(n)	*n* elements default value (not for associative containers).
container(n,x)	*n* copies of *x* (not for associative containers).
container(first,last)	Initial elements from [*first*:*last*[.
container(x)	Copy constructor; initial elements from container *x*.
~container()	Destroy the container and all of its elements.

Assignments (§16.3.4)	
operator=(x)	Copy assignment; elements from container ***x***.
assign(n,x)	Assign ***n*** copies of ***x*** (not for associative containers).
assign(first,last)	Assign from [***first***:***last***[.

Associative containers provide lookup based on keys:

Associative Operations (§17.4.1)	
operator[](k)	Access the element with key ***k*** (for containers with unique keys).
find(k)	Find the element with key ***k***.
lower_bound(k)	Find the first element with key ***k***.
upper_bound(k)	Find the first element with key greater than ***k***.
equal_range(k)	Find the ***lower_bound*** and ***upper_bound*** of elements with key ***k***.
key_comp()	Copy of the key comparison object.
value_comp()	Copy of the ***mapped_value*** comparison object.

In addition to these common operations, most containers provide a few specialized operations.

17.1.2 Container Summary [cont.summary]

The standard containers can be summarized like this:

Standard Container Operations					
	[]	List Operations	Front Operations	Back (Stack) Operations	Iterators
	§16.3.3 §17.4.1.3	§16.3.6 §20.3.9	§17.2.2.2 §20.3.9	§16.3.5 §20.3.12	§19.2.1
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
deque	const	O(n)	const	const	Ran
stack				const+	
queue			const	const+	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
valarray	const				Ran
bitset	const				

In the *iterators* column, ***Ran*** means random-access iterator and ***Bi*** means bidirectional iterator; the operations for a bidirectional operator are a subset of those of a random-access iterator (§19.2.1).

Other entries are measures of the efficiency of the operations. A ***const*** entry means the operation takes an amount of time that does not depend on the number of elements in the container. Another conventional notation for *constant time* is $O(1)$. An $O(n)$ entry means the entry takes time proportional to the number of elements involved. A + suffix indicates that occasionally a significant extra cost is incurred. For example, inserting an element into a ***list*** has a fixed cost (so it is listed as ***const***), whereas the same operation on a ***vector*** involves moving the elements following the insertion point (so it is listed as $O(n)$). Occasionally, all elements must be relocated (so I added a +). The ''big O'' notation is conventional. I added the + for the benefit of programmers who care about predictability in addition to average performance. A conventional term for $O(n)+$ is *amortized linear time*.

Naturally, if a constant is large it can dwarf a small cost proportional to the number of elements. However, for large data structures ***const*** tends to mean ''cheap,'' $O(n)$ to mean ''expensive,'' and $O(log(n))$ to mean ''fairly cheap.'' For even moderately large values of ***n***, $O(log(n))$ is closer to constant time than to $O(n)$. People who care about cost must take a closer look. In particular, they must understand what elements are counted to get the ***n***. No basic operation is ''very expensive,'' that is, $O(n*n)$ or worse.

Except for ***string***, the measures of costs listed here reflect requirements in the standard. The ***string*** estimates are my assumptions.

These measures of complexity and cost are upper bounds. The measures exist to give users some guidance as to what they can expect from implementations. Naturally, implementers will try to do better in important cases.

17.1.3 Representation [cont.rep]

The standard doesn't prescribe a particular representation for each standard container. Instead, the standard specifies the container interfaces and some complexity requirements. Implementers will choose appropriate and often cleverly optimized implementations to meet the general requirements. A container will almost certainly be represented by a data structure holding the elements accessed through a handle holding size and capacity information. For a ***vector***, the element data structure is most likely an array:

Similarly, a ***list*** is most likely represented by a set of links pointing to the elements:

A ***map*** is most likely implemented as a (balanced) tree of nodes pointing to (key,value) pairs:

A ***string*** might be implemented as outlined in §11.12 or maybe as a sequence of arrays holding a few characters each:

17.1.4 Element Requirements [cont.elem]

Elements in a container are copies of the objects inserted. Thus, to be an element of a container, an object must be of a type that allows the container implementation to copy it. The container may copy elements using a copy constructor or an assignment; in either case, the result of the copy must be an equivalent object. This roughly means that any test for equality that you can devise on the value of the objects must deem the copy equal to the original. In other words, copying an element must work much like an ordinary copy of built-in types (including pointers). For example,

```
X& X::operator=(const X& a)    // proper assignment operator
{
        // copy all of a's members to *this
        return *this;
}
```

makes ***X*** acceptable as an element type for a standard container, but

```
void Y::operator=(const Y& a) // improper assignment operator
{
        // zero out all of a's members
}
```

renders ***Y*** unsuitable because ***Y***'s assignment has neither the conventional return type nor the conventional semantics.

Some violations of the rules for standard containers can be detected by a compiler, but others cannot and might then cause unexpected behavior. For example, a copy operation that throws an exception might leave a partially copied element behind. It could even leave the container itself in a state that could cause trouble later. Such copy operations are themselves bad design (§14.4.6.1).

When copying elements isn't right, the alternative is to put pointers to objects into containers instead of the objects themselves. The most obvious example is polymorphic types (§2.5.4, §12.2.6). For example, we use *vector<Shape*>* rather than *vector<Shape>* to preserve polymorphic behavior.

17.1.4.1 Comparisons [cont.comp]

Associative containers require that their elements can be ordered. So do many operations that can be applied to containers (for example *sort*( )). By default, the < operator is used to define the order. If < is not suitable, the programmer must provide an alternative (§17.4.1.5, §18.4.2). The ordering criterion must define a *strict weak ordering*. Informally, this means that both less-than and equality must be transitive. That is, for an ordering criterion *cmp*:

[1] *cmp*(*x*,*x*) is *false*.
[2] If *cmp*(*x*,*y*) and *cmp*(*y*,*z*), then *cmp*(*x*,*z*).
[3] Define *equiv*(*x*,*y*) to be !(*cmp*(*x*,*y*)||*cmp*(*y*,*x*)). If *equiv*(*x*,*y*) and *equiv*(*y*,*z*), then *equiv*(*x*,*z*).

Consider:

```
template<class Ran> void sort(Ran first, Ran last);              // use < for comparison
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);// use cmp
```

The first version uses < and the second uses a user-supplied comparison *cmp*. For example, we might decide to sort *fruit* using a comparison that isn't case-sensitive. We do that by defining a function object (§11.9, §18.4) that does the comparison when invoked for a pair of *string*s:

```
class Nocase {          // case-insensitive string compare
public:
        bool operator()(const string&, const string&) const;
};

bool Nocase::operator()(const string& x, const string& y) const
        // return true if x is lexicographically less than y, not taking case into account
{
        string::const_iterator p = x.begin();
        string::const_iterator q = y.begin();

        while (p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q)) {
                ++p;
                ++q;
        }
        if (p == x.end()) return q != y.end();
        return toupper(*p) < toupper(*q);
}
```

We can call *sort*( ) using that comparison criterion. For example, given:

```
fruit:
    apple   pear   Apple   Pear   lemon
```

Sorting using *sort*(*fruit*.*begin*(), *fruit*.*end*(), *Nocase*()) would yield:

```
fruit:
    Apple   apple   lemon   Pear   pear
```

whereas plain *sort*(*fruit*.*begin*(), *fruit*.*end*()) would give:

```
fruit:
    Apple   Pear   apple   lemon   pear
```

assuming a character set in which uppercase letters precede lowercase letters.

Beware that < on C-style strings (that is, *char**) does not define lexicographical order (§13.5.2). Thus, associative containers will not work as most people would expect them to when C-style strings are used as keys. To make them work properly, a less-than operation that compares based on lexicographical order must be used. For example:

```
struct Cstring_less {
    bool operator()(const char* p, const char* q) const { return strcmp(p,q)<0; }
};
map<char*, int, Cstring_less> m;        // map that uses strcmp() to compare const char* keys
```

17.1.4.2 Other Relational Operators [cont.relops]

By default, containers and algorithms use < when they need to do a less-than comparison. When the default isn't right, a programmer can supply a comparison criterion. However, no mechanism is provided for also passing an equality test. Instead, when a programmer supplies a comparison *cmp*, equality is tested using two comparisons. For example:

```
if (x == y) // not done where the user supplied a comparison
if (!cmp(x,y) && !cmp(y,x)) // done where the user supplied a comparison cmp
```

This saves us from having to add an equality parameter to every associative container and most algorithms. It may look expensive, but the library doesn't check for equality very often, and in 50% of the cases, only a single call of *cmp*() is needed.

Using an equivalence relationship defined by less-than (by default <) rather than equality (by default ==) also has practical uses. For example, associative containers (§17.4) compare keys using an equivalence test !(*cmp*(*x*,*y*)||*cmp*(*y*,*x*)). This implies that equivalent keys need not be equal. For example, a *multimap* (§17.4.2) that uses case-insensitive comparison as its comparison criteria will consider the strings *Last*, *last*, *lAst*, *laSt*, and *lasT* equivalent, even though == for strings deems them different. This allows us to ignore differences we consider insignificant when sorting.

Given < and ==, we can easily construct the rest of the usual comparisons. The standard library defines them in the namespace *std*::*rel_ops* and presents them in <*utility*>:

```
template<class T> bool rel_ops::operator!=(const T& x, const T& y) { return !(x==y); }
template<class T> bool rel_ops::operator>(const T& x, const T& y) { return y<x; }
template<class T> bool rel_ops::operator<=(const T& x, const T& y) { return !(y<x); }
template<class T> bool rel_ops::operator>=(const T& x, const T& y) { return !(x<y); }
```

Placing these operations in ***rel_ops*** ensures that they are easy to use when needed, yet they don't get created implicitly unless extracted from that namespace:

```
void f()
{
    using namespace std;
    // !=, >, etc., not generated by default
}

void g()
{
    using namespace std;
    using namespace std::rel_ops;
    // !=, >, etc., generated by default
}
```

The `!=`, etc., operations are not defined directly in ***std*** because they are not always needed and sometimes their definition would interfere with user code. For example, if I were writing a generalized math library, I would want *my* relational operators and not the standard library versions.

17.2 Sequences [cont.seq]

Sequences follow the pattern described for ***vector*** (§16.3). The fundamental sequences provided by the standard library are:

vector ***list*** ***deque***

From these,

stack ***queue*** ***priority_queue***

are created by providing suitable interfaces. These sequences are called *container adapters*, *sequence adapters*, or simply *adapters* (§17.3).

17.2.1 Vector [cont.vector]

The standard ***vector*** is described in detail in §16.3. The facilities for reserving space (§16.3.8) are unique to ***vector***. By default, subscripting using `[]` is not range checked. If a check is needed, use ***at***`()` (§16.3.3), a checked vector (§3.7.1), or a checked iterator (§19.3). A ***vector*** provides random-access iterators (§19.2.1).

17.2.2 List [cont.list]

A ***list*** is a sequence optimized for insertion and deletion of elements. Compared to ***vector*** (and ***deque***; §17.2.3), subscripting would be painfully slow, so subscripting is not provided for ***list***. Consequently, ***list*** provides bidirectional iterators (§19.2.1) rather than random-access iterators. This implies that a ***list*** will typically be implemented using some form of a doubly-linked list (see §17.8[16]).

A ***list*** provides all of the member types and operations offered by ***vector*** (§16.3), with the exceptions of subscripting, ***capacity***( ), and ***reserve***( ):

```
template <class T, class A = allocator<T> > class std::list {
public:
        // types and operations like vector's, except [], at(), capacity(), and reserve()
        // ...
};
```

17.2.2.1 Splice, Sort, and Merge [cont.splice]

In addition to the general sequence operations, ***list*** provides several operations specially suited for list manipulation:

```
template <class T, class A = allocator<T> > class list {
public:
        // ...
        // list-specific operations:

        void splice(iterator pos, list& x);                  // move all elements from x to before
                                                             // pos in this list without copying.
        void splice(iterator pos, list& x, iterator p);      // move *p from x to before
                                                             // pos in this list without copying.
        void splice(iterator pos, list& x, iterator first, iterator last);

        void merge(list&);          // merge sorted lists
        template <class Cmp> void merge(list&, Cmp);

        void sort();
        template <class Cmp> void sort(Cmp);

        // ...
};
```

These ***list*** operations are all *stable*; that is, they preserve the relative order of elements that have equivalent values.

The ***fruit*** examples from §16.3.6 work with ***fruit*** defined to be a ***list***. In addition, we can extract elements from one list and insert them into another by a single ''splice'' operation. Given:

```
fruit:
        apple    pear

citrus:
        orange    grapefruit    lemon
```

we can splice the *orange* from *citrus* into *fruit* like this:

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));
fruit.splice(p, citrus, citrus.begin());
```

The effect is to remove the first element from *citrus* (*citrus.begin()*) and place it just before the first element of *fruit* with the initial letter *p*, thereby giving:

```
fruit:
    apple    orange    pear
citrus:
    grapefruit    lemon
```

Note that *splice()* doesn't copy elements the way *insert()* does (§16.3.6). It simply modifies the *list* data structures that refer to the element.

In addition to splicing individual elements and ranges, we can *splice()* all elements of a *list*:

```
fruit.splice(fruit.begin(), citrus);
```

This yields:

```
fruit:
    grapefruit    lemon    apple    orange    pear
citrus:
    <empty>
```

Each version of *splice()* takes as its second argument the *list* from which elements are taken. This allows elements to be removed from their original *list*. An iterator alone wouldn't allow that because there is no general way to determine the container holding an element given only an iterator to that element (§18.6).

Naturally, an iterator argument must be a valid iterator for the *list* into which it is supposed to point. That is, it must point to an element of that *list* or be the *list*'s *end()*. If not, the result is undefined and possibly disastrous. For example:

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));
fruit.splice(p, citrus, citrus.begin());      // ok
fruit.splice(p, citrus, fruit.begin());       // error: fruit.begin() doesn't point into citrus
citrus.splice(p, fruit, fruit.begin());       // error: p doesn't point into citrus
```

The first *splice()* is ok even though *citrus* is empty.

A *merge()* combines two sorted lists by removing the elements from one *list* and entering them into the other while preserving order. For example,

```
f1:
    apple    quince    pear
f2:
    lemon    grapefruit    orange    lime
```

can be sorted and merged like this:

```
f1.sort();
f2.sort();
f1.merge(f2);
```

This yields:

```
f1:
    apple   grapefruit   lemon   lime   orange   pear   quince
f2:
    <empty>
```

If one of the lists being merged is not sorted, *merge*( ) will still produce a list containing the union of elements of the two lists. However, there are no guarantees made about the order of the result.

Like *splice*( ), *merge*( ) refrains from copying elements. Instead, it removes elements from the source list and splices them into the target list. After an *x*.*merge*(*y*), the *y* list is empty.

17.2.2.2 Front Operations [cont.front]

Operations that refer to the first element of a *list* are provided to complement the operations referring to the last element provided by every sequence (§16.3.6):

```
template <class T, class A = allocator<T> > class list {
public:
    // ...
    // element access:

    reference front();                  // reference to first element
    const_reference front() const;

    void push_front(const T&);          // add new first element
    void pop_front();                   // remove first element

    // ...
};
```

The first element of a container is called its *front*. For a *list*, front operations are as efficient and convenient as back operations (§16.3.5). When there is a choice, back operations should be preferred over front operations. Code written using back operations can be used for a *vector* as well as for a *list*. So if there is a chance that the code written using a *list* will ever evolve into a generic algorithm applicable to a variety of containers, it is best to prefer the more widely available back operations. This is a special case of the rule that to achieve maximal flexibility, it is usually wise to use the minimal set of operations to do a task (§17.1.4.1).

17.2.2.3 Other Operations [cont.list.etc]

Insertion and removal of elements are particularly efficient for *list*s. This, of course, leads people to prefer *list*s when these operations are frequent. That, in turn, makes it worthwhile to support common ways of removing elements directly:

```
template <class T, class A = allocator<T> > class list {
public:
    // ...

    void remove(const T& val);
    template <class Pred> void remove_if(Pred p);

    void unique();                                      // remove duplicates using ==
    template <class BinPred> void unique(BinPred b);    // remove duplicates using b

    void reverse();                                     // reverse order of elements
};
```

For example, given

```
fruit:
    apple   orange   grapefruit   lemon   orange   lime   pear   quince
```

we can remove all elements with the value "*orange*" like this:

```
fruit.remove("orange");
```

yielding:

```
fruit:
    apple   grapefruit   lemon   lime   pear   quince
```

Often, it is more interesting to remove all elements that meet some criterion rather than simply all elements with a given value. The ***remove_if*( )** operation does that. For example,

```
fruit.remove_if(initial('l'));
```

removes every element with the initial *´l´* from ***fruit*** giving:

```
fruit:
    apple   grapefruit   pear   quince
```

A common reason for removing elements is to eliminate duplicates. The ***unique*( )** operation is provided for that. For example:

```
fruit.sort();
fruit.unique();
```

The reason for sorting is that ***unique*** removes only duplicates that appear consecutively. For example, had fruit contained:

```
apple   pear   apple   apple   pear
```

a simple ***fruit.unique*( )** would have produced

```
apple   pear   apple   pear
```

whereas sorting first gives:

```
apple   pear
```

If only certain duplicates should be eliminated, we can provide a predicate to specify which

duplicates we want to remove. For example, we might define a binary predicate (§18.4.2) *initial2*(*x*) to compare *string*s that have the initial *x* but yield *false* for every *string* that doesn't. Given:

pear pear apple apple

we can remove consecutive duplicates of every *fruit* with the initial *p* by a call

```
fruit.unique(initial2('p'));
```

This would give

pear apple apple

As noted in §16.3.2, we sometimes want to view a container in reverse order. For a *list*, it is possible to reverse the elements so that the first becomes the last, etc., without copying the elements. The *reverse*() operation is provided to do that. Given:

```
fruit:
    banana cherry lime strawberry
```

fruit.*reverse*() produces:

```
fruit:
    strawberry lime cherry banana
```

An element that is removed from a list is destroyed. However, note that destroying a pointer does not imply that the object it points to is *delete*d. If you want a container of pointers that *delete*s elements pointed to when the pointer is removed from the container or the container is destroyed, you must write one yourself (§17.8[13]).

17.2.3 Deque [cont.deque]

A *deque* (it rhymes with check) is a double-ended queue. That is, a *deque* is a sequence optimized so that operations at both ends are about as efficient as for a *list*, whereas subscripting approaches the efficiency of a *vector*:

```
template <class T, class A = allocator<T> > class std::deque {
    // types and operations like vector (§16.3.3, §16.3.5, §16.3.6)
    // plus front operations (§17.2.2.2) like list
};
```

Insertion and deletion of elements ''in the middle'' have *vector*-like (in)efficiencies rather than *list*-like efficiencies. Consequently, a *deque* is used where additions and deletions take place ''at the ends.'' For example, we might use a *deque* to model a section of a railroad or to represent a deck of cards in a game:

```
deque<car> siding_no_3;
deque<Card> bonus;
```

17.3 Sequence Adapters [cont.adapters]

The *vector*, *list*, and *deque* sequences cannot be built from each other without loss of efficiency. On the other hand, *stack*s and *queue*s can be elegantly and efficiently implemented using those three basic sequences. Therefore, *stack* and *queue* are defined not as separate containers, but as adaptors of basic containers.

A container adapter provides a restricted interface to a container. In particular, adapters do not provide iterators; they are intended to be used only through their specialized interfaces.

The techniques used to create a container adapter from a container are generally useful for non-intrusively adapting the interface of a class to the needs of its users.

17.3.1 Stack [cont.stack]

The *stack* container adapter is defined in <*stack*>. It is so simple that the best way to describe it is to present an implementation:

```
template <class T, class C = deque<T> > class std::stack {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit stack(const C& a = C()) : c(a) { }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

That is, a *stack* is simply an interface to a container of the type passed to it as a template argument. All *stack* does is to eliminate the non-stack operations on its container from the interface and give *back*( ), *push_back*( ), and *pop_back*( ) their conventional names: *top*( ), *push*( ), and *pop*( ).

By default, a *stack* makes a *deque* to hold its elements, but any sequence that provides *back*( ), *push_back*( ), and *pop_back*( ) can be used. For example:

```
stack<char> s1;                  // uses a deque<char> to store elements of type char
stack< int, vector<int> > s2;    // uses a vector<int> to store elements of type int
```

It is possible to supply an existing container to initialize a stack. For example:

```
void print_backwards(vector<int>& v)
{
    stack<int> state(v);   // initialize state from v
    while (state.size()) {
        cout << state.top();
        state.pop();
    }
}
```

However, the elements of a container argument are copied, so supplying an existing container can be expensive.

Elements are added to a ***stack*** using ***push_back()*** on the container that is used to store the elements. Consequently, a ***stack*** cannot overflow as long as there is memory available on the machine for the container to acquire (using its allocator; see §19.4).

On the other hand, a ***stack*** can underflow:

```
void f()
{
    stack<int> s;
    s.push(2);
    if (s.empty()) {                        // underflow is preventable
        // don't pop
    }
    else {                                  // but not impossible
        s.pop();  // fine: s.size() becomes 0
        s.pop();  // undefined effect, probably bad
    }
}
```

Note that one does not ***pop()*** an element to use it. Instead, the ***top()*** is accessed and then ***pop()***'d when it is no longer needed. This is not too inconvenient, and it is more efficient when the ***pop()*** isn't necessary:

```
void f(stack<char>& s)
{
    if (s.top()=='c') s.pop();     // remove optional initial 'c'
    // ...
}
```

Unlike fully developed containers, ***stack*** (like other container adapters) doesn't have an allocator template parameter. Instead, the ***stack*** and its users rely on the allocator from the container used to implement the ***stack***.

17.3.2 Queue [cont.queue]

Defined in ***<queue>***, a ***queue*** is an interface to a container that allows the insertion of elements at the ***back()*** and the extraction of elements at the ***front()***:

```
template <class T, class C = deque<T> > class std::queue {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit queue(const C& a = C()) : c(a) { }

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }

    value_type& back() { return c.back(); }
    const value_type& back() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

By default, a ***queue*** makes a ***deque*** to hold its elements, but any sequence that provides ***front*()**, ***back*()**, ***push_back*()**, and ***pop_front*()** can be used. Because a ***vector*** does not provide ***pop_front*()**, a ***vector*** cannot be used as the underlying container for a queue.

Queues seem to pop up somewhere in every system. One might define a server for a simple message-based system like this:

```
struct Message {
    // ...
};

void server(queue<Message>& q)
{
    while(!q.empty()) {
        Message& m = q.front();   // get hold of message
        m.service();              // call function to serve request
        q.pop();                  // destroy message
    }
}
```

Messages would be put on the ***queue*** using ***push*()**.

If the requester and the server are running in different processes or threads, some form of synchronization of the queue access would be necessary. For example:

```
void server2(queue<Message>& q, Lock& lck)
{
    while(!q.empty()) {
        Message m;
        {   LockPtr h(lck);          // hold lock only while extracting message (see §14.4.7)
            if (q.empty()) return;   // somebody else got the message
```

```
                m = q.front();
                q.pop();
            }
            m.service();                    // call function to serve request
        }
    }
```

There is no standard definition of concurrency or locking in C++ or in the world in general. Have a look to see what your system has to offer and how to access it from C++ (§17.8[8]).

17.3.3 Priority Queue [cont.pqueue]

A *priority_queue* is a queue in which each element is given a priority that controls the order in which the elements get to be *top*( ):

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
    C c;
    Cmp cmp;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
        : c(a2), cmp(a1) { }
    template <class In>
    priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    const value_type& top() const { return c.front(); }

    void push(const value_type&);
    void pop();
};
```

The declaration of *priority_queue* is found in <*queue*>.

By default, the *priority_queue* simply compares elements using the < operator and *pop*( ) returns the largest element:

```
struct Message {
    int priority;
    bool operator<(const Message& x) const { return priority < x.priority; }
    // ...
};
```

```
void server(priority_queue<Message>& q, Lock& lck)
{
    while(!q.empty()) {
        Message m;
        {   LockPtr h(lck);          // hold lock only while extracting message (see §14.4.7)
            if (q.empty()) return;   // somebody else got the message
            m = q.top();
            q.pop();
        }
        m.service();    // call function to serve request
    }
}
```

This example differs from the ***queue*** example (§17.3.2) in that ***messages*** with higher priority will get served first. The order in which elements with equal priority come to the head of the queue is not defined. Two elements are considered of equal priority if neither has higher priority than the other (§17.4.1.5).

An alternative to < for comparison can be provided as a template argument. For example, we could sort strings in a case-insensitive manner by placing them in

```
priority_queue<string,Nocase> pq;    // use Nocase::operator()() for comparisons (§17.1.4.1)
```

using ***pq.push()*** and then retrieving them using ***pq.top()*** and ***pq.pop()***.

Objects defined by templates given different template arguments are of different types (§13.6.3.1). For example:

```
void f(priority_queue<string>& pq1)
{
    pq = pq1;  // error: type mismatch
}
```

We can supply a comparison criterion without affecting the type of a ***priority_queue*** by providing a comparison object of the appropriate type as a constructor argument. For example:

```
struct String_cmp {   // type used to express comparison criteria at run time
    String_cmp(int n = 0);    // use comparison criteria n
    // ...
};

void g(priority_queue<string,String_cmp>& pq)
{
    priority_queue<string> pq2(String_cmp(nocase));
    pq = pq2;  // ok: pq and pq2 are of the same type, pq now also uses String_cmp(nocase)
}
```

Keeping elements in order isn't free, but it needn't be expensive either. One useful way of implementing a ***priority_queue*** is to use a tree structure to keep track of the relative positions of elements. This gives an $O(log(n))$ cost of both ***push()*** and ***pop()***.

By default, a ***priority_queue*** makes a ***vector*** to hold its elements, but any sequence that provides ***front()***, ***push_back()***, ***pop_back()***, and random iterators can be used. A ***priority_queue*** is most likely implemented using a ***heap*** (§18.8).

17.4 Associative Containers [cont.assoc]

An *associative array* is one of the most useful general, user-defined types. In fact, it is often a built-in type in languages primarily concerned with text processing and symbolic processing. An associative array, often called a *map* and sometimes called a *dictionary*, keeps pairs of values. Given one value, called the *key*, we can access the other, called the *mapped value*. An associative array can be thought of as an array for which the index need not be an integer:

```
template<class K, class V> class Assoc {
public:
    V& operator[](const K&); // return a reference to the V corresponding to K
    // ...
};
```

Thus, a key of type ***K*** names a mapped value of type ***V***.

Associative containers are a generalization of the notion of an associative array. The ***map*** is a traditional associative array, where a single value is associated with each unique key. A ***multimap*** is an associative array that allows duplicate elements for a given key, and ***set*** and ***multiset*** can be seen as degenerate associative arrays in which no value is associated with a key.

17.4.1 Map [cont.map]

A ***map*** is a sequence of (key,value) pairs that provides for fast retrieval based on the key. At most one value is held for each key; in other words, each key in a ***map*** is unique. A ***map*** provides bidirectional iterators (§19.2.1).

The ***map*** requires that a less-than operation exist for its key types (§17.1.4.1) and keeps its elements sorted so that iteration over a ***map*** occurs in order. For elements for which there is no obvious order or when there is no need to keep the container sorted, we might consider using a ***hash_map*** (§17.6).

17.4.1.1 Types [cont.map.types]

A ***map*** has the usual container member types (§16.3.1) plus a few relating to its specific function:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class std::map {
public:
    // types:

    typedef Key key_type;
    typedef T mapped_type;

    typedef pair<const Key, T> value_type;

    typedef Cmp key_compare;
    typedef A allocator_type;

    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
```

```
        typedef implementation_defined1 iterator;
        typedef implementation_defined2 const_iterator;

        typedef typename A::size_type size_type;
        typedef typename A::difference_type difference_type;

        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
        // ...
    };
```

Note that the ***value_type*** of a ***map*** is a (key,value) ***pair***. The type of the mapped values is referred to as the ***mapped_type***. Thus, a ***map*** is a sequence of ***pair<const Key, mapped_type>*** elements.

As usual, the actual iterator types are implementation-defined. Since a ***map*** most likely is implemented using some form of a tree, these iterators usually provide some form of tree traversal.

The reverse iterators are constructed from the standard ***reverse_iterator*** templates (§19.2.5).

17.4.1.2 Iterators and Pairs [cont.map.iter]

A ***map*** provides the usual set of functions that return iterators (§16.3.2):

```
    template <class Key, class T, class Cmp = less<Key>,
              class A = allocator< pair<const Key, T> > > class map {
    public:
        // ...
        // iterators:

        iterator begin();
        const_iterator begin() const;

        iterator end();
        const_iterator end() const;

        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;

        reverse_iterator rend();
        const_reverse_iterator rend() const;

        // ...
    };
```

Iteration over a ***map*** is simply an iteration over a sequence of ***pair<const Key, mapped_type>*** elements. For example, we might print out the entries of a phone book like this:

```
    void f(map<string, number>& phone_book)
    {
        typedef map<string, number>::const_iterator CI;
        for (CI p = phone_book.begin(); p!=phone_book.end(); ++p)
            cout << p->first << '\t' << p->second << '\n';
    }
```

A ***map*** iterator presents the elements in ascending order of its keys (§17.4.1.5). Therefore, the ***phone_book*** entries will be output in lexicographical order.

We refer to the first element of any ***pair*** as ***first*** and the second as ***second*** independently of what types they actually are:

```
template <class T1, class T2> struct std::pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair() :first(T1()), second(T2()) { }
    pair(const T1& x, const T2& y) :first(x), second(y) { }
    template<class U, class V>
        pair(const pair<U, V>& p) :first(p.first), second(p.second) { }
};
```

The last constructor exists to allow conversions in the initializer (§13.6.2). For example:

```
pair<int,double> f(char c, int i)
{
    return pair<int,double>(c,i);  // conversions required
}
```

In a ***map***, the key is the first element of the pair and the mapped value is the second.

The usefulness of ***pair*** is not limited to the implementation of ***map***, so it is a standard library class in its own right. The definition of ***pair*** is found in ***<utility>***. A function to make it convenient to create ***pair***s is also provided:

```
template <class T1, class T2> pair<T1,T2> std::make_pair(T1 t1, T2 t2)
{
    return pair<T1,T2>(t1,t2);
}
```

A ***pair*** is by default initialized to the default values of its element types. In particular, this implies that elements of built-in types are initialized to ***0*** (§5.1.1) and ***string***s are initialized to the empty string (§20.3.4). A type without a default constructor can be an element of a ***pair*** only provided the pair is explicitly initialized.

17.4.1.3 Subscripting [cont.map.element]

The characteristic ***map*** operation is the associative lookup provided by the subscript operator:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...

    mapped_type& operator[](const key_type& k);  // access element with key k

    // ...
};
```

The subscript operator performs a lookup on the key given as an index and returns the corresponding value. If the key isn't found, an element with the key and the default value of the ***mapped_type*** is inserted into the ***map***. For example:

```
void f()
{
    map<string, int> m;    // map starting out empty
    int x = m["Henry"]; // create new entry for "Henry", initialize to 0, return 0
    m["Harry"] = 7;     // create new entry for "Harry", initialize to 0, and assign 7
    int y = m["Henry"]; // return the value from "Henry"'s entry
    m["Harry"] = 9;     // change the value from "Harry"'s entry to 9
}
```

As a slightly more realistic example, consider a program that calculates sums of items presented as input in the form of (item-name,value) pairs such as

```
nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250
```

and also calculates the sum for each item. The main work can be done while reading the (item-name,value) pairs into a ***map***:

```
void readitems(map<string, int>& m)
{
    string word;
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}
```

The subscript operation ***m[word]*** identifies the appropriate ***(string, int)*** pair and returns a reference to its ***int*** part. This code takes advantage of the fact that a new element gets its ***int*** value set to ***0*** by default.

A ***map*** constructed by ***readitems()*** can then be output using a conventional loop:

```
int main()
{
    map<string, int> tbl;
    readitems(tbl);

    int total = 0;
    typedef map<string, int>::const_iterator CI;

    for (CI p = tbl.begin(); p!=tbl.end(); ++p) {
        total += p->second;
        cout << p->first << ´\t´ << p->second << ´\n´;
    }

    cout << "----------\ntotal\t" << total << ´\n´;

    return !cin;
}
```

Given the input above, the output is:

```
hammer    9
nail      1350
saw       7
----------------
total     1366
```

Note that the items are printed in lexical order (§17.4.1, §17.4.1.5).

A subscripting operation must find the key in the *map*. This, of course, is not as cheap as subscripting an array with an integer. The cost is $O(log(size_of_map))$, which is acceptable for many applications. For applications for which this is too expensive, a hashed container is often the answer (§17.6).

Subscripting a *map* adds a default element when the key is not found. Therefore, there is no version of *operator*[]() for *const map*s. Furthermore, subscripting can be used only if the *mapped_type* (value type) has a default value. If the programmer simply wants to see if a key is present, the *find*() operation (§17.4.1.6) can be used to locate a *key* without modifying the *map*.

17.4.1.4 Constructors [cont.map.ctor]

A *map* provides the usual complement of constructors, etc. (§16.3.4) :

```
template <class Key, class T, class Cmp =less<Key>,
          class A =allocator<pair<const Key,T> > >
class map {
public:
    // ...
    // construct/copy/destroy:

    explicit map(const Cmp& = Cmp(), const A& = A());
    template <class In> map(In first, In last, const Cmp& = Cmp(), const A& = A());
    map(const map&);

    ~map();

    map& operator=(const map&);

    // ...
};
```

Copying a container implies allocating space for its elements and making copies of each element (§16.3.4). This can be very expensive and should be done only when necessary. Consequently, containers such as *map*s tend to be passed by reference.

The member template constructor takes a sequence of *pair<const Key,T>*s described by a pair input iterator *In*. It *insert*()s (§17.4.1.7) the elements from the sequence into the *map*.

17.4.1.5 Comparisons [cont.map.comp]

To find an element in a *map* given a key, the *map* operations must compare keys. Also, iterators traverse a *map* in order of increasing key values, so insertion will typically also compare keys (to place an element into a tree structure representing the *map*).

By default, the comparison used for keys is < (less than), but an alternative can be provided as a

template parameter or as a constructor argument (see §17.3.3). The comparison given is a comparison of keys, but the *value_type* of a *map* is a (key,value) pair. Consequently, *value_comp*( ) is provided to compare such pairs using the key comparison function:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class map {
public:
    // ...

    typedef Cmp key_compare;

    class value_compare : public binary_function<value_type, value_type, bool> {
    friend class map;
    protected:
        Cmp cmp;
        value_compare(Cmp c) : cmp(c) {}
    public:
        bool operator()(const T& x, const T& y) const { return cmp(x.first, y.first); }
    };

    key_compare key_comp() const;
    value_compare value_comp() const;

    // ...
};
```

For example:

```
map<string, int> m1;
map<string, int, Nocase> m2;                    // specify comparison type (§17.1.4.1)
map<string, int, String_cmp> m3;                // specify comparison type (§17.1.4.1)
map<string, int> m4(String_cmp(literary));      // pass comparison object
```

The *key_comp*( ) and *value_comp*( ) member functions make it possible to query a *map* for the kind of comparisons used for keys and values. This is usually done to supply the same comparison criterion to some other container or algorithm. For example:

```
void f(map<string, int>& m)
{
    map<string, int> mm;                        // compare using < by default
    map<string, int> mmm(m.key_comp());         // compare the way m does
    // ...
}
```

See §17.1.4.1 for an example of how to define a particular comparison and §18.4 for an explanation of function objects in general.

17.4.1.6 Map Operations [cont.map.map]

The crucial idea for *map*s and indeed for all associative containers is to gain information based on a key. Several specialized operations are provided for that:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class map {
public:
    // ...
    // map operations:

    iterator find(const key_type& k);                    // find element with key k
    const_iterator find(const key_type& k) const;

    size_type count(const key_type& k) const;            // find number of elements with key k

    iterator lower_bound(const key_type& k);             // find first element with key k
    const_iterator lower_bound(const key_type& k) const;
    iterator upper_bound(const key_type& k);             // find first element with key greater than k
    const_iterator upper_bound(const key_type& k) const;

    pair<iterator, iterator> equal_range(const key_type& k);
    pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    // ...
};
```

A *m.find(k)* operation simply yields an iterator to an element with the key *k*. If there is no such element, the iterator returned is *m.end()*. For a container with unique keys, such as *map* and *set*, the resulting iterator will point to the unique element with the key *k*. For a container with non-unique keys, such as *multimap* and *multiset*, the resulting iterator will point to the first element that has that key. For example:

```
void f(map<string, int>& m)
{
    map<string, int>::iterator p = m.find("Gold");
    if (p!=m.end()) {                          // if "Gold" was found
        // ...
    }
    else if (m.find("Silver")!=m.end()) {      // look for "Silver"
        // ...
    }
    // ...
}
```

For a *multimap* (§17.4.2), finding the first match is rarely as useful as finding all matches; *m.lower_bound(k)* and *m.upper_bound(k)* give the beginning and the end of the subsequence of elements of *m* with the key *k*. As usual, the end of a sequence is an iterator to the one-past-the-last element of the sequence. For example:

```
void f(multimap<string, int>& m)
{
    multimap<string, int>::iterator lb = m.lower_bound("Gold");
    multimap<string, int>::iterator ub = m.upper_bound("Gold");
```

```
        for (multimap<string, int>::iterator p = lb; p!=ub; ++p) {
            // ...
        }
    }
```

Finding the upper bound and lower bound by two separate operations is neither elegant nor efficient. Consequently, the operation *equal_range*( ) is provided to deliver both. For example:

```
void f(multimap<string, int>& m)
{
    typedef multimap<string, int>::iterator MI;
    pair<MI, MI> g = m.equal_range("Gold");

    for (MI p = g.first; p!=g.second; ++p) {
        // ...
    }
}
```

If *lower_bound*(*k*) doesn't find *k*, it returns an iterator to the first element that has a key greater than *k*, or *end*( ) if no such greater element exists. This way of reporting failure is also used by *upper_bound*( ) and *equal_range*( ).

17.4.1.7 List Operations [cont.map.modifier]

The conventional way of entering a value into an associative array is simply to assign to it using subscripting. For example:

```
phone_book["Order department"] = 8226339;
```

This will make sure that the Order department has the desired entry in the *phone_book* independently of whether it had a prior entry. It is also possible to *insert*( ) entries directly and to remove entries using *erase*( ):

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class map {
public:
    // ...
    // list operations:

    pair<iterator, bool> insert(const value_type& val);    // insert (key,value) pair
    iterator insert(iterator pos, const value_type& val);  // pos is just a hint
    template <class In> void insert(In first, In last);    // insert elements from sequence

    void erase(iterator pos);                  // erase the element pointed to
    size_type erase(const key_type& k);        // erase element with key k (if present)
    void erase(iterator first, iterator last); // erase range
    void clear();

    // ...
};
```

The operation *m*.*insert*(*val*) attempts to add a (*Key*, *T*) pair *val* to *m*. Since *map*s rely on

unique keys, insertion takes place only if there is not already an element in the *m* with that key. The return value of *m.insert(val)* is a *pair<iterator, bool>*. The *bool* is *true* if *val* was actually inserted. The iterator refers to the element of *m* holding the key *k*. For example:

```
void f(map<string, int>& m)
{
    pair<string, int> p99("Paul", 99);

    pair<map<string, int>::iterator, bool> p = m.insert(p99);
    if (p.second) {
        // "Paul" was inserted
    }
    else {
        // "Paul" was there already
    }
    map<string, int>::iterator i = p.first;        // points to m["Paul"]
    // ...
}
```

Usually, we do not care whether a key is newly inserted or was present in the *map* before the *insert()*. When we are interested, it is often because we want to register the fact that a value is in a *map* somewhere else (outside the *map*). The other two versions of *insert()* do not return an indication of whether a value was actually inserted.

Specifying a position, *insert(pos, val)*, is simply a hint to the implementation to start the search for the key *val* at *pos*. If the hint is good, significant performance improvements can result. If the hint is bad, you'd have done better without it both notationally and efficiency-wise. For example:

```
void f(map<string, int>& m)
{
    m["Dilbert"] = 3;    // neat, possibly less efficient
    m.insert(m.begin(), make_pair(const string("Dogbert"), 99));        // ugly
}
```

In fact, [] is little more than a convenient notation for *insert()*. The result of *m[k]* is equivalent to the result of *(*(m.insert(make_pair(k, V())).first)).second*, where *V()* is the default value for the mapped type. When you understand that equivalence, you probably understand associative containers.

Because [] always uses *V()*, you cannot use subscripting on a *map* with a value type that does not have a default value. This is an unfortunate limitation of the standard associative containers. However, the requirement of a default value is not a fundamental property of associative containers (see §17.6.2).

You can erase elements specified by a key. For example:

```
void f(map<string, int>& m)
{
    int count = phone_book.erase("Ratbert");
    // ...
}
```

The integer returned is the number of erased elements. In particular, *count* is *0* if there was no element with the key "*Ratbert*" to erase. For a *multimap* or *multiset*, the value can be larger than *1*. Alternatively, one can erase an element given an iterator pointing to it or a range of elements given a sequence. For example:

```
void g(map<string,int>& m)
{
    m.erase(m.find("Catbert"));
    m.erase(m.find("Alice"),m.find("Wally"));
}
```

Naturally, it is faster to erase an element for which you already have an iterator than to first find the element given its key and then erase it. After *erase*( ), the iterator cannot be used again because the element to which it pointed is no longer there. Erasing *end*( ) is harmless.

17.4.1.8 Other Functions [cont.map.etc]

Finally, a *map* provides the usual functions dealing with the number of elements and a specialized *swap*( ):

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...
    // capacity:

    size_type size() const;          // number of elements
    size_type max_size() const;      // size of largest possible map
    bool empty() const { return size()==0; }

    void swap(map&);
};
```

As usual, a value returned by *size*( ) or *max_size*( ) is a number of elements.

In addition, *map* provides ==, !=, <, >, <=, >=, and *swap*( ) as nonmember functions:

```
template <class Key, class T, class Cmp, class A>
bool operator==(const map<Key,T,Cmp,A>&, const map<Key,T,Cmp,A>&);

// similarly !=, <, >, <=, and >=

template <class Key, class T, class Cmp, class A>
void swap(map<Key,T,Cmp,A>&, map<Key,T,Cmp,A>&);
```

Why would anyone want to compare two *map*s? When we specifically compare two *map*s, we usually want to know not just if the *map*s differ, but also how they differ if they do. In such cases, we don't use == or !=. However, by providing ==, <, and *swap*( ) for every container, we make it possible to write algorithms that can be applied to every container. For example, these functions allow us to *sort*( ) a *vector* of *map*s and to have a *set* of *map*s.

17.4.2 Multimap [cont.multimap]

A ***multimap*** is like a ***map***, except that it allows duplicate keys:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class std::multimap {
public:
    // like map, except:

    iterator insert(const value_type&);    // returns iterator, not pair

    // no subscript operator []
};
```

For example (using ***Cstring_less*** from §17.1.4.1 to compare C-style strings):

```
void f(map<char*, int, Cstring_less>& m, multimap<char*, int, Cstring_less>& mm)
{
    m.insert(make_pair("x",4));
    m.insert(make_pair("x",5));  // no effect: there already is an entry for "x" (§17.4.1.7)
    // now m["x"] == 4

    mm.insert(make_pair("x",4));
    mm.insert(make_pair("x",5));
    // mm now holds both ("x",4) and ("x",5)
}
```

This implies that ***multimap*** cannot support subscripting by key values in the way ***map*** does. The ***equal_range***( ), ***lower_bound***( ), and ***upper_bound***( ) operations (§17.4.1.6) are the primary means of accessing multiple values with the same key.

Naturally, where several values can exist for a single key, a ***multimap*** is preferred over a ***map***. That happens far more often than people first think when they hear about ***multimap***. In some ways, a ***multimap*** is even cleaner and more elegant than a ***map***.

Because a person can easily have several phone numbers, a phone book is a good example of a ***multimap***. I might print my phone numbers like this:

```
void print_numbers(const multimap<string, int>& phone_book)
{
    typedef multimap<string, int>::const_iterator I;
    pair<I, I> b = phone_book.equal_range("Stroustrup");
    for (I i = b.first; i != b.second; ++i) cout << i->second << '\n';
}
```

For a ***multimap***, the argument to ***insert***( ) is always inserted. Consequently, the ***multimap***::***insert***( ) returns an iterator rather than a ***pair<iterator, bool>*** like ***map*** does. For uniformity, the library could have provided the general form of ***insert***( ) for both ***map*** and ***multimap*** even though the ***bool*** would have been redundant for a ***multimap***. Yet another design alternative would have been to provide a simple ***insert***( ) that didn't return a ***bool*** in either case and then supply users of ***map*** with some other way of figuring out whether a key was newly inserted. This is a case in which different interface design ideas clash.

17.4.3 Set [cont.set]

A *set* can be seen as a *map* (§17.4.1), where the values are irrelevant, so we keep track of only the keys. This leads to only minor changes to the user interface:

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key> >
class std::set {
public:
    // like map except:

    typedef Key value_type;          // the key itself is the value
    typedef Cmp value_compare;
    // no subscript operator []
};
```

Defining *value_type* as the *key_type* type is a trick to allow code that uses *map*s and *set*s to be identical in many cases.

Note that *set* relies on a comparison operation (by default <) rather than equality (==). This implies that equivalence of elements is defined by inequality (§17.1.4.1) and that iteration through a *set* has a well-defined order.

Like *map*, *set* provides ==, !=, <, >, <=, >=, and *swap*( ).

17.4.4 Multiset [cont.multiset]

A *multiset* is a *set* that allows duplicate keys:

```
template <class Key, class T, class Cmp = less<Key>, class A = allocator<Key> >
class std::multiset {
public:
    // like set, except:
    iterator insert(const value_type&);    // returns iterator, not pair
};
```

The *equal_range*( ), *lower_bound*( ), and *upper_bound*( ) operations (§17.4.1.6) are the primary means of accessing multiple occurrences of a key.

17.5 Almost Containers [cont.etc]

Built-in arrays (§5.2), *string*s (Chapter 20), *valarray*s (§22.4), and *bitset*s (§17.5.3) hold elements and can therefore be considered containers for many purposes. However, each lacks some aspect or other of the standard container interface, so these ‘‘almost containers’’ are not completely interchangeable with fully developed containers such as *vector* and *list*.

17.5.1 String [cont.string]

A *basic_string* provides subscripting, random-access iterators, and most of the notational conveniences of a container (Chapter 20). However, *basic_string* does not provide as wide a selection of types as elements. It also is optimized for use as a string of characters and is typically used in ways that differ significantly from a container.

17.5.2 Valarray [cont.valarray]

A ***valarray*** (§22.4) is a vector for optimized numeric computation. Consequently, a ***valarray*** doesn't attempt to be a general container. A ***valarray*** provides many useful numeric operations. However, of the standard container operations (§17.1.1), it offers only ***size*()** and a subscript operator (§22.4.2). A pointer to an element of a ***valarray*** is a random-access iterator (§19.2.1).

17.5.3 Bitset [cont.bitset]

Often, aspects of a system, such as the state of an input stream (§21.3.3), are represented as a set of flags indicating binary conditions such as good/bad, true/false, and on/off. C++ supports the notion of small sets of flags efficiently through bitwise operations on integers (§6.2.4). These operations include & (and), | (or), ^ (exclusive or), << (shift left), and >> (shift right). Class ***bitset<N>*** generalizes this notion and offers greater convenience by providing operations on a set of ***N*** bits indexed from ***0*** through ***N-1***, where ***N*** is known at compile time. For sets of bits that don't fit into a ***long int***, using a ***bitset*** is much more convenient than using integers directly. For smaller sets, there may be an efficiency tradeoff. If you want to name the bits, rather than numbering them, using a ***set*** (§17.4.3), an enumeration (§4.8), or a bitfield (§C.8.1) are alternatives.

A ***bitset<N>*** is an array of ***N*** bits. A ***bitset*** differs from a ***vector<bool>*** (§16.3.11) by being of fixed size, from ***set*** (§17.4.3) by having its bits indexed by integers rather than associatively by value, and from both ***vector<bool>*** and ***set*** by providing operations to manipulate the bits.

It is not possible to address a single bit directly using a built-in pointer (§5.1). Consequently, ***bitset*** provides a reference-to-bit type. This is actually a generally useful technique for addressing objects for which a built-in pointer for some reason is unsuitable:

```
template<size_t N> class std::bitset {
public:
    class reference {              // reference to a single bit:
        friend class bitset;
        reference();
    public:                                   // b[i] refers to the (i+1)'th bit:
        ~reference();
        reference& operator=(bool x);             // for b[i] = x;
        reference& operator=(const reference&);   // for b[i] = b[j];
        bool operator~() const;                   // return ~b[i]
        operator bool() const;                    // for x = b[i];
        reference& flip();                        // b[i].flip();
    };
    // ...
};
```

The ***bitset*** template is defined in namespace ***std*** and presented in ***<bitset>***.

For historical reasons, ***bitset*** differs somewhat in style from other standard library classes. For example, if an index (also known as a *bit position*) is out of range, an ***out_of_range*** exception is thrown. No iterators are provided. Bit positions are numbered right to left in the same way bits often are in a word, so the value of ***b[i]*** is ***pow(i,2)***. Thus, a bitset can be thought of as an ***N***-bit binary number:

position:	9	8	7	6	5	4	3	2	1	0
bitset<10>:	1	1	1	1	0	1	1	1	0	1

17.5.3.1 Constructors [cont.bitset.ctor]

A ***bitset*** can be constructed with default values, from the bits in an ***unsigned long int***, or from a ***string***:

```
template<size_t N> class bitset {
public:
    // ...
    // constructors:

    bitset();                          // N zero-bits
    bitset(unsigned long val);         // bits from val

    template<class Ch, class Tr, class A>                  // Tr is a character trait (§20.2)
    explicit bitset(const basic_string<Ch,Tr,A>& str,      // bits from string str
        basic_string<Ch,Tr,A>::size_type pos = 0,
        basic_string<Ch,Tr,A>::size_type n = basic_string<Ch,Tr,A>::npos);

    // ...
};
```

The default value of a bit is ***0***. When an ***unsigned long int*** argument is supplied, each bit in the integer is used to initialize the corresponding bit in the bitset (if any). A ***basic_string*** (Chapter 20) argument does the same, except that the character ´***0***´ gives the bitvalue ***0***, the character ´***1***´ gives the bitvalue ***1***, and other characters cause an ***invalid_argument*** exception to be thrown. By default, a complete string is used for initialization. However, in the style of a ***basic_string*** constructor (§20.3.4), a user can specify that only the range of characters from ***pos*** to the end of the string or to ***pos+n*** are to be used. For example:

```
void f()
{
    bitset<10> b1;   // all 0

    bitset<16> b2 = 0xaaaa;                  // 1010101010101010
    bitset<32> b3 = 0xaaaa;                  // 00000000000000001010101010101010

    bitset<10> b4("1010101010");             // 1010101010
    bitset<10> b5("10110111011110",4);       // 0111011110
    bitset<10> b6("10110111011110",2,8);     // 0011011101

    bitset<10> b7("n0g00d");                 // invalid_argument thrown
    bitset<10> b8 = "n0g00d";                // error: no char* to bitset conversion
}
```

A key idea in the design of ***bitset*** is that an optimized implementation can be provided for bitsets that fit in a single word. The interface reflects this assumption.

17.5.3.2 Bit Manipulation Operations [cont.bitset.oper]

A ***bitset*** provides the operators for accessing individual bits and for manipulating all bits in the set:

```
template<size_t N> class std::bitset {
public:
    // ...
    // bitset operations:

    reference operator[](size_t pos);          // b[i]

    bitset& operator&=(const bitset& s);       // and
    bitset& operator|=(const bitset& s);       // or
    bitset& operator^=(const bitset& s);       // exclusive or

    bitset& operator<<=(size_t n);             // logical left shift (fill with zeros)
    bitset& operator>>=(size_t n);             // logical right shift (fill with zeros)

    bitset& set();                             // set every bit to 1
    bitset& set(size_t pos, int val = 1);      // b[pos]=val

    bitset& reset();                           // set every bit to 0
    bitset& reset(size_t pos);                 // b[pos]=0

    bitset& flip();                            // change the value of every bit
    bitset& flip(size_t pos);                  // change the value of b[pos]

    bitset operator~() const { return bitset<N>(*this).flip(); }     // make complement set
    bitset operator<<(size_t n) const { return bitset<N>(*this)<<=n; }  // make shifted set
    bitset operator>>(size_t n) const { return bitset<N>(*this)>>=n; }  // make shifted set

    // ...
};
```

The subscript operator throws ***out_of_range*** if the subscript is out of range. There is no unchecked subscript operation.

The ***bitset***& returned by these operations is ****this***. An operator returning a ***bitset*** (rather than a ***bitset***&) makes a copy of ****this***, applies its operation to that copy, and returns the result. In particular, >> and << really are shift operations rather than I/O operations. The output operator for a ***bitset*** is a << that takes an ***ostream*** and a ***bitset*** (§17.5.3.3).

When bits are shifted, a logical (rather than cyclic) shift is used. That implies that some bits ‘‘fall off the end’’ and that some positions get the default value 0. Note that because ***size_t*** is an unsigned type, it is not possible to shift by a negative number. It does, however, imply that ***b<<-1*** shifts by a very large positive value, thus leaving every bit of the ***bitset b*** with the value ***0***. Your compiler should warn against this.

17.5.3.3 Other Operations [cont.bitset.etc]

A ***bitset*** also supports common operations such as ***size***( ), ==, I/O , etc.:

```
template<size_t N> class bitset {
public:
    // ...

    unsigned long to_ulong() const;

    template <class Ch, class Tr, class A> basic_string<Ch,Tr,A> to_string() const;

    size_t count() const;                  // number of bits with value 1
    size_t size() const { return N; }      // number of bits

    bool operator==(const bitset& s) const;
    bool operator!=(const bitset& s) const;

    bool test(size_t pos) const;           // true if b[pos] is 1
    bool any() const;                      // true if any bit is 1
    bool none() const;                     // true if no bit is 1
};
```

The operations *to_ulong*( ) and *to_string*( ) provide the inverse operations to the constructors. To avoid nonobvious conversions, named operations were preferred over conversion operations. If the value of the *bitset* has so many significant bits that it cannot be represented as an *unsigned long*, *to_ulong*( ) throws *overflow_error*.

The *to_string*( ) operation produces a string of the desired type holding a sequence of *´0´* and *´1´* characters; *basic_string* is the template used to implement strings (Chapter 20). We could use *to_string* to write out the binary representation of an *int*:

```
void binary(int i)
{
    bitset<8*sizeof(int)> b = i;        // assume 8-bit byte (see also §22.2)
    cout << b.template to_string<char>() << ´\n´;
}
```

Unfortunately, invoking an explicitly qualified member template requires a rather elaborate and rare syntax (§C.13.6).

In addition to the member functions, *bitset* provides binary & (and), | (or), ^ (exclusive or), and the usual I/O operators:

```
template<size_t N> bitset<N>& std::operator&(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N>& std::operator|(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N>& std::operator^(const bitset<N>&, const bitset<N>&);

template <class charT, class Tr, size_t N>
basic_istream<charT,Tr>& std::operator>>(basic_istream<charT,Tr>&, bitset<N>&);
template <class charT, class Tr, size_t N>
basic_ostream<charT,Tr>& std::operator<<(basic_ostream<charT,Tr>&, const bitset<N>&);
```

We can therefore write out a bitset without first converting it to a string. For example:

```
void binary(int i)
{
    bitset<8*sizeof(int)> b = i;        // assume 8-bit byte (see also §22.2)
    cout << b << '\n';
}
```

This prints the bits represented as *1*s and *0*s left-to-right, with the most significant bit leftmost.

17.5.4 Built-In Arrays [cont.array]

A built-in array supplies subscripting and random-access iterators in the form of ordinary pointers (§2.7.2). However, an array doesn't know its own size, so users must keep track of that size. In general, an array doesn't provide the standard member operations and types.

It is possible, and sometimes useful, to provide an ordinary array in a guise that provides the notational convenience of a standard container without changing its low-level nature:

```
template<class T, int max> struct c_array {
    typedef T value_type;

    typedef T* iterator;
    typedef const T* const_iterator;

    typedef T& reference;
    typedef const T& const_reference;

    T v[max];
    operator T*() { return v; }

    reference operator[](size_t i) { return v[i]; }
    const_reference operator[](size_t i) const { return v[i]; }

    iterator begin() { return v; }
    const_iterator begin() const { return v; }

    iterator end() { return v+max; }
    const_iterator end() const { return v+max; }

    ptrdiff_t size() const { return max; }
};
```

The *c_array* template is not part of the standard library. It is presented here as a simple example of how to fit a ''foreign'' container into the standard container framework. It can be used with standard algorithms (Chapter 18) using *begin*(), *end*(), etc. It can be allocated on the stack without any indirect use of dynamic memory. Also, it can be passed to a C-style function that expects a pointer. For example:

```
void f(int* p, int sz);     // C-style

void g()
{
    c_array<int,10> a;
```

```
        f(a,a.size());                                          // C-style use
        c_array<int,10>::iterator p = find(a.begin(),a.end(),777);  // C++/STL style use
        // ...
    }
```

17.6 Defining a New Container [cont.hash]

The standard containers provide a framework to which a user can add. Here, I show how to provide a container in such a way that it can be used interchangeably with the standard containers wherever reasonable. The implementation is meant to be realistic, but it is not optimal. The interface is chosen to be very close to that of existing, widely-available, and high-quality implementations of the notion of a ***hash_map***. Use the ***hash_map*** provided here to study the general issues. Then, use a supported ***hash_map*** for production use.

17.6.1 Hash_map [cont.hash.map]

A ***map*** is an associative container that accepts almost any type as its element type. It does that by relying only on a less-than operation for comparing elements (§17.4.1.5). However, if we know more about a key type we can often reduce the time needed to find an element by providing a hash function and implementing a container as a hash table.

A hash function is a function that quickly maps a value to an index in such a way that two distinct values rarely end up with the same index. Basically, a hash table is implemented by placing a value at its index, unless another value is already placed there, and ''nearby'' if one is. Finding an element placed at its index is fast, and finding one ''nearby'' is not slow, provided equality testing is reasonably fast. Consequently, it is not uncommon for a ***hash_map*** to provide five to ten times faster lookup than a ***map*** for larger containers, where the speed of lookup matters most. On the other hand, a ***hash_map*** with an ill-chosen hash function can be much slower than a ***map***.

There are many ways of implementing a hash table. The interface of ***hash_map*** is designed to differ from that of the standard associative containers only where necessary to gain performance through hashing. The most fundamental difference between a ***map*** and a ***hash_map*** is that a ***map*** requires a **<** for its element type, while a ***hash_map*** requires an **==** and a hash function. Thus, a ***hash_map*** must differ from a ***map*** in the non-default ways of creating one. For example:

```
map<string,int> m1;                 // compare strings using <
map<string,int,Nocase> m2;          // compare strings using Nocase() (§17.1.4.1)

hash_map<string,int> hm1;           // hash using Hash<string>() (§17.6.2.3), compare using ==
hash_map<string,int,hfct> hm2;      // hash using hfct(), compare using ==
hash_map<string,int,hfct,eql> hm3;  // hash using hfct(), compare using eql
```

A container using hashed lookup is implemented using one or more tables. In addition to holding its elements, the container needs to keep track of which values have been associated with each hashed value (''index'' in the prior explanation); this is done using a ''hash table.'' Most hash table implementations seriously degrade in performance if that table gets ''too full,'' say 75% full. Consequently, the ***hash_map*** defined next is automatically resized when it gets too full. However, resizing can be expensive, so it is useful to be able to specify an initial size.

Thus, a first approximation of a ***hash_map*** looks like this:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // like map, except:

    typedef H Hasher;
    typedef EQ key_equal;

    hash_map(const T& dv =T(), size_type n =101, const H& hf =H(), const EQ& =EQ());
    template<class In> hash_map(In first, In last,
        const T& dv =T(), size_type n =101, const H& hf =H(), const EQ& =EQ());
};
```

Basically, this is the ***map*** interface (§17.4.1.4), with < replaced by == and a hash function.

The uses of a ***map*** in this book so far (§3.7.4, §6.1, §17.4.1) can be converted to use a ***hash_map*** simply by changing the name ***map*** to ***hash_map***. Often, a change between a ***map*** and a ***hash_map*** can be eased by using ***typedef***. For example:

```
typedef hash_map<string,record> Map;
Map dictionary;
```

The ***typedef*** is also useful to further hide the actual type of the dictionary from its users.

Though not strictly correct, I think of the tradeoff between a ***map*** and a ***hash_map*** as simply a space/time tradeoff. If efficiency isn't an issue, it isn't worth wasting time choosing between them: either will do well. For large and heavily used tables, ***hash_map*** has a definite speed advantage and should be used unless space is a premium. Even then, I might consider other ways of saving space before choosing a ''plain'' ***map***. Actual measurement is essential to avoid optimizing the wrong code.

The key to efficient hashing is the quality of the hash function. If a good hash function isn't available, a ***map*** can easily outperform a ***hash_map***. Hashing based on a C-style string, a ***string***, or an integer is usually very effective. However, it is worth remembering that the effectiveness of a hash function critically depends on the actual values being hashed (§17.8[35]). A ***hash_map*** must be used where < is not defined or is unsuitable for the intended key. Conversely, a hash function does not define an ordering the way < does, so a ***map*** must be used when it is important to keep the elements sorted.

Like ***map***, ***hash_map*** provides ***find***( ) to allow a programmer to determine whether a key has been inserted.

17.6.2 Representation and Construction [cont.hash.rep]

Many different implementations of a ***hash_map*** are possible. Here, I use one that is reasonably fast and whose most important operations are fairly simple. The key operations are the constructors, the lookup (operator [ ]), the resize operation, and the operation removing an element (***erase***( )).

The simple implementation chosen here relies on a hash table that is a ***vector*** of pointers to entries. Each ***Entry*** holds a ***key***, a ***value***, a pointer to the next ***Entry*** (if any) with the same hash value, and an ***erased*** bit :

Expressed as declarations, it looks like this:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...
private:                // representation
    struct Entry {
        key_type key;
        mapped_type val;
        Entry* next;    // hash overflow link
        bool erased;
        Entry(key_type k, mapped_type v, Entry* n)
            : key(k), val(v), next(n), erased(false) { }
    };

    vector<Entry> v;    // the actual entries
    vector<Entry*> b;   // the hash table: pointers into v

    // ...
};
```

Note the ***erased*** bit. The way several values with the same hash value are handled here makes it hard to remove an element. So instead of actually removing an element when ***erase***( ) is called, I simply mark the element ***erased*** and ignore it until the table is resized.

In addition to the main data structure, a ***hash_map*** needs a few pieces of administrative data. Naturally, each constructor needs to set up all of this. For example:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...

    hash_map(const T& dv =T(), size_type n =101, const H& h =H(), const EQ& e =EQ())
        : default_value(dv), b(n), no_of_erased(0), hash(h), eq(e)
    {
        set_load();                     // defaults
        v.reserve(max_load*b.size());   // reserve space for growth
    }

    void set_load(float m = 0.7, float g = 1.6) { max_load = m; grow = g; }

    // ...
```

```
private:
    float max_load;             // keep v.size()<=b.size()*max_load
    float grow;                 // when necessary, resize(bucket_count()*grow)

    size_type no_of_erased;     // number of entries in v occupied by erased elements

    Hasher hash;                // hash function
    key_equal eq;               // equality

    const T default_value;      // default value used by []
};
```

The standard associative containers require that a mapped type have a default value (§17.4.1.7). This restriction is not logically necessary and can be inconvenient. Making the default value an argument allows us to write:

```
hash_map<string,Number> phone_book1;                // default: Number()
hash_map<string,Number> phone_book2(Number(411));   // default: Number(411)
```

17.6.2.1 Lookup [cont.hash.lookup]

Finally, we can provide the crucial lookup operations:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...
    mapped_type& operator[](const key_type& k);

    iterator find(const key_type&);
    const_iterator find(const key_type&) const;
    // ...
};
```

To find a ***value***, ***operator[]()*** uses a hash function to find an index in the hash table for the ***key***. It then searches through the entries until it finds a matching ***key***. The ***value*** in that ***Entry*** is the one we are seeking. If it is not found, a default value is entered:

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<T> >
mapped_type& hash_map::operator[](const key_type& k)
{
    size_type i = hash(k)%b.size();                 // hash

    for(Entry* p = b[i]; p; p = p->next) // search among entries hashed to i
        if (eq(k,p->key)) {             // found
            if (p->erased) {            // re-insert
                p->erased = false;
                no_of_erased--;
                return p->val = default_value;
            }
            return p->val;
        }
```

```
        // not found:

        if (b.size()*max_load < v.size()) { // if ''too full''
            resize(b.size()*grow);          // grow
            return operator[](k);           // rehash
        }

        v.push_back(Entry(k,default_value,b[i]));    // add Entry
        b[i] = &v.back();                            // point to new element

        return b[i]->val;
    }
```

Unlike ***map***, ***hash_map*** doesn't rely on an equality test synthesized from a less-than operation (§17.1.4.1). This is because of the call of ***eq***( ) in the loop that looks through elements with the same hash value. This loop is crucial to the performance of the lookup, and for common and obvious key types such as ***string*** and C-style strings, the overhead of an extra comparison could be significant.

I could have used a ***set<Entry>*** to represent the set of values that have the same hash value. However, if we have a good hash function (***hash***( )) and an appropriately-sized hash table (***b***), most such sets will have exactly one element. Consequently, I linked the elements of that set together using the ***next*** field of ***Entry*** (§17.8[27]).

Note that ***b*** keeps pointers to elements of ***v*** and that elements are added to ***v***. In general, ***push_back***( ) can cause reallocation and thus invalidate pointers to elements (§16.3.5). However, in this case constructors (§17.6.2) and ***resize***( ) carefully ***reserve***( ) enough space so that no unexpected reallocation happens.

17.6.2.2 Erase and Rehash [cont.hash.erase]

Hashed lookup becomes inefficient when the table gets too full. To lower the chance of that happening, the table is automatically ***resize***( )d by the subscript operator. The ***set_load***( ) (§17.6.2) provides a way of controlling when and how resizing happens. Other functions are provided to allow a programmer to observe the state of a ***hash_map***:

```
template<class Key, class T, class H = Hash<Key>,
            class EQ = equal_to<Key>, class A = allocator<T> >
class hash_map {
    // ...

    void resize(size_type n);                   // make the size of the hash table n

    void erase(iterator position);              // erase the element pointed to

    size_type size() const { return v.size()-no_of_erased; }  // number of elements

    size_type bucket_count() const { return b.size(); }       // size of hash table

    Hasher hash_fun() const { return hash; }                  // hash function used
    key_equal key_eq() const { return eq; }                   // equality used
```

```
        // ...
    };
```

The *resize*( ) operation is essential, reasonably simple, and potentially expensive:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
void hash_map::resize(size_type s)
{
    if (s <= b.size()) return;
    b.resize(s);    // add s-b.size() pointers
    b.clear();
    v.reserve(s*max_load);    // if v needs to reallocate, let it happen now

    if (no_of_erased) {    // really remove erased elements
        for (size_type i = v.size()-1; 0<=i; i--)
            if (v[i].erased) {
                v.erase(&v[i]);
                if (--no_of_erased == 0) break;
            }
    }

    for (size_type i = 0; i<v.size(); i++) {            // rehash:
        size_type ii = hash(v[i].key)%b.size();         // hash
        v[i].next = b[ii];                              // link
        b[ii] = &v[i];
    }
}
```

If necessary, a user can ‘‘manually’’ call *resize*( ) to ensure that the cost is incurred at a predictable time. I have found a *resize*( ) operation important in some applications, but it is not fundamental to the notion of hash tables. Some implementation strategies don’t need it.

All of the real work is done elsewhere (and only if a *hash_map* is resized) , so *erase*( ) is trivial:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<T> >
void hash_map::erase(iterator p)    // erase the element pointed to
{
    if (p->erased == false) no_of_erased++;
    p->erased = true;
}
```

17.6.2.3 Hashing [cont.hasher]

To complete *hash_map*::*operator*[ ]( ), we need to define *hash*( ) and *eq*( ). For reasons that will become clear in §18.4, a hash function is best defined as *operator*( )( ) for a function object:

```
template <class T> struct Hash : unary_function<T, size_t> {
    size_t operator()(const T& key) const;
};
```

A good hash function takes a key and returns an integer so that different keys yield different integers with high probability. Choosing a good hash function is an art. However, exclusive-or'ing the bits of the key's representation into an integer is often acceptable:

```
template <class T> size_t Hash<T>::operator()(const T& key) const
{
    size_t res = 0;

    size_t len = sizeof(T);
    const char* p = reinterpret_cast<const char*>(&key);

    while (len--) res = (res<<1)^*p++; // use bytes of key's representation
    return res;
}
```

The use of ***reinterpret_cast*** (§6.2.7) is a good indication that something unsavory is going on and that we can do better in cases when we know more about the object being hashed. In particular, if an object contains a pointer, if the object is large, or if the alignment requirements on members have left unused space (''holes'') in the representation, we can usually do better (see §17.8[29]).

A C-style string is a pointer (to the characters), and a ***string*** contains a pointer. Consequently, specializations are in order:

```
size_t Hash<char*>::operator()(const char* key) const
{
    size_t res = 0;

    while (*key) res = (res<<1)^*key++;      // use int value of characters
    return res;
}

template <class C>
size_t Hash< basic_string<C> >::operator()(const basic_string<C>& key) const
{
    size_t res = 0;

    typedef basic_string<C>::const_iterator CI;
    CI p = key.begin();
    CI end = key.end();

    while (p!=end) res = (res<<1)^*p++;      // use int value of characters
    return res;
}
```

An implementation of ***hash_map*** will include hash functions for at least integer and string keys. For more adventurous key types, the user may have to help out with suitable specializations. Experimentation supported by good measurement is essential when choosing a hash function. Intuition tends to work poorly in this area.

To complete the ***hash_map***, we need to define the iterators and a minor host of trivial functions; this is left as an exercise (§17.8[34]).

17.6.3 Other Hashed Associative Containers [cont.hash.other]

For consistency and completeness, the ***hash_map*** should have matching ***hash_set***, ***hash_multimap***, and ***hash_multiset***. Their definitions are obvious from those of ***hash_map***, ***map***, ***multimap***, ***set***, and ***multiset***, so I leave these as an exercise (§17.8[34]). Good public domain and commercial implementations of these hashed associative containers are available. For real programs, these should be preferred to locally concocted versions, such as mine.

17.7 Advice [cont.advice]

[1] By default, use ***vector*** when you need a container; §17.1.
[2] Know the cost (complexity, big-O measure) of every operation you use frequently; §17.1.2.
[3] The interface, implementation, and representation of a container are distinct concepts. Don't confuse them; §17.1.3.
[4] You can sort and search according to a variety of criteria; §17.1.4.1.
[5] Do not use a C-style string as a key unless you supply a suitable comparison criterion; §17.1.4.1.
[6] You can define a comparison criteria so that equivalent, yet different, key values map to the same key; §17.1.4.1.
[7] Prefer operations on the end of a sequence (***back***-operations) when inserting and deleting elements; §17.1.4.1.
[8] Use ***list*** when you need to do many insertions and deletions from the front or the middle of a container; §17.2.2.
[9] Use ***map*** or ***multimap*** when you primarily access elements by key; §17.4.1.
[10] Use the minimal set of operations to gain maximum flexibility; §17.1.1
[11] Prefer a ***map*** to a ***hash_map*** if the elements need to be kept in order; §17.6.1.
[12] Prefer a ***hash_map*** to a ***map*** when speed of lookup is essential; §17.6.1.
[13] Prefer a ***hash_map*** to a ***map*** if no less-than operation can be defined for the elements; §17.6.1.
[14] Use ***find*( )** when you need to check if a key is in an associative container; §17.4.1.6.
[15] ***Use equal_range*( )** to find all elements of a given key in an associative container; §17.4.1.6.
[16] Use ***multimap*** when several values need to be kept for a single key; §17.4.2.
[17] Use ***set*** or ***multiset*** when the key itself is the only value you need to keep; §17.4.3.

17.8 Exercises [cont.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems. Then, look at your implementation's version of the containers and their operations.

1. (∗2.5) Understand the ***O*( )** notation (§17.1.2). Do some measurements of operations on standard containers to determine the constant factors involved.
2. (∗2) Many phone numbers don't fit into a ***long***. Write a ***phone_number*** type and a class that provides a set of useful operations on a container of ***phone_numbers***.

3. (∗2) Write a program that lists the distinct words in a file in alphabetical order. Make two versions: one in which a word is simply a whitespace-separated sequence of characters and one in which a word is a sequence of letters separated by any sequence of non-letters.
4. (∗2.5) Implement a simple solitaire card game.
5. (∗1.5) Implement a simple test of whether a word is a palindrome (that is, if its representation is symmetric; examples are ***ada***, ***otto***, and ***tut***). Implement a simple test of whether an integer is a palindrome. Implement a simple test of a whether sentence is a palindrome. Generalize.
6. (∗1.5) Define a queue using (only) two ***stack***s.
7. (∗1.5) Define a stack similar to ***stack*** (§17.3.1), except that it doesn't copy its underlying container and that it allows iteration over its elements.
8. (∗3) Your computer will have support for concurrent activities through the concept of a thread, task, or process. Figure out how that is done. The concurrency mechanism will have a concept of locking to prevent two tasks accessing the same memory simultaneously. Use the machine's locking mechanism to implement a lock class.
9. (∗2.5) Read a sequence of dates such as ***Dec85***, ***Dec50***, ***Jan76***, etc., from input and then output them so that later dates come first. The format of a date is a three-letter month followed by a two-digit year. Assume that all the years are from the same century.
10. (∗2.5) Generalize the input format for dates to allow dates such as ***Dec1985***, ***12/3/1990***, ***(Dec,30,1950)***, ***3/6/2001***, etc. Modify exercise §17.8[9] to cope with the new formats.
11. (∗1.5) Use a ***bitset*** to print the binary values of some numbers, including ***0***, ***1***, ***-1***, ***18***, ***-18***, and the largest positive ***int***.
12. (∗1.5) Use ***bitset*** to represent which students in a class were present on a given day. Read the ***bitset***s for a series of 12 days and determine who was present every day. Determine which students were present at least 8 days.
13. (∗1.5) Write a ***List*** of pointers that ***delete***s the objects pointed to when it itself is destroyed or if the element is removed from the ***List***.
14. (∗1.5) Given a ***stack*** object, print its elements in order (without changing the value of the stack).
15. (∗2.5) Complete ***hash_map*** (§17.6.1). This involves implementing ***find()*** and ***equal_range()*** and devising a way of testing the completed template. Test ***hash_map*** with at least one key type for which the default hash function would be unsuitable.
16. (∗2.5) Implement and test a list in the style of the standard ***list***.
17. (∗2) Sometimes, the space overhead of a ***list*** can be a problem. Write and test a singly-linked list in the style of a standard container.
18. (∗2.5) Implement a list that is like a standard ***list***, except that it supports subscripting. Compare the cost of subscripting for a variety of lists to the cost of subscripting a ***vector*** of the same length.
19. (∗2) Implement a template function that merges two containers.
20. (∗1.5) Given a C-style string, determine whether it is a palindrome. Determine whether an initial sequence of at least three words in the string is a palindrome.
21. (∗2) Read a sequence of ***(name,value)*** pairs and produce a sorted list of ***(name,total,mean,median)*** 4-tuples. Print that list.
22. (∗2.5) Determine the space overhead of each of the standard containers on your implementation.
23. (∗3.5) Consider what would be a reasonable implementation strategy for a ***hash_map*** that needed to use minimal space. Consider what would be a reasonable implementation strategy for

a ***hash_map*** that needed to use minimal lookup time. In each case, consider what operations you might omit so as to get closer to the ideal (no space overhead and no lookup overhead, respectively). Hint: There is an enormous literature on hash tables.

24. (∗2) Devise a strategy for dealing with overflow in ***hash_map*** (different values hashing to the same hash value) that makes ***equal_range*( )** trivial to implement.
25. (∗2.5) Estimate the space overhead of a ***hash_map*** and then measure it. Compare the estimate to the measurements. Compare the space overhead of your ***hash_map*** and your implementation's ***map***.
26. (∗2.5) Profile your ***hash_map*** to see where the time is spent. Do the same for your implementation's ***map*** and a widely-distributed ***hash_map***.
27. (∗2.5) Implement a ***hash_map*** based on a ***vector<map<K, V>*>*** so that each ***map*** holds all keys that have the same hash value.
28. (∗3) Implement a ***hash_map*** using Splay trees (see D. Sleator and R. E. Tarjan: *Self-Adjusting Binary Search Trees*, JACM, Vol. 32. 1985).
29. (∗2) Given a data structure describing a string-like entity:

```
struct St {
        int size;
        char type_indicator;
        char* buf;              // point to size characters
        st(const char* p);      // allocate and fill buf
};
```

Create 1000 ***St***s and use them as keys for a ***hash_map***. Devise a program to measure the performance of the ***hash_map***. Write a hash function (a ***Hash***; §17.6.2.3) specifically for ***St*** keys.

30. (∗2) Give at least four different ways of removing the ***erased*** elements from a ***hash_map***. You should use a standard library algorithm (§3.8, Chapter 18) to avoid an explicit loop.
31. (∗3) Implement a ***hash_map*** that erases elements immediately.
32. (∗2) The hash function presented in §17.6.2.3 doesn't always consider all of the representation of a key. When will part of a representation be ignored? Write a hash function that always considers all of the representations of a key. Give an example of when it might be wise to ignore part of a key and write a hash function that computes its value based only on the part of a key considered relevant.
33. (∗2.5) The code of hash functions tends to be similar: a loop gets more data and then hashes it. Define a ***Hash*** (§17.6.2.3) that gets its data by repeatedly calling a function that a user can define on a per-type basis. For example:

```
size_t res = 0;
while (size_t v = hash(key)) res = (res<<3)^v;
```

Here, a user can define ***hash*(*K*)** for each type ***K*** that needs to be hashed.

34. (∗3) Given some implementation of ***hash_map***, implement ***hash_multimap***, ***hash_set***, and ***hash_multiset***.
35. (∗2.5) Write a hash function intended to map uniformly distributed ***int*** values into hash values intended for a table size of about 1024. Given that function, devise a set of 1024 key values, all of which map to the same value.

18

Algorithms and Function Objects

Form is liberating.
– engineers´ proverb

Introduction — overview of standard algorithms — sequences — function objects — predicates — arithmetic objects — binders — member function objects — ***for_each*** — finding elements — ***count*** — comparing sequences — searching — copying — ***transform*** — replacing and removing elements — filling a sequence — reordering — ***swap*** — sorted sequences — ***binary_search*** — ***merge*** — set operations — ***min*** and ***max***— heaps — permutations — C-style algorithms — advice — exercises.

18.1 Introduction [algo.intro]

A container by itself is really not that interesting. To be genuinely useful, a container must be supported by basic operations such as finding its size, iterating, copying, sorting, and searching for elements. Fortunately, the standard library provides algorithms to serve the most common and basic needs that users have of containers.

This chapter summarizes the standard algorithms and gives a few examples of their uses, a presentation of the key principles and techniques used to express the algorithms in C++, and a more detailed explanation of a few key algorithms.

Function objects provide a mechanism through which a user can customize the behavior of the standard algorithms. Function objects supply key information that an algorithm needs in order to operate on a user's data. Consequently, emphasis is placed on how function objects can be defined and used.

18.2 Overview of Standard Library Algorithms [algo.summary]

At first glimpse, the standard library algorithms can appear overwhelming. However, there are just 60 of them. I have seen classes with more member functions. Furthermore, many algorithms share a common basic behavior and a common interface style that eases understanding. As with language features, a programmer should use the algorithms actually needed and understood – and only those. There are no awards for using the highest number of standard algorithms in a program. Nor are there awards for using standard algorithms in the most clever and obscure way. Remember, a primary aim of writing code is to make its meaning clear to the next person reading it – and that person just might be yourself a few years hence. On the other hand, when doing something with elements of a container, consider whether that action could be expressed as an algorithm in the style of the standard library. That algorithm might already exist. If you don't consider work in terms of general algorithms, you will reinvent the wheel.

Each algorithm is expressed as a template function (§13.3) or a set of template functions. In that way, an algorithm can operate on many kinds of sequences containing elements of a variety of types. Algorithms that return an iterator (§19.1) as a result generally use the end of an input sequence to indicate failure. For example:

```
void f(list<string>& ls)
{
    list<string>::const_iterator p = find(ls.begin(), ls.end(), "Fred");

    if (p == ls.end()) {
        // didn't find "Fred"
    }
    else {
        // here, p points to "Fred"
    }
}
```

The algorithms do not perform range checking on their input or output. Range errors must be prevented by other means (§18.3.1, §19.3). When an algorithm returns an iterator, that iterator is of the same type as one of its inputs. In particular, an algorithm's arguments control whether it returns a ***const_iterator*** or a non-***const iterator***. For example:

```
void f(list<int>& li, const list<string>& ls)
{
    list<int>::iterator p = find(li.begin(), li.end(), 42);
    list<string>::const_iterator q = find(ls.begin(), ls.end(), "Ring");
}
```

The algorithms in the standard library cover the most common general operations on containers such as traversals, sorting, searching, and inserting and removing elements. The standard algorithms are all in the ***std*** namespace and their declarations are found in ***<algorithm>***. Interestingly, most of the really common algorithms are so simple that the template functions are typically inline. This implies that the loops expressed by the algorithms benefit from aggressive per-function optimization.

The standard function objects are also in namespace ***std***, but their declarations are found in ***<functional>***. The function objects are designed to be easy to inline.

Nonmodifying sequence operations are used to extract information from a sequence or to find the positions of elements in a sequence:

Nonmodifying Sequence Operations (§18.5) <algorithm>	
for_each()	Do operation for each element in a sequence.
find()	Find first occurrence of a value in a sequence.
find_if()	Find first match of a predicate in a sequence.
find_first_of()	Find a value from one sequence in another.
adjacent_find()	Find an adjacent pair of values.
count()	Count occurrences of a value in a sequence.
count_if()	Count matches of a predicate in a sequence.
mismatch()	Find the first elements for which two sequences differ.
equal()	True if the elements of two sequences are pairwise equal.
search()	Find the first occurrence of a sequence as a subsequence.
find_end()	Find the last occurrence of a sequence as a subsequence.
search_n()	Find the ***n***th occurrence of a value in a sequence.

Most algorithms allow a user to specify the actual action performed for each element or pair of elements. This makes the algorithms much more general and useful than they appear at first glance. In particular, a user can supply the criteria used for equality and difference (§18.4.2). Where reasonable, the most common and useful action is provided as a default.

Modifying sequence operations have little in common beyond the obvious fact that they might change the values of elements of a sequence:

Modifying Sequence Operations (§18.6) <algorithm>	
transform()	Apply an operation to every element in a sequence.
copy()	Copy a sequence starting with its first element.
copy_backward()	Copy a sequence starting with its last element.
swap()	Swap two elements.
iter_swap()	Swap two elements pointed to by iterators.
swap_ranges()	Swap elements of two sequences.
replace()	Replace elements with a given value.
replace_if()	Replace elements matching a predicate.
replace_copy()	Copy sequence replacing elements with a given value.
replace_copy_if()	Copy sequence replacing elements matching a predicate.
fill()	Replace every element with a given value.
fill_n()	Replace first ***n*** elements with a given value.
generate()	Replace every element with the result of an operation.
generate_n()	Replace first ***n*** elements with the result of an operation.
remove()	Remove elements with a given value.
remove_if()	Remove elements matching a predicate.

Modifying Sequence Operations (continued) (§18.6) <algorithm>	
remove_copy()	Copy a sequence removing elements with a given value.
remove_copy_if()	Copy a sequence removing elements matching a predicate.
unique()	Remove equal adjacent elements.
unique_copy()	Copy a sequence removing equal adjacent elements.
reverse()	Reverse the order of elements.
reverse_copy()	Copy a sequence into reverse order.
rotate()	Rotate elements.
rotate_copy()	Copy a sequence into a rotated sequence.
random_shuffle()	Move elements into a uniform distribution.

Every good design shows traces of the personal traits and interests of its designer. The containers and algorithms in the standard library clearly reflect a strong concern for classical data structures and the design of algorithms. The standard library provides not only the bare minimum of containers and algorithms needed by essentially every programmer. It also includes many of the tools used to provide those algorithms and needed to extend the library beyond that minimum.

The emphasis here is not on the design of algorithms or even on the use of any but the simplest and most obvious algorithms. For information on the design and analysis of algorithms, you should look elsewhere (for example, [Knuth,1968] and [Tarjan,1983]). Instead, this chapter lists the algorithms offered by the standard library and explains how they are expressed in C++. This focus allows someone who understands algorithms to use the library well and to extend it in the spirit in which it was built.

The standard library provides a variety of operations for sorting, searching, and manipulating sequences based on an ordering:

Sorted Sequences (§18.7) <algorithm>	
sort()	Sort with good average efficiency.
stable_sort()	Sort maintaining order of equal elements.
partial_sort()	Get the first part of sequence into order.
partial_sort_copy()	Copy getting the first part of output into order.
nth_element()	Put the nth element in its proper place.
lower_bound()	Find the first occurrence of a value.
upper_bound()	Find the first element larger than a value.
equal_range()	Find a subsequence with a given value.
binary_search()	Is a given value in a sorted sequence?
merge()	Merge two sorted sequences.
inplace_merge()	Merge two consecutive sorted subsequences.
partition()	Place elements matching a predicate first.
stable_partition()	Place elements matching a predicate first, preserving relative order.

Set Algorithms (§18.7.5) <algorithm>	
includes()	True if a sequence is a subsequence of another.
set_union()	Construct a sorted union.
set_intersection()	Construct a sorted intersection.
set_difference()	Construct a sorted sequence of elements in the first but not the second sequence.
set_symmetric_difference()	Construct a sorted sequence of elements in one but not both sequences.

Heap operations keep a sequence in a state that makes it easy to sort when necessary:

Heap Operations (§18.8) <algorithm>	
make_heap()	Make sequence ready to be used as a heap.
push_heap()	Add element to heap.
pop_heap()	Remove element from heap.
sort_heap()	Sort the heap.

The library provides a few algorithms for selecting elements based on a comparison:

Minimum and Maximum (§18.9) <algorithm>	
min()	Smaller of two values.
max()	Larger of two values.
min_element()	Smallest value in sequence.
max_element()	Largest value in sequence.
lexicographical_compare()	Lexicographically first of two sequences.

Finally, the library provides ways of permuting a sequence:

Permutations (§18.10) <algorithm>	
next_permutation()	Next permutation in lexicographical order.
prev_permutation()	Previous permutation in lexicographical order.

In addition, a few generalized numerical algorithms are provided in ***<numeric>*** (§22.6).

In the description of algorithms, the template parameter names are significant. ***In***, ***Out***, ***For***, ***Bi***, and ***Ran*** mean input iterator, output iterator, forward iterator, bidirectional iterator, and random-access iterator, respectively (§19.2.1). ***Pred*** means unary predicate, ***BinPred*** means binary predicate (§18.4.2), ***Cmp*** means a comparison function (§17.1.4.1, §18.7.1), ***Op*** means unary operation, and ***BinOp*** means binary operation (§18.4). Conventionally, much longer names have been used for template parameters. However, I find that after only a brief acquaintance with the standard library, those long names decrease readability rather than enhancing it.

A random-access iterator can be used as a bidirectional iterator, a bidirectional iterator as a forward iterator, and a forward iterator as an input or an output iterator (§19.2.1). Passing a type that doesn't provide the required operations will cause template-instantiation-time errors (§C.13.7). Providing a type that has the right operations with the wrong semantics will cause unpredictable run-time behavior (§17.1.4).

18.3 Sequences and Containers [algo.seq]

It is a good general principle that the most common use of something should also be the shortest, the easiest to express, and the safest. The standard library violates this principle in the name of generality. For a standard library, generality is essential. For example, we can find the first two occurrences of *42* in a list like this:

```
void f(list<int>& li)
{
    list<int>::iterator p = find(li.begin(), li.end(), 42);      // first occurrence
    if (p != li.end()) {
        list<int>::iterator q = find(++p, li.end(), 42);         // second occurrence
        // ...
    }
    // ...
}
```

Had *find*( ) been expressed as an operation on a container, we would have needed some additional mechanism for finding the second occurrence. Importantly, generalizing such an ''additional mechanism'' for every container and every algorithm is hard. Instead, standard library algorithms work on sequences of elements. That is, the input of an algorithm is expressed as a pair of iterators that delineate a sequence. The first iterator refers to the first element of the sequence, and the second refers to a point one-beyond-the-last element (§3.8, §19.2). Such a sequence is called ''half open'' because it includes the first value mentioned and not the second. A half-open sequence allows many algorithms to be expressed without making the empty sequence a special case.

A sequence – especially a sequence in which random access is possible – is often called a *range*. Traditional mathematical notations for a half-open range are [*first*, *last*) and [*first*, *last*[. Importantly, a sequence can be the elements of a container or a subsequence of a container. Further, some sequences, such as I/O streams, are not containers. However, algorithms expressed in terms of sequences work just fine.

18.3.1 Input Sequences [algo.range]

Writing *x*.*begin*( ), *x*.*end*( ) to express ''all the elements of *x*'' is common, tedious, and can even be error-prone. For example, when several iterators are used, it is too easy to provide an algorithm with a pair of arguments that does not constitute a sequence:

```
void f(list<string>& fruit, list<string>& citrus)
{
    typedef list<string>::const_iterator LI;

    LI p1 = find(fruit.begin(), citrus.end(), "apple");   // wrong! (different sequences)
    LI p2 = find(fruit.begin(), fruit.end(), "apple");    // ok
    LI p3 = find(citrus.begin(), citrus.end(), "pear");   // ok
    LI p4 = find(p2, p3, "peach");                        // wrong! (different sequences)
    // ...
}
```

In this example there are two errors. The first is obvious (once you suspect an error), but it isn't

easily detected by a compiler. The second is hard to spot in real code even for an experienced programmer. Cutting down on the number of explicit iterators used alleviates this problem. Here, I outline an approach to dealing with this problem by making the notion of an input sequence explicit. However, to keep the discussion of standard algorithms strictly within the bounds of the standard library, I do not use explicit input sequences when presenting algorithms in this chapter.

The key idea is to be explicit about taking a sequence as input. For example:

```
template<class In, class T> In find(In first, In last, const T& v)     // standard
{
        while (first!=last && *first!=v) ++first;
        return first;
}

template<class In, class T> In find(Iseq<In> r, const T& v)            // extension
{
        return find(r.first,r.second,v);
}
```

In general, overloading (§13.3.2) allows the input-sequence version of an algorithm to be preferred when an ***Iseq*** argument is used.

Naturally, an input sequence is implemented as a pair (§17.4.1.2) of iterators:

```
template<class In> struct Iseq : public pair<In,In> {
        Iseq(In i1, In i2) : pair<In,In>(i1,i2) { }
};
```

We can explicitly make the ***Iseq*** needed to invoke the second version of ***find*( )**:

```
        LI p = find(Iseq<LI>(fruit.begin(),fruit.end()),"apple");
```

However, that is even more tedious than calling the original ***find*( )** directly. Simple helper functions relieve the tedium. In particular, the ***Iseq*** of a container is the sequence of elements from its ***begin*( )** to its ***end*( )**:

```
template<class C> Iseq<C::iterator_type> iseq(C& c)                    // for container
{
        return Iseq<C::iterator_type>(c.begin(),c.end());
}
```

This allows us to express algorithms on containers compactly and without repetition. For example:

```
void f(list<string>& ls)
{
        list<string>::iterator p = find(ls.begin(),ls.end(),"standard");
        list<string>::iterator q = find (iseq(ls),"extension");
        // ..
}
```

It is easy to define versions of ***iseq*( )** that produce ***Iseq***s for arrays, input streams, etc. (§18.13[6]).

The key benefit of ***Iseq*** is that it makes the notion of an input sequence explicit. The immediate practical effect is that use of ***iseq*( )** eliminates much of the tedious and error-prone repetition needed to express every input sequence as a pair of iterators.

The notion of an output sequence is also useful. However, it is less simple and less immediately useful than the notion of an input sequence (§18.13[7]; see also §19.2.4).

18.4 Function Objects [algo.fct]

Many algorithms operate on sequences using iterators and values only. For example, we can *find*( ) the first element with the value *7* in a sequence like this:

```
void f(list<int>& c)
{
    list<int>::iterator p = find(c.begin(), c.end(), 7);
    // ...
}
```

To do more interesting things we want the algorithms to execute code that we supply (§3.8.4). For example, we can find the first element in a sequence with a value of less than *7* like this:

```
bool less_than_7(int v)
{
    return v<7;
}

void f(list<int>& c)
{
    list<int>::iterator p = find_if(c.begin(), c.end(), less_than_7);
    // ...
}
```

There are many obvious uses for functions passed as arguments: logical predicates, arithmetic operations, operations for extracting information from elements, etc. It is neither convenient nor efficient to write a separate function for each use. Nor is a function logically sufficient to express all that we would like to express. Often, the function called for each element needs to keep data between invocations and to return the result of many applications. A member function of a class serves such needs better than a free-standing function does because its object can hold data. In addition, the class can provide operations for initializing and extracting such data.

Consider how to write a function – or rather a function-like class – to calculate a sum:

```
template<class T> class Sum {
    T res;
public:
    Sum(T i = 0) : res(i) { }                // initialize
    void operator()(T x) { res += x; }          // accumulate
    T result() const { return res; }         // return sum
};
```

Clearly, *Sum* is designed for arithmetic types for which initialization by *0* and += are defined. For example:

```
void f(list<double>& ld)
{
    Sum<double> s;
    s = for_each(ld.begin(), ld.end(), s);          // invoke s() for each element of ld
    cout << "the sum is" << s.result() << '\n';
}
```

Here, *for_each*( ) (§18.5.1) invokes *Sum<double>::operator*( )(*double*) for each element of *ld* and returns the object passed as its third argument.

The key reason this works is that *for_each*( ) doesn't actually assume its third argument to be a function. It simply assumes that its third argument is something that can be called with an appropriate argument. A suitably-defined object serves as well as – and often better than – a function. For example, it is easier to inline the application operator of a class than to inline a function passed as a pointer to function. Consequently, function objects often execute faster than do ordinary functions. An object of a class with an application operator (§11.9) is called a *function-like object*, a *functor*, or simply a *function object*.

18.4.1 Function Object Bases [algo.bases]

The standard library provides many useful function objects. To aid the writing of function objects, the library provides a couple of base classes:

```
template <class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

template <class Arg, class Arg2, class Res> struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};
```

The purpose of these classes is to provide standard names for the argument and return types for use by users of classes derived from *unary_function* and *binary_function*. Using these bases consistently the way the standard library does will save the programmer from discovering the hard way why they are useful (§18.4.4.1).

18.4.2 Predicates [algo.pred]

A predicate is a function object (or a function) that returns a *bool*. For example, *<functional>* defines:

```
template <class T> struct logical_not : public unary_function<T, bool> {
    bool operator()(const T& x) const { return !x; }
};

template <class T> struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x<y; }
};
```

Unary and binary predicates are often useful in combination with algorithms. For example, we can compare two sequences, looking for the first element of one that is not less than its corresponding element in the other:

```
void f(vector<int>& vi, list<int>& li)
{
    typedef list<int>::iterator LI;
    typedef vector<int>::iterator VI;
    pair<VI,LI> p1 = mismatch(vi.begin(),vi.end(),li.begin(),less<int>());
    // ...
}
```

The ***mismatch*( )** algorithm applies its binary predicate repeatedly to pairs of corresponding elements until it fails (§18.5.4). It then returns the iterators for the elements that failed the comparison. Because an object is needed rather than a type, ***less<int>*( )** (with the parentheses) is used rather than the tempting ***less<int>***.

Instead of finding the first element ***not less*** than its corresponding element in the other sequence, we might like to find the first element ***less*** than its corresponding element. We can do this by presenting the sequences to ***mismatch*( )** in the opposite order:

```
pair<LI,VI> p2 = mismatch(li.begin(),li.end(),vi.begin(),less<int>());
```

or we can use the complementary predicate ***greater_equal***:

```
p1 = mismatch(vi.begin(),vi.end(),li.begin(),greater_equal<int>());
```

In §18.4.4.4, I show how to express the predicate ‘‘not less.’’

18.4.2.1 Overview of Predicates [algo.pred.std]

In ***<functional>***, the standard library supplies a few common predicates:

Predicates <functional>		
equal_to	Binary	arg1==arg2
not_equal_to	Binary	arg1!=arg2
greater	Binary	arg1>arg2
less	Binary	arg1<arg2
greater_equal	Binary	arg1>=arg2
less_equal	Binary	arg1<=arg2
logical_and	Binary	arg1&&arg2
logical_or	Binary	arg1||arg2
logical_not	Unary	!arg

The definitions of ***less*** and ***logical_not*** are presented in §18.4.2.

In addition to the library-provided predicates, users can write their own. Such user-supplied predicates are essential for simple and elegant use of the standard libraries and algorithms. The ability to define predicates is particularly important when we want to use algorithms for classes designed without thought of the standard library and its algorithms. For example, consider a variant of the ***Club*** class from §10.4.6:

```
class Person { /* ... */ };

struct Club {
    string name;
    list<Person*> members;
    list<Person*> officers;
    // ...
    Club(const name& n);
};
```

Looking for a ***Club*** with a given name in a ***list<Club>*** is clearly a reasonable thing to do. However, the standard library algorithm ***find_if( )*** doesn't know about ***Club***s. The library algorithms know how to test for equality, but we don't want to find a ***Club*** based on its complete value. Rather, we want to use ***Club::name*** as the key. So we write a predicate to reflect that:

```
class Club_eq : public unary_function<Club, bool> {
    string s;
public:
    explicit Club_eq(const string& ss) : s(ss) { }
    bool operator()(const Club& c) const { return c.name==s; }
};
```

Defining useful predicates is simple. Once suitable predicates have been defined for user-defined types, their use with the standard algorithms is as simple and efficient as examples involving containers of simple types. For example:

```
void f(list<Club>& lc)
{
    typedef list<Club>::iterator LCI;
    LCI p = find_if(lc.begin(), lc.end(), Club_eq("Dining Philosophers"));
    // ...
}
```

18.4.3 Arithmetic Function Objects [algo.arithmetic]

When dealing with numeric classes, it is sometimes useful to have the standard arithmetic functions available as function objects. Consequently, in ***<functional>*** the standard library provides:

Arithmetic Operations <functional>		
plus	Binary	arg1+arg2
minus	Binary	arg1–arg2
multiplies	Binary	arg1*arg2
divides	Binary	arg1/arg2
modulus	Binary	arg1%arg2
negate	Unary	–arg

We might use ***multiplies*** to multiply elements in two vectors, thereby producing a third:

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
        transform(a.begin(), a.end(), b.begin(), back_inserter(res), multiplies<double>());
}
```

The *back_inserter*( ) is described in §19.2.4. A few numerical algorithms can be found in §22.6.

18.4.4 Binders, Adapters, and Negaters [algo.adapter]

We can use predicates and arithmetic function objects we have written ourselves and rely on the ones provided by the standard library. However, when we need a new predicate we often find that the new predicate is a minor variation of an existing one. The standard library supports the composition of function objects:

§18.4.4.1 A *binder* allows a two-argument function object to be used as a single-argument function by binding one argument to a value.

§18.4.4.2 A *member function adapter* allows a member function to be used as an argument to algorithms.

§18.4.4.3 A *pointer to function adapter* allows a pointer to function to be used as an argument to algorithms.

§18.4.4.4 A *negater* allows us to express the opposite of a predicate.

Collectively, these function objects are referred to as ***adapters***. These adapters all have a common structure relying on the function object bases ***unary_function*** and ***binary_function*** (§18.4.1). For each of these adapters, a helper function is provided to take a function object as an argument and return a suitable function object. When invoked by its *operator*( )( ), that function object will perform the desired action. That is, an adapter is a simple form of a higher-order function: it takes a function argument and produces a new function from it:

Binders, Adapters, and Negaters <functional>		
bind2nd(y)	binder2nd	Call binary function with *y* as 2nd argument.
bind1st(x)	*binder1st*	Call binary function with *x* as 1st argument.
mem_fun()	*mem_fun_t*	Call 0-arg member through pointer.
	mem_fun1_t	Call unary member through pointer.
	const_mem_fun_t	Call 0-arg const member through pointer.
	const_mem_fun1_t	Call unary const member through pointer.
mem_fun_ref()	*mem_fun_ref_t*	Call 0-arg member through reference.
	mem_fun1_ref_t	Call unary member through reference.
	const_mem_fun_ref_t	Call 0-arg const member through reference.
	const_mem_fun1_ref_t	Call unary const member through reference.
ptr_fun()	*pointer_to_unary_function*	Call unary pointer to function.
ptr_fun()	*pointer_to_binary_function*	Call binary pointer to function.
not1()	*unary_negate*	Negate unary predicate.
not2()	*binary_negate*	Negate binary predicate.

18.4.4.1 Binders [algo.binder]

Binary predicates such as *less* (§18.4.2) are useful and flexible. However, we soon discover that the most useful kind of predicate is one that compares a fixed argument repeatedly against a container element. The *less_than_7*( ) function (§18.4) is a typical example. The *less* operation needs two arguments explicitly provided in each call, so it is not immediately useful. Instead, we might define:

```
template <class T> class less_than : public unary_function<T, bool> {
    T arg2;
public:
    explicit less_than(const T& x) : arg2(x) { }
    bool operator()(const T& x) const { return x<arg2; }
};
```

We can now write:

```
void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(), c.end(), less_than<int>(7));
    // ...
}
```

We must write *less_than<int>*(*7*) rather than *less_than*(*7*) because the template argument *<int>* cannot be deduced from the type of the constructor argument (*7*) (§13.3.1).

The *less_than* predicate is generally useful. Importantly, we defined it by fixing or binding the second argument of *less*. Such composition by binding an argument is so common, useful, and occasionally tedious that the standard library provides a standard class for doing it:

```
template <class BinOp>
class binder2nd : public unary_function<BinOp::first_argument_type, BinOp::result_type> {
protected:
    BinOp op;
    typename BinOp::second_argument_type arg2;
public:
    binder2nd(const BinOp& x, const typename BinOp::second_argument_type& v)
        : op(x), arg2(v) { }
    result_type operator()(const argument_type& x) const { return op(x,arg2); }
};

template <class BinOp, class T> binder2nd<BinOp> bind2nd(const BinOp& op, const T& v)
{
    return binder2nd<BinOp>(op,v);
}
```

For example, we can use *bind2nd*( ) to create the unary predicate ‘‘less than *7*’’ from the binary predicate ‘‘less’’ and the value *7*:

```
void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(), c.end(), bind2nd(less<int>(), 7));
    // ...
}
```

Is this readable? Is this efficient? Given an average C++ implementation, this version is actually more efficient in time and space than is the original version using the function *less_than_7*( ) from §18.4! The comparison is easily inlined.

The notation is logical, but it does take some getting used to. Often, the definition of a named operation with a bound argument is worthwhile after all:

```
template <class T> struct less_than : public binder2nd<less<T>, T> {
    explicit less_than(const T& x) : binder2nd(less<T>(), x) { }
};

void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(), c.end(), less_than<int>(7));
    // ...
}
```

It is important to define *less_than* in terms of *less* rather than using < directly. That way, *less_than* benefits from any specializations that *less* might have (§13.5, §19.2.2).

In parallel to *bind2nd*( ) and *binder2nd*, *<functional>* provides *bind1st*( ) and *binder1st* for binding the first argument of a binary function.

By binding an argument, *bind1st*( ) and *bind2nd*( ) perform a service very similar to what is commonly referred to as *Currying*.

18.4.4.2 Member Function Adapters [algo.memfct]

Most algorithms invoke a standard or user-defined operation. Naturally, users often want to invoke a member function. For example (§3.8.5):

```
void draw_all(list<Shape*>& lsp)
{
    for_each(c.begin(), c.end(), &Shape::draw);  // oops! error
}
```

The problem is that a member function *mf*( ) needs to be invoked for an object: *p->mf*( ). However, algorithms such as *for_each*( ) invoke their function operands by simple application: *f*( ). Consequently, we need a convenient and efficient way of creating something that allows an algorithm to invoke a member function. The alternative would be to duplicate the set of algorithms: one version for member functions plus one for ordinary functions. Worse, we'd need additional versions of algorithms for containers of objects (rather than pointers to objects). As for the binders (§18.4.4.1), this problem is solved by a class plus a function. First, consider the common case in which we want to call a member function taking no arguments for the elements of a container of pointers:

```
template<class R, class T> class mem_fun_t : public unary_function<T*,R> {
    R (T::*pmf)();
public:
    explicit mem_fun_t(R (T::*p)()) :pmf(p) {}
    R operator()(T* p) const { return (p->*pmf)(); } // call through pointer
};

template<class R, class T> mem_fun_t<R,T> mem_fun(R (T::*f)())
{
    return mem_fun_t<R,T>(f);
}
```

This handles the *Shape::draw()* example:

```
void draw_all(list<Shape*>& lsp)    // call 0-argument member through pointer to object
{
    for_each(lsp.begin(),lsp.end(),mem_fun(&Shape::draw));    // draw all shapes
}
```

In addition, we need a class and a *mem_fun()* function for handling a member function taking an argument. We also need versions to be called directly for an object rather than through a pointer; these are named *mem_fun_ref()*. Finally, we need versions for *const* member functions:

```
template<class R, class T> mem_fun_t<R,T> mem_fun(R (T::*f)());
// and versions for unary member, for const member, and const unary member (see table in §18.4.4)

template<class R, class T> mem_fun_ref_t<R,T> mem_fun_ref(R (T::*f)());
// and versions for unary member, for const member, and const unary member (see table in §18.4.4)
```

Given these member function adapters from *<functional>*, we can write:

```
void f(list<string>& ls)    // use member function that takes no argument for object
{
    typedef list<string>::iterator LSI;
    LSI p = find_if(ls.begin(),ls.end(),mem_fun_ref(&string::empty));// find ""
}

void rotate_all(list<Shape*>& ls, int angle)
    // use member function that takes one argument through pointer to object
{
    for_each(ls.begin(),ls.end(),bind2nd(mem_fun(&Shape::rotate),angle));
}
```

The standard library need not deal with member functions taking more than one argument because no standard library algorithm takes a function with more than two arguments as operands.

18.4.4.3 Pointer to Function Adapters [algo.ptof]

An algorithm doesn't care whether a ''function argument'' is a function, a pointer to function, or a function object. However, a binder (§18.4.4.1) does care because it needs to store a copy for later use. Consequently, the standard library supplies two adapters to allow pointers to functions to be used together with the standard algorithms in *<functional>*. The definition and implementation

closely follows that of the member function adapters (§18.4.4.2). Again, a pair of functions and a pair of classes are used:

```
template <class A, class R> pointer_to_unary_function<A,R> ptr_fun(R (*f)(A));

template <class A, class A2, class R>
    pointer_to_binary_function<A,A2,R> ptr_fun(R (*f)(A, A2));
```

Given these pointer to function adapters, we can use ordinary functions together with binders:

```
class Record { /* ... */ };

bool name_key_eq(const Record&, const Record&);  // compare based on names
bool ssn_key_eq(const Record&, const Record&);   // compare based on number

void f(list<Record>& lr)   // use pointer to function
{
    typedef typename list<Record>::iterator LI;
    LI p = find_if(lr.begin(), lr.end(), bind2nd(ptr_fun(name_key_eq), "John Brown"));
    LI q = find_if(lr.begin(), lr.end(), bind2nd(ptr_fun(ssn_key_eq), 1234567890));
    // ...
}
```

This looks for elements of the list *lr* that match the keys *John Brown* and *1234567890*.

18.4.4.4 Negaters [algo.negate]

The predicate negaters are related to the binders in that they take an operation and produce a related operation from it. The definition and implementation of negaters follow the pattern of the member function adapters (§18.4.4.2). Their definitions are trivial, but their simplicity is obscured by the use of long standard names:

```
template <class Pred>
class unary_negate : public unary_function<typename Pred::argument_type, bool> {

    unary_function<argument_type, bool> op;
public:
    explicit unary_negate(const Pred& p) : op(p) { }
    bool operator()(const argument_type& x) const { return !op(x); }
};

template <class Pred>
class binary_negate : public binary_function<typename Pred::first_argument_type,
                                             typename Pred::second_argument_type, bool> {

    typedef first_argument_type Arg;
    typedef second_argument_type Arg2;

    binary_function<Arg, Arg2, bool> op;
public:
    explicit binary_negate(const Pred& p) : op(p) { }
    bool operator()(const Arg& x, const Arg2& y) const { return !op(x,y); }
};
```

```
template<class Pred> unary_negate<Pred> not1(const Pred& p);        // negate unary
template<class Pred> binary_negate<Pred> not2(const Pred& p);       // negate binary
```

These classes and functions are declared in *<functional>*. The names *first_argument_type*, *second_argument_type*, etc., come from the standard base classes *unary_function* and *binary_function*.

Like the binders, the negaters are most conveniently used indirectly through their helper functions. For example, we can express the binary predicate ''not less than'' and use it to find the first corresponding pair of elements whose first element is greater than or equal to its second:

```
void f(vector<int>& vi, list<int>& li) // revised example from §18.4.2
{
    // ...
    p1 = mismatch(vi.begin(), vi.end(), li.begin(), not2(less<int>()));
    // ...
}
```

That is, *p1* identifies the first pair of elements for which the predicate *not less than* failed.

Predicates deal with Boolean conditions, so there are no equivalents to the bitwise operators |, &, ^, and ~.

Naturally, binders, adapters, and negaters are useful in combination. For example:

```
extern "C" int strcmp(const char*, const char*);     // from <cstdlib>

void f(list<char*>& ls)     // use pointer to function
{
    typedef typename list<char*>::const_iterator LI;
    LI p = find_if(ls.begin(), ls.end(), not1(bind2nd(ptr_fun(strcmp), "funny")));
}
```

This finds an element of the list *ls* that contains the C-style string "*funny*". The negater is needed because *strcmp*( ) returns *0* when strings compare equal.

18.5 Nonmodifying Sequence Algorithms [algo.nonmodifying]

Nonmodifying sequence algorithms are the basic means for finding something in a sequence without writing a loop. In addition, they allow us to find out things about elements. These algorithms can take const-iterators (§19.2.1) and – with the excetion of *for_each*( ) – should not be used to invoke operations that modify the elements of the sequence.

18.5.1 For_each [algo.foreach]

We use a library to benefit from the work of others. Using a library function, class, algorithm, etc., saves the work of inventing, designing, writing, debugging, and documenting something. Using the standard library also makes the resulting code easier to read for others who are familiar with that library, but who would have to spend time and effort understanding home-brewed code.

A key benefit of the standard library algorithms is that they save the programmer from writing explicit loops. Loops can be tedious and error-prone. The *for_each*( ) algorithm is the simplest

algorithm in the sense that it does nothing but eliminate an explicit loop. It simply calls its operator argument for a sequence:

```
template<class In, class Op> Op for_each(In first, In last, Op f)
{
    while (first != last) f(*first++);
    return f;
}
```

What functions would people want to call this way? If you want to accumulate information from the elements, consider ***accumulate***( ) (§22.6). If you want to find something in a sequence, consider ***find***( ) and ***find_if***( ) (§18.5.2). If you change or remove elements, consider ***replace***( ) (§18.6.4) or ***remove***( ) (§18.6.5). In general, before using ***for_each***( ), consider if there is a more specialized algorithm that would do more for you.

The result of ***for_each***( ) is the function or function object passed as its third argument. As shown in the ***Sum*** example (§18.4), this allows information to be passed back to a caller.

One common use of ***for_each***( ) is to extract information from elements of a sequence. For example, consider collecting the names of any of a number of ***Club***s:

```
void extract(const list<Club>& lc, list<Person*>& off) // place the officers from 'lc' on 'off'
{
    for_each(lc.begin(), lc.end(), Extract_officers(off));
}
```

In parallel to the examples from §18.4 and §18.4.2, we define a function class that extracts the desired information. In this case, the names to be extracted are found in ***list<Person*>***s in our ***list<Club>***. Consequently, ***Extract_officers*** needs to copy the officers from a ***Club***'s ***officers*** list to our list:

```
class Extract_officers {
    list<Person*>& lst;
public:
    explicit Extract_officers(list<Person*>& x) : lst(x) { }

    void operator()(const Club& c)
        { copy(c.officers.begin(), c.officers.end(), back_inserter(lst)); }
};
```

We can now print out the names, again using ***for_each***( ):

```
void extract_and_print(const list<Club>& lc)
{
    list<Person*> off;
    extract(lc, off);
    for_each(off.begin(), off.end(), Print_name(cout));
}
```

Writing ***Print_name*** is left as an exercise (§18.13[4]).

The ***for_each***( ) algorithm is classified as nonmodifying because it doesn't explicitly modify a sequence. However, if applied to a non-***const*** sequence ***for_each***( )'s operation (its third argument) may change the elements of the sequence. For an example, see ***delete_ptr***( ) in §18.6.2.

18.5.2 The Find Family [algo.find]

The *find*( ) algorithms look through a sequence or a pair of sequences to find a value or a match on a predicate. The simple versions of *find*( ) look for a value or for a match with a predicate:

```
template<class In, class T> In find(In first, In last, const T& val);

template<class In, class Pred> In find_if(In first, In last, Pred p);
```

The algorithms *find*( ) and *find_if*( ) return an iterator to the first element that matches a value and a predicate, respectively. In fact, *find*( ) can be understood as the version of *find_if*( ) with the predicate ==. Why aren't they both called *find*( )? The reason is that function overloading cannot always distinguish calls of two template functions with the same number of arguments. Consider:

```
bool pred(int);

void f(vector<bool(*f)(int)>& v1, vector<int>& v2)
{
    find(v1.begin(),v1.end(),pred);          // find 'pred'
    find_if(v2.begin(),v2.end(),pred);       // find int for which pred() returns true
}
```

If *find*( ) and *find_if*( ) had had the same name, surprising ambiguities would have resulted. In general, the *_if* suffix is used to indicate that an algorithm takes a predicate.

The *find_first_of*( ) algorithm finds the first element of a sequence that has a match in a second sequence:

```
template<class For, class For2>
    For find_first_of(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For find_first_of(For first, For last, For2 first2, For2 last2, BinPred p);
```

For example:

```
int x[] = { 1,3,4 };
int y[] = { 0,2,3,4,5};

void f()
{
    int* p = find_first_of(x,x+3,y,y+5);        // p = &x[1]
    int* q = find_first_of(p+1,x+3,y,y+5);      // q = &x[2]
}
```

The pointer *p* will point to *x*[*1*] because *3* is the first element of *x* with a match in *y*. Similarly, *q* will point to *x*[*2*].

The *adjacent_find*( ) algorithm finds a pair of adjacent matching values:

```
template<class For> For adjacent_find(For first, For last);

template<class For, class BinPred> For adjacent_find(For first, For last, BinPred p);
```

The return value is an iterator to the first matching element. For example:

```
void f(vector<string>& text)
{
    vector<string>::iterator p = adjacent_find(text.begin(), text.end(), "the");
    if (p != text.end()) {
        // I duplicated "the" again!
    }
}
```

18.5.3 Count [algo.count]

The *count*( ) and *count_if*( ) algorithms count occurrences of a value in a sequence:

```
template<class In, class T>
    iterator_traits<In>::difference_type count(In first, In last, const T& val);

template<class In, class Pred>
    iterator_traits<In>::difference_type count_if(In first, In last, Pred p);
```

The return type of *count*( ) is interesting. Consider an obvious and somewhat simple-minded version of *count*( ):

```
template<class In, class T> int count(In first, In last, const T& val)
{
    int res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

The problem is that an *int* might not be the right type for the result. On a machine with small *int*s, there might be too many elements in the sequence for *count*( ) to fit in an *int*. Conversely, a high-performance implementation on a specialized machine might prefer to keep the count in a *short*.

Clearly, the number of elements in the sequence cannot be larger than the maximum difference between its iterators (§19.2.1). Consequently, the first idea for a solution to this problem is to define the return type as

```
typename In::difference_type
```

However, a standard algorithm should be applicable to built-in arrays as well as to standard containers. For example:

```
void f(const char* p, int size)
{
    int n = count(p,p+size, 'e');    // count the number of occurrences of the letter 'e'
}
```

Unfortunately, *int**::*difference_type* is not valid C++. This problem is solved by partial specialization of an *iterator_traits* (§19.2.2).

18.5.4 Equal and Mismatch [algo.equal]

The *equal*( ) and *mismatch*( ) algorithms compare two sequences:

```
template<class In, class In2> bool equal(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
    bool equal(In first, In last, In2 first2, BinPred p);

template<class In, class In2> pair<In, In2> mismatch(In first, In last, In2 first2);

template<class In, class In2, class BinPred>
    pair<In, In2> mismatch(In first, In last, In2 first2, BinPred p);
```

The *equal*( ) algorithm simply tells whether all corresponding pairs of elements of two sequences compare equal; *mismatch*( ) looks for the first pair of elements that compares unequal and returns iterators to those elements. No end is specified for the second sequence; that is, there is no *last2*. Instead, it is assumed that there are at least as many elements in the second sequence as in the first and *first2*+(*last*-*first*) is used as *last2*. This technique is used throughout the standard library, where pairs of sequences are used for operations on pairs of elements.

As shown in §18.5.1, these algorithms are even more useful than they appear at first glance because the user can supply predicates defining what it means to be equal and to match.

Note that the sequences need not be of the same type. For example:

```
void f(list<int>& li, vector<double>& vd)
{
    bool b = equal(li.begin(), li.end(), vd.begin());
}
```

All that is required is that the elements be acceptable as operands of the predicate.

The two versions of *mismatch*( ) differ only in their use of predicates. In fact, we could implement them as one function with a default template argument:

```
template<class In, class In2, class BinPred>
pair<In, In2> mismatch(In first, In last, In2 first2,
                       BinPred p = equal_to<In::value_type, In2::value_type>()) // §18.4.2.1
{
    while (first != last && p(*first, *first2)) {
        ++first;
        ++first2;
    }
    return pair<In, In2>(first, first2);
}
```

The difference between having two functions and having one with a default argument can be observed by someone taking pointers to functions. However, thinking of many of the variants of the standard algorithms as simply ‘‘the version with the default predicate’’ roughly halves the number of template functions that need to be remembered.

18.5.5 Search [algo.search]

The *search*( ), *search_n*( ), and *find_end*( ) algorithms find one sequence as a subsequence in another:

```
template<class For, class For2>
    For search(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For search(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class For2>
    For find_end(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For find_end(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class Size, class T>
    For search_n(For first, For last, Size n, const T& val);

template<class For, class Size, class T, class BinPred>
    For search_n(For first, For last, Size n, const T& val, BinPred p);
```

The *search*( ) algorithm looks for its second sequence as a subsequence of its first. If that second sequence is found, an iterator for the first matching element in the first sequence is returned. The end of sequence (*last*) is returned to represent ‘‘not found.’’ Thus, the return value is always in the [*first*, *last*] sequence. For example:

```
string quote("Why waste time learning, when ignorance is instantaneous?");

bool in_quote(const string& s)
{
    char* p = search(quote.begin(), quote.end(), s.begin(), s.end());  // find s in quote
    return p!=quote.end();
}

void g()
{
    bool b1 = in_quote("learning");      // b1 = true
    bool b2 = in_quote("lemming");       // b2 = false
}
```

Thus, *search*( ) is an operation for finding a substring generalized to all sequences. This implies that *search*( ) is a very useful algorithm.

The *find_end*( ) algorithm looks for its second input sequence as a subsequence of its first input sequence. If that second sequence is found, *find_end*( ) returns an iterator pointing to the last match in its first input. In other words, *find_end*( ) is *search*( ) ‘‘backwards.’’ It finds the last occurrence of its second input sequence in its first input sequence, rather than the first occurrence of its second sequence.

The *search_n*( ) algorithm finds a sequence of at least ***n*** matches for its ***value*** argument in the sequence. It returns an iterator to the first element of the sequence of ***n*** matches.

18.6 Modifying Sequence Algorithms [algo.modifying]

If you want to change a sequence, you can explicitly iterate through it. You can then modify values. Wherever possible, however, we prefer to avoid this kind of programming in favor of simpler and more systematic styles of programming. The alternative is algorithms that traverse sequences performing specific tasks. The nonmodifying algorithms (§18.5) serve this need when we just read from the sequence. The modifying sequence algorithms are provided to do the most common forms of updates. Some update a sequence, while others produce a new sequence based on information found during a traversal.

Standard algorithms work on data structures through iterators. This implies that inserting a new element into a container or deleting one is not easy. For example, given only an iterator, how can we find the container from which to remove the element pointed to? Unless special iterators are used (e.g., inserters, §3.8, §19.2.4), operations through iterators do not change the size of a container. Instead of inserting and deleting elements, the algorithms change the values of elements, swap elements, and copy elements. Even *remove*( ) operates by overwriting the elements to be removed (§18.6.5). In general, the fundamental modifying operations produce outputs that are modified copies of their inputs. The algorithms that appear to modify a sequence are variants that copy within a sequence.

18.6.1 Copy [algo.copy]

Copying is the simplest way to produce one sequence from another. The definitions of the basic copy operations are trivial:

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first != last) *res++ = *first++;
    return res;
}

template<class Bi, class Bi2> Bi2 copy_backward(Bi first, Bi last, Bi2 res)
{
    while (first != last) *--res = *--last;
    return res;
}
```

The target of a copy algorithm need not be a container. Anything that can be described by an output iterator (§19.2.6) will do. For example:

```
void f(list<Club>& lc, ostream& os)
{
    copy(lc.begin(), lc.end(), ostream_iterator<Club>(os));
}
```

To read a sequence, we need a sequence describing where to begin and where to end. To write, we need only an iterator describing where to write to. However, we must take care not to write beyond the end of the target. One way to ensure that we don't do this is to use an inserter (§19.2.4) to grow the target as needed. For example:

```
void f(vector<char>& vs)
{
    vector<char> v;

    copy(vs.begin(), vs.end(), v.begin());          // might overwrite end of v
    copy(vs.begin(), vs.end(), back_inserter(v));   // add elements from vs to end of v
}
```

The input sequence and the output sequence may overlap. We use *copy*( ) when the sequences do not overlap or if the end of the output sequence is in the input sequence. We use *copy_backward*( ) when the beginning of the output sequence is in the input sequence. In that way, no element is overwritten until after it has been copied. See also §18.13[13].

Naturally, to copy something backwards we need a bidirectional iterator (§19.2.1) for both the input and the output sequences. For example:

```
void f(vector<char>& vc)
{
    vector<char> v(vc.size());

    copy_backward(vc.begin(), vc.end(), output_iterator<char>(cout)); // error

    copy_backward(vc.begin(), vc.end(), v.end());                      // ok
    copy(v.begin(), v.end(), ostream_iterator<char>(os));

}
```

Often, we want to copy only elements that fulfill some criterion. Unfortunately, *copy_if*( ) was somehow dropped from the set of algorithms provided by the standard library (mea culpa). On the other hand, it is trivial to define:

```
template<class In, class Out, class Pred> Out copy_if(In first, In last, Out res, Pred p)
{
    while (first != last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

Now if we want to print elements with a value larger than ***n***, we can do it like this:

```
void f(list<int>&ld, int n, ostream& os)
{
    copy_if(ld.begin(), ld.end(), ostream_iterator<int>(os), bind2nd(greater<int>(), n));
}
```

See also *remove_copy_if*( ) (§18.6.5).

18.6.2 Transform [algo.transform]

Somewhat confusingly, *transform*( ) doesn't necessarily change its input. Instead, it produces an output that is a transformation of its input based on a user-supplied operation:

```
template<class In, class Out, class Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first != last) *res++ = op(*first++);
    return res;
}

template<class In, class In2, class Out, class BinOp>
Out transform(In first, In last, In2 first2, Out res, BinOp op)
{
    while (first != last) *res++ = op(*first++, *first2++);
    return res;
}
```

The *transform*( ) that reads a single sequence to produce its output is rather similar to *copy*( ). Instead of writing its element, it writes the result of its operation on that element. Thus, we could have defined *copy*( ) as *transform*( ) with an operation that returns its argument:

```
template<class T> T identity(const T& x) { return x; }

template<class In, class Out> Out copy(In first, In last, Out res)
{
    return transform(first, last, res, identity);
}
```

Another way to view *transform*( ) is as a variant of *for_each* that explicitly produces output. For example, we can produce a list of name *string*s from a list of *Club*s using *transform*( ):

```
string nameof(const Club& c)    // extract name string
{
    return c.name;
}

void f(list<Club>& lc)
{
    transform(lc.begin(), lc.end(), ostream_iterator<string>(cout), nameof);
}
```

One reason *transform*( ) is called ''transform'' is that the result of the operation is often written back to where the argument came from. As an example, consider deleting the objects pointed to by a set of pointers:

```
template<class T> T* delete_ptr(T* p) { delete p; return 0; }

void purge(deque<Shape*>& s)
{
    transform(s.begin(), s.end(), s.begin(), delete_ptr);
    // ...
}
```

The *transform*( ) algorithm always produces an output sequence. Here, I directed the result back to the input sequence so that *delete_ptr*(*p*) has the effect *p=delete_ptr*(*p*). This was why I chose to return *0* from *delete_ptr*( ).

The *transform*( ) algorithm that takes two sequences allows people to combine information from two sources. For example, an animation may have a routine that updates the position of a list of shapes by applying a translation:

```
Shape* move_shape(Shape* s, Point p)    // *s += p
{
    s->move_to(s->center()+p);
    return s;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
    // invoke operation on corresponding object:
    transform(ls.begin(), ls.end(), oper.begin(), ls.begin(), move_shape);
}
```

I didn't really want to produce a return value from *move_shape*( ). However, *transform*( ) insists on assigning the result of its operation, so I let *move_shape*( ) return its first operand so that I could write it back to where it came from.

Sometimes, we do not have the freedom to do that. For example, an operation that I didn't write and don't want to modify might not return a value. Sometimes, the input sequence is *const*. In such cases, we might define a two-sequence *for_each*( ) to match the two-sequence *transform*( ):

```
template<class In, class In2, class BinOp>
BinOp for_each(In first, In last, In2 first2, BinOp op)
{
    while (first != last) op(*first++, *first2++);
    return op;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
    for_each(ls.begin(), ls.end(), oper.begin(), move_shape);
}
```

At other times, it can be useful to have an output iterator that doesn't actually write anything (§19.6[2]).

There are no standard library algorithms that read three or more sequences. Such algorithms are easily written, though. Alternatively, you can use *transform*( ) repeatedly.

18.6.3 Unique [algo.unique]

Whenever information is collected, duplication can occur. The *unique*( ) and *unique_copy*( ) algorithms eliminate adjacent duplicate values:

```
template<class For> For unique(For first, For last);
template<class For, class BinPred> For unique(For first, For last, BinPred p);

template<class In, class Out> Out unique_copy(In first, In last, Out res);
template<class In, class Out, class BinPred>
    Out unique_copy(In first, In last, Out res, BinPred p);
```

The ***unique*( )** algorithm eliminates adjacent duplicates from a sequence, ***unique_copy*( )** makes a copy without duplicates. For example:

```
void f(list<string>& ls, vector<string>& vs)
{
    ls.sort(); // list sort (§17.2.2.1)
    unique_copy(ls.begin(), ls.end(), back_inserter(vs));
}
```

This copies ***ls*** to ***vs***, eliminating duplicates in the process. The ***sort*( )** is needed to get equal strings adjacent.

Like other standard algorithms, ***unique*( )** operates on iterators. It has no way of knowing the type of container these iterators point into, so it cannot modify that container. It can only modify the values of the elements. This implies that ***unique*( )** does not eliminate duplicates from its input sequence in the way we naively might expect. Rather, it moves unique elements towards the front (head) of a sequence and returns an iterator to the end of the subsequence of unique elements:

```
template <class For> For unique(For first, For last)
{
    first = adjacent_find(first, last);        // §18.5.2
    return unique_copy(first, last, first);
}
```

The elements after the unique subsequence are left unchanged. Therefore, this does not eliminate duplicates in a vector:

```
void f(vector<string>& vs)        // warning: bad code!
{
    sort(vs.begin(), vs.end());          // sort vector
    unique(vs.begin(), vs.end());        // eliminate duplicates (no it doesn't!)
}
```

In fact, by moving the last elements of a sequence forward to eliminate duplicates, ***unique*( )** can introduce new duplicates. For example:

```
int main()
{
    char v[] = "abbcccde";

    char* p = unique(v, v+strlen(v));
    cout << v << ´ ´ << p-v << ´\n´;
}
```

produced

```
abcdecde 5
```

That is, ***p*** points to the second ***c***.

Algorithms that might have removed elements (but can't) generally come in two forms: the ''plain'' version that reorders elements in a way similar to ***unique*( )** and a version that produces a new sequence in a way similar to ***unique_copy*( )**. The ***_copy*** suffix is used to distinguish these two kinds of algorithms.

To eliminate duplicates from a container, we must explicitly shrink it:

```
template<class C> void eliminate_duplicates(C& c)
{
    sort(c.begin(),c.end());                                  // sort
    typename C::iterator p = unique(c.begin(),c.end());       // compact
    c.erase(p,c.end());                                       // shrink
}
```

Note that ***eliminate_duplicates*()** would make no sense for a built-in array, yet ***unique*()** can still be applied to arrays.

An example of ***unique_copy*()** can be found in §3.8.3.

18.6.3.1 Sorting Criteria [algo.criteria]

To eliminate all duplicates, the input sequences must be sorted (§18.7.1). Both ***unique*()** and ***unique_copy*()** use == as the default criterion for comparison and allow the user to supply alternative criteria. For instance, we might modify the example from §18.5.1 to eliminate duplicate names. After extracting the names of the ***Club*** officers, we were left with a ***list<Person*>*** called ***off*** (§18.5.1). We could eliminate duplicates like this:

```
eliminate_duplicates(off);
```

However, this relies on sorting pointers and assumes that each pointer uniquely identifies a person. In general, we would have to examine the ***Person*** records to determine whether we would consider them equal. We might write:

```
bool operator==(const Person& x, const Person& y) // equality for object
{
    // compare x and y for equality
}
bool operator<(const Person& x, const Person& y)  // less than for object
{
    // compare x and y for order
}
bool Person_eq(const Person* x, const Person* y)  // equality through pointer
{
    return *x == *y;
}
bool Person_lt(const Person* x, const Person* y)  // less than through pointer
{
    return *x < *y;
}
```

```
void extract_and_print(const list<Club>& lc)
{
    list<Person*> off;
    extract(lc, off);
    sort(off.begin(), off.end(), Person_lt);
    list<Club>::iterator p = unique(off.begin(), off.end(), Person_eq);
    for_each(off.begin(), p, Print_name(cout));
}
```

It is wise to make sure that the criterion used to sort matches the one used to eliminate duplicates. The default meanings of < and == for pointers are rarely useful as comparison criteria for the objects pointed to.

18.6.4 Replace [algo.replace]

The ***replace*( )** algorithms traverse a sequence, replacing values by other values as specified. They follow the patterns outlined by ***find/find_if*** and ***unique/unique_copy***, thus yielding four variants in all. Again, the code is simple enough to be illustrative:

```
template<class For, class T>
void replace(For first, For last, const T& val, const T& new_val)
{
    while (first != last) {
        if (*first == val) *first = new_val;
        ++first;
    }
}

template<class For, class Pred, class T>
void replace_if(For first, For last, Pred p, const T& new_val)
{
    while (first != last) {
        if (p(*first)) *first = new_val;
        ++first;
    }
}

template<class In, class Out, class T>
Out replace_copy(In first, In last, Out res, const T& val, const T& new_val)
{
    while (first != last) {
        *res++ = (*first == val) ? new_val : *first;
        ++first;
    }
    return res;
}
```

```
template<class In, class Out, class Pred, class T>
Out replace_copy_if(In first, In last, Out res, Pred p, const T& new_val)
{
    while (first != last) {
        *res++ = p(*first) ? new_val : *first;
        ++first;
    }
    return res;
}
```

We might want to go through a list of ***string***s, replacing the usual English transliteration of the name of my home town Aarhus with its proper name Århus:

```
void f(list<string>& towns)
{
    replace(towns.begin(), towns.end(), "Aarhus", "Århus");
}
```

This relies on an extended character set (§C.3.3).

18.6.5 Remove [algo.remove]

The ***remove*( )** algorithms remove elements from a sequence based on a value or a predicate:

```
template<class For, class T> For remove(For first, For last, const T& val);

template<class For, class Pred> For remove_if(For first, For last, Pred p);

template<class In, class Out, class T>
    Out remove_copy(In first, In last, Out res, const T& val);

template<class In, class Out, class Pred>
    Out remove_copy_if(In first, In last, Out res, Pred p);
```

Assuming that a ***Club*** has an address, we could produce a list of ***Club***s located in Copenhagen:

```
class located_in {
    string town;
public:
    located_in(const string& ss) :town(ss) { }
    bool operator()(const Club& c) const { return c.town == town; }
};

void f(list<Club>& lc)
{
    remove_copy_if(lc.begin(), lc.end(),
                   output_iterator<Club>(cout), not1(located_in("København")));
}
```

Thus, ***remove_copy_if*( )** is ***copy_if*( )** (§18.6.1) with the inverse condition. That is, an element is placed on the output by ***remove_copy_if*( )** if the element does not match the predicate.

The ‘‘plain’’ ***remove*( )** compacts non-matching elements at the beginning of the sequence and returns an iterator for the end of the compacted sequence (see also §18.6.3).

18.6.6 Fill and Generate [algo.fill]

The *fill*( ) and *generate*( ) algorithms exist to systematically assign values to sequences:

```
template<class For, class T> void fill(For first, For last, const T& val);
template<class Out, class Size, class T> void fill_n(Out res, Size n, const T& val);

template<class For, class Gen> void generate(For first, For last, Gen g);
template<class Out, class Size, class Gen> void generate_n(Out res, Size n, Gen g);
```

The *fill*( ) algorithm assigns a specified value; the *generate*( ) algorithm assigns values obtained by calling its function argument repeatedly. Thus, *fill*( ) is simply the special case of *generate*( ) in which the generator function returns the same value repeatedly. The *_n* versions assign to the first *n* elements of the sequence.

For example, using the random-number generators *Randint* and *Urand* from §22.7:

```
int v1[900];
int v2[900];
vector v3;

void f()
{
    fill(v1,&v1[900],99);                  // set all elements of v1 to 99
    generate(v2,&v2[900],Randint);         // set to random values (§22.7)

    // output 200 random integers in the interval [0..99]:
    generate_n(ostream_iterator<int>(cout),200,Urand(100));

    fill_n(back_inserter(v3),20,99);       // add 20 elements with the value 99 to v3
}
```

The *generate*( ) and *fill*( ) functions assign rather than initialize. If you need to manipulate raw storage, say to turn a region of memory into objects of well-defined type and state, you must use an algorithm like *uninitialized_fill*( ) from *<memory>* (§19.4.4) rather than algorithms from *<algorithm>*.

18.6.7 Reverse and Rotate [algo.reverse]

Occasionally, we need to reorder the elements of a sequence:

```
template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> Out reverse_copy(Bi first, Bi last, Out res);

template<class For> void rotate(For first, For middle, For last);
template<class For, class Out> Out rotate_copy(For first, For middle, For last, Out res);

template<class Ran> void random_shuffle(Ran first, Ran last);
template<class Ran, class Gen> void random_shuffle(Ran first, Ran last, Gen& g);
```

The *reverse*( ) algorithm reverses the order of the elements so that the first element becomes the last, etc. The *reverse_copy*( ) algorithm produces a copy of its input in reverse order.

The *rotate*( ) algorithm considers its [*first*, *last*[sequence a circle and rotates its elements until its former *middle* element is placed where its *first* element used to be. That is, the element in

position *first+i* moves to position *first+(i+(last-middle))%(last-first)*. The % (modulo) is what makes the rotation cyclic rather than simply a shift to the left. For example:

```
void f()
{
    string v[] = { "Frog", "and", "Peach" };

    reverse(v, v+3);          // Peach and Frog
    rotate(v, v+1, v+3);      // and Frog Peach
}
```

The *rotate_copy()* algorithm produces a copy of its input in rotated order.

By default, *random_shuffle()* shuffles its sequence using a uniform distribution random-number generator. That is, it chooses a permutation of the elements of the sequence in such a way that each permutation has the same chance of being chosen. If you want a different distribution or simply a better random-number generator, you can supply one. For example, using the *Urand* generator from §22.7 we might shuffle a deck of cards like this:

```
void f(deque<Card>& dc)
{
    random_shuffle(dc.begin(), dc.end(), Urand(52));
    // ...
}
```

The movement of elements done by *rotate()*, etc., is done using *swap()* (§18.6.8).

18.6.8 Swap [algo.swap]

To do anything at all interesting with elements in a container, we need to move them around. Such movement is best expressed – that is, expressed most simply and most efficiently – as *swap()*s:

```
template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

template<class For, class For2> void iter_swap(For x, For2 y);

template<class For, class For2> For2 swap_ranges(For first, For last, For2 first2)
{
    while (first != last) iter_swap(first++, first2++);
    return first2;
}
```

To swap elements, you need a temporary. There are clever tricks to eliminate that need in specialized cases, but they are best avoided in favor of the simple and obvious. The *swap()* algorithm is specialized for important types for which it matters (§16.3.9, §13.5.2).

The *iter_swap()* algorithm swaps the elements pointed to by its iterator arguments.

The *swap_ranges* algorithm swaps elements in its two input ranges.

18.7 Sorted Sequences [algo.sorted]

Once we have collected some data, we often want to sort it. Once the sequence is sorted, our options for manipulating the data in a convenient manner increase significantly.

To sort a sequence, we need a way of comparing elements. This is done using a binary predicate (§18.4.2). The default comparison is *less* (§18.4.2), which in turn uses < by default.

18.7.1 Sorting [algo.sort]

The *sort*( ) algorithms require random-access iterators (§19.2.1). That is, they work best for *vector*s (§16.3) and similar containers:

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);

template<class Ran> void stable_sort(Ran first, Ran last);
template<class Ran, class Cmp> void stable_sort(Ran first, Ran last, Cmp cmp);
```

The standard *list* (§17.2.2) does not provide random-access iterators, so *list*s should be sorted using the specific *list* operations (§17.2.2.1).

The basic *sort*( ) is efficient – on average $N*log(N)$ – but its worst-case performance is poor – $O(N*N)$. Fortunately, the worst case is rare. If guaranteed worst-case behavior is important or a stable sort is required, *stable_sort*( ) should be used; that is, an $N*log(N)*log(N)$ algorithm that improves towards $N*log(N)$ when the system has sufficient extra memory. The relative order of elements that compare equal is preserved by *stable_sort*( ) but not by *sort*( ).

Sometimes, only the first elements of a sorted sequence are needed. In that case, it makes sense to sort the sequence only as far as is needed to get the first part in order. That is a partial sort:

```
template<class Ran> void partial_sort(Ran first, Ran middle, Ran last);
template<class Ran, class Cmp>
    void partial_sort(Ran first, Ran middle, Ran last, Cmp cmp);

template<class In, class Ran>
    Ran partial_sort_copy(In first, In last, Ran first2, Ran last2);
template<class In, class Ran, class Cmp>
    Ran partial_sort_copy(In first, In last, Ran first2, Ran last2, Cmp cmp);
```

The plain *partial_sort*( ) algorithms put the elements in the range *first* to *middle* in order. The *partial_sort_copy*( ) algorithms produce *N* elements, where *N* is the lower of the number of elements in the output sequence and the number of elements in the input sequence. We need to specify both the start and the end of the result sequence because that's what determines how many elements we need to sort. For example:

```
class Compare_copies_sold {
public:
    int operator()(const Book& b1, const Book& b2) const
        { return b1.copies_sold()<b2.copies_sold(); }
};
```

```
void f(const vector<Book>& sales)    // find the top ten books
{
    vector<Book> bestsellers(10);
    partial_sort_copy(sales.begin(), sales.end(),
                      bestsellers.begin(), bestsellers.end(), Compare_copies_sold());
    copy(bestsellers.begin(), bestsellers.end(), ostream_iterator<Book>(cout));
}
```

Because the target of ***partial_sort_copy*( )** must be a random-access iterator, we cannot sort directly to ***cout***.

Finally, algorithms are provided to sort only as far as is necessary to get the ***N***th element to its proper place with no element comparing less than the ***Nth*** element placed after it in the sequence:

```
template<class Ran> void nth_element(Ran first, Ran nth, Ran last);
template<class Ran, class Cmp> void nth_element(Ran first, Ran nth, Ran last, Cmp cmp);
```

This algorithm is particularly useful for people – such as economists, sociologists, and teachers – who need to look for medians, percentiles, etc.

18.7.2 Binary Search [algo.bsearch]

A sequential search such as ***find*( )** (§18.5.2) is terribly inefficient for large sequences, but it is about the best we can do without sorting or hashing (§17.6). Once a sequence is sorted, however, we can use a binary search to determine whether a value is in a sequence:

```
template<class For, class T> bool binary_search(For first, For last, const T& val);

template<class For, class T, class Cmp>
    bool binary_search(For first, For last, const T& value, Cmp cmp);
```

For example:

```
void f(list<int>& c)
{
    if (binary_search(c.begin(), c.end(), 7)) {    // is 7 in c?
        // ...
    }
    // ...
}
```

A ***binary_search*( )** returns a ***bool*** indicating whether a value was present. As with ***find*( )**, we often also want to know where the elements with that value are in that sequence. However, there can be many elements with a given value in a sequence, and we often need to find either the first or all such elements. Consequently, algorithms are provided for finding a range of equal elements, ***equal_range*( )**, and algorithms for finding the ***lower_bound*( )** and ***upper_bound*( )** of that range:

```
template<class For, class T> For lower_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
    For lower_bound(For first, For last, const T& val, Cmp cmp);
```

```
template<class For, class T> For upper_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
    For upper_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> pair<For, For> equal_range(For first, For last, const T& val);
template<class For, class T, class Cmp>
    pair<For, For> equal_range(For first, For last, const T& val, Cmp cmp);
```

These algorithms correspond to the operations on *multimap*s (§17.4.2). We can think of *lower_bound*( ) as a fast *find*( ) and *find_if*( ) for sorted sequences. For example:

```
void g(vector<int>& c)
{
    typedef vector<int>::iterator VI;

    VI p = find(c.begin(), c.end(), 7);          // probably slow: O(N); c needn't be sorted
    VI q = lower_bound(c.begin(), c.end(), 7);   // probably fast: O(log(N)); c must be sorted
    // ...
}
```

If *lower_bound*(*first*, *last*, *k*) doesn't find *k*, it returns an iterator to the first element with a key greater than *k*, or *last* if no such greater element exists. This way of reporting failure is also used by *upper_bound*( ) and *equal_range*( ). This means that we can use these algorithms to determine where to insert a new element into a sorted sequence so that the sequence remains sorted.

18.7.3 Merge [algo.merge]

Given two sorted sequences, we can merge them into a new sorted sequence using *merge*( ) or merge two parts of a sequence using *inplace_merge*( ):

```
template<class In, class In2, class Out>
    Out merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class Bi> void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Cmp> void inplace_merge(Bi first, Bi middle, Bi last, Cmp cmp);
```

Note that these merge algorithms differ from *list*'s merge (§17.2.2.1) by ***not*** removing elements from their input sequences. Instead, elements are copied.

For elements that compare equal, elements from the first range will always precede elements from the second.

The *inplace_merge*( ) algorithm is primarily useful when you have a sequence that can be sorted by more than one criterion. For example, you might have a *vector* of fish sorted by species (for example, cod, haddock, and herring). If the elements of each species are sorted by weight, you can get the whole vector sorted by weight by applying *inplace_merge*( ) to merge the information for the different species (§18.13[20]).

18.7.4 Partitions [algo.partition]

To partition a sequence is to place every element that satisfies a predicate before every element that doesn't. The standard library provides a *stable_partition*( ), which maintains relative order among the elements that do and do not satisfy the predicate. In addition, the library offers *partition*( ) which doesn't maintain relative order, but which runs a bit faster when memory is limited:

```
template<class Bi, class Pred> Bi partition(Bi first, Bi last, Pred p);
template<class Bi, class Pred> Bi stable_partition(Bi first, Bi last, Pred p);
```

You can think of a partition as a kind of sort with a very simple sorting criterion. For example:

```
void f(list<Club>& lc)
{
    list<Club>::iterator p = partition(lc.begin(), lc.end(), located_in("København"));
    // ...
}
```

This ''sorts'' the *list* so that *Club*s in Copenhagen comes first. The return value (here *p*) points either to the first element that doesn't satisfy the predicate or to the end.

18.7.5 Set Operations on Sequences [algo.set]

A sequence can be considered a set. Looked upon that way, it makes sense to provide set operations such as union and intersection for sequences. However, such operations are horribly inefficient unless the sequences are sorted, so the standard library provides set operations for sorted sequences only. In particular, the set operations work well for *set*s (§17.4.3) and *multiset*s (§17.4.4), both of which are sorted anyway.

If these set algorithms are applied to sequences that are not sorted, the resulting sequences will not conform to the usual set-theoretical rules. These algorithms do not change their input sequences, and their output sequences are ordered.

The *includes*( ) algorithm tests whether every member of the first sequence is also a member of the second:

```
template<class In, class In2>
    bool includes(In first, In last, In2 first2, In2 last2);
template<class In, class In2, class Cmp>
    bool includes(In first, In last, In2 first2, In2 last2, Cmp cmp);
```

The *set_union*( ) and *set_intersection*( ) produce their obvious outputs as sorted sequences:

```
template<class In, class In2, class Out>
    Out set_union(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_union(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_intersection(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_intersection(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

The *set_difference*( ) algorithm produces a sequence of elements that are members of its first, but

not its second, input sequence. The *set_symmetric_difference*( ) algorithm produces a sequence of elements that are members of either, but not of both, of its input sequences:

```
template<class In, class In2, class Out>
    Out set_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

For example:

```
char v1[] = "abcd";
char v2[] = "cdef";

void f(char v3[])
{
    set_difference(v1,v1+4,v2,v2+4,v3);                  // v3 = "ab"
    set_symmetric_difference(v1,v1+4,v2,v2+4,v3);        // v3 = "abef"
}
```

18.8 Heaps [algo.heap]

The word *heap* means different things in different contexts. When discussing algorithms, ‘‘heap’’ often refers to a way of organizing a sequence such that it has a first element that is the element with the highest value. Addition of an element (using *push_heap*( )) and removal of an element (using *pop_heap*( )) are reasonably fast, with a worst-case performance of $O(log(N))$, where N is the number of elements in the sequence. Sorting (using *sort_heap*( )) has a worst-case performance of $O(N*log(N))$. A heap is implemented by this set of functions:

```
template<class Ran> void push_heap(Ran first, Ran last);
template<class Ran, class Cmp> void push_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void pop_heap(Ran first, Ran last);
template<class Ran, class Cmp> void pop_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void make_heap(Ran first, Ran last);      // turn sequence into heap
template<class Ran, class Cmp> void make_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void sort_heap(Ran first, Ran last);      // turn heap into sequence
template<class Ran, class Cmp> void sort_heap(Ran first, Ran last, Cmp cmp);
```

The style of the heap algorithms is odd. A more natural way of presenting their functionality would be to provide an adapter class with four operations. Doing that would yield something like a *priority_queue* (§17.3.3). In fact, a *priority_queue* is almost certainly implemented using a heap.

The value pushed by *push_heap*(*first*, *last*) is *(*last-1*). The assumption is that [*first*, *last-1*[is already a heap, so *push_heap*( ) extends the sequence to [*first*, *last*[by including the next element. Thus, you can build a heap from an existing sequence by a series of

push_heap( ) operations. Conversely, *pop_heap*(*first*, *last*) removes the first element of the heap by swapping it with the last element (*(*last*-1)) and making [*first*, *last*-1[into a heap.

18.9 Min and Max [algo.min]

The algorithms described here select a value based on a comparison. It is obviously useful to be able to find the maximum and minimum of two values:

```
template<class T> const T& max(const T& a, const T& b)
{
    return (a<b) ? b : a;
}

template<class T, class Cmp> const T& max(const T& a, const T& b, Cmp cmp)
{
    return (cmp(a,b)) ? b : a;
}

template<class T> const T& min(const T& a, const T& b);

template<class T, class Cmp> const T& min(const T& a, const T& b, Cmp cmp);
```

The *max*( ) and *min*( ) operations can be generalized to apply to sequences in the obvious manner:

```
template<class For> For max_element(For first, For last);
template<class For, class Cmp> For max_element(For first, For last, Cmp cmp);

template<class For> For min_element(For first, For last);
template<class For, class Cmp> For min_element(For first, For last, Cmp cmp);
```

Finally, lexicographical ordering is easily generalized from strings of characters to sequences of values of a type with comparison:

```
template<class In, class In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2, Cmp cmp)
{
    while (first != last && first2 != last2) {
        if (cmp(*first, *first2)) return true;
        if (cmp(*first2++, *first++)) return false;
    }
    return first == last && first2 != last2;
}
```

This is very similar to the function presented for general strings in (§13.4.1). However, *lexicographical_compare*( ) compares sequences in general and not just strings. It also returns a *bool* rather than the more useful *int*. The result is *true* (only) if the first sequence compares < the second. In particular, the result is *false* when the sequences compare equal.

C-style strings and *string*s are sequences, so *lexicographical_compare*( ) can be used as a string compare function. For example:

```
char v1[] = "yes";
char v2[] = "no";
string s1 = "Yes";
string s2 = "No";

void f()
{
    bool b1 = lexicographical_compare(v1, v1+strlen(v1), v2, v2+strlen(v2));
    bool b2 = lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end());

    bool b3 = lexicographical_compare(v1, v1+strlen(v1), s1.begin(), s1.end());
    bool b4 = lexicographical_compare(s1.begin(), s1.end(), v1, v1+strlen(v1), Nocase);
}
```

The sequences need not be of the same type – all we need is to compare their elements – and the comparison criterion can be supplied. This makes ***lexicographical_compare*()** more general and potentially a bit slower than ***string***'s compare. See also §20.3.8.

18.10 Permutations [algo.perm]

Given a sequence of four elements, we can order them in 4*3*2 ways. Each of these orderings is called a *permutation*. For example, from the four characters ***abcd*** we can produce 24 permutations:

```
abcd abdc acbd acdb adbc adcb bacd badc
bcad bcda bdac bdca cabd cadb cbad cbda
cdab cdba dabc dacb dbac dbca dcab dcba
```

The ***next_permutation*()** and ***prev_permutation*()** functions deliver such permutations of a sequence:

```
template<class Bi> bool next_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool next_permutation(Bi first, Bi last, Cmp cmp);

template<class Bi> bool prev_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool prev_permutation(Bi first, Bi last, Cmp cmp);
```

The permutations of ***abcd*** were produced like this:

```
int main()
{
    char v[] = "abcd";
    cout << v << '\t';
    while(next_permutation(v, v+4)) cout << v << '\t';
}
```

The permutations are produced in lexicographical order (§18.9). The return value of ***next_permutation*()** indicates whether a next permutation actually exists. If not, ***false*** is returned and the sequence is the permutation in which the elements are in lexicographical order.

18.11 C-Style Algorithms [algo.c]

From the C standard library, the C++ standard library inherited a few algorithms dealing with C-style strings (§20.4.1), plus a quicksort and a binary search, both limited to arrays.

The *qsort*( ) and *bsearch*( ) functions are presented in <*cstdlib*> and <*stdlib.h*>. They each operate on an array of *n* elements of size *elem_size* using a less-than comparison function passed as a pointer to function. The elements must be of a type without a user-defined copy constructor, copy assignment, or destructor:

```
typedef int(*__cmp)(const void*, const void*);      // typedef for presentation only

void qsort(void* p, size_t n, size_t elem_size, __cmp);                    // sort p
void* bsearch(const void* key, void* p, size_t n, size_t elem_size, __cmp); // find key in p
```

The use of *qsort*( ) is described in §7.7.

These algorithms are provided solely for C compatibility; *sort*( ) (§18.7.1) and *search*( ) (§18.5.5) are more general and should also be more efficient.

18.12 Advice [algo.advice]

[1] Prefer algorithms to loops; §18.5.1.
[2] When writing a loop, consider whether it could be expressed as a general algorithm; §18.2.
[3] Regularly review the set of algorithms to see if a new application has become obvious; §18.2.
[4] Be sure that a pair of iterator arguments really do specify a sequence; §18.3.1.
[5] Design so that the most frequently-used operations are simple and safe; §18.3, §18.3.1.
[6] Express tests in a form that allows them to be used as predicates; §18.4.2.
[7] Remember that predicates are functions and objects, not types; §18.4.2.
[8] You can use binders to make unary predicates out of binary predicates; §18.4.4.1.
[9] Use *mem_fun*( ) and *mem_fun_ref*( ) to apply algorithms on containers; §18.4.4.2.
[10] Use *ptr_fun*( ) when you need to bind an argument of a function; §18.4.4.3.
[11] Remember that *strcmp*( ) differs from == by returning *0* to indicate ‘‘equal;’’ §18.4.4.4.
[12] Use *for_each*( ) and *transform*( ) only when there is no more-specific algorithm for a task; §18.5.1.
[13] Use predicates to apply algorithms using a variety of comparison and equality criteria; §18.4.2.1, §18.6.3.1.
[14] Use predicates and other function objects so as to use standard algorithms with a wider range of meanings; §18.4.2.
[15] The default == and < on pointers are rarely adequate for standard algorithms; §18.6.3.1.
[16] Algorithms do not directly add or subtract elements from their argument sequences; §18.6.
[17] Be sure that the less-than and equality predicates used on a sequence match; §18.6.3.1.
[18] Sometimes, sorted sequences can be used to increase efficiency and elegance; §18.7.
[19] Use *qsort*( ) and *bsearch*( ) for compatibility only; §18.11.

18.13 Exercises [algo.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

1. (∗2) Learn *O*( ) notation. Give a realistic example in which an *O*(*N*∗*N*) algorithm is faster than an *O*(*N*) algorithm for some *N>10*.
2. (∗2) Implement and test the four *mem_fun*( ) and *mem_fun_ref*( ) functions (§18.4.4.2).
3. (∗1) Write an algorithm *match*( ) that is like *mismatch*( ), except that it returns iterators to the first corresponding pair that matches the predicate.
4. (∗1.5) Implement and test *Print_name* from §18.5.1.
5. (∗1) Sort a *list* using only standard library algorithms.
6. (∗2.5) Define versions of *iseq*( ) (§18.3.1) for built-in arrays, *istream*, and iterator pairs. Define a suitable set of overloads for the nonmodifying standard algorithms (§18.5) for *Iseq*s. Discuss how best to avoid ambiguities and an explosion in the number of template functions.
7. (∗2) Define an *oseq*( ) to complement *iseq*( ). The output sequence given as the argument to *oseq*( ) should be replaced by the output produced by an algorithm using it. Define a suitable set of overloads for at least three standard algorithms of your choice.
8. (∗1.5) Produce a *vector* of squares of numbers 1 through 100. Print a table of squares. Take the square root of the elements of that *vector* and print the resulting vector.
9. (∗2) Write a set of functional objects that do bitwise logical operations on their operands. Test these objects on vectors of *char*, *int*, and *bitset<67>*.
10. (∗1) Write a *binder3*( ) that binds the second and third arguments of a three-argument function to produce a unary predicate. Give an example where *binder3*( ) is a useful function.
11. (∗1.5) Write a small program that that removes adjacent repeated words from from a file file. Hint: The program should remove a *that*, a *from*, and a *file* from the previous statement.
12. (∗2.5) Define a format for records of references to papers and books kept in a file. Write a program that can write out records from the file identified by year of publication, name of author, keyword in title, or name of publisher. The user should be able to request that the output be sorted according to similar criteria.
13. (∗2) Implement a *move*( ) algorithm in the style of *copy*( ) in such a way that the input and output sequences can overlap. Be reasonably efficient when given random-access iterators as arguments.
14. (∗1.5) Produce all anagrams of the word *food*. That is, all four-letter combinations of the letters *f*, *o*, *o*, and *d*. Generalize this program to take a word as input and produce anagrams of that word.
15. (∗1.5) Write a program that produces anagrams of sentences; that is, a program that produces all permutations of the words in the sentences (rather than permutations of the letters in the words).
16. (∗1.5) Implement *find_if*( ) (§18.5.2) and then implement *find*( ) using *find_if*( ). Find a way of doing this so that the two functions do not need different names.
17. (∗2) Implement *search*( ) (§18.5.5). Provide an optimized version for random-access iterators.
18. (∗2) Take a sort algorithm (such as *sort*( ) from your standard library or the Shell sort from §13.5.2) and insert code so that it prints out the sequence being sorted after each swap of elements.

19. (*2) There is no *sort*( ) for bidirectional iterators. The conjecture is that copying to a vector and then sorting is faster than sorting a sequence using bidirectional iterators. Implement a general sort for bidirectional iterators and test the conjecture.
20. (*2.5) Imagine that you keep records for a group of sports fishermen. For each catch, keep a record of species, length, weight, date of catch, name of fisherman, etc. Sort and print the records according to a variety of criteria. Hint: *inplace_merge*( ).
21. (*2) Create lists of students taking Math, English, French, and Biology. Pick about 20 names for each class out of a set of 40 names. List students who take both Math and English. List students who take French but not Biology or Math. List students who do not take a science course. List students who take French and Math but neither English nor Biology.
22. (*1.5) Write a *remove*( ) function that actually removes elements from a container.

19

Iterators and Allocators

The reason that data structures and algorithms
can work together seamlessly is ... that they
do not know anything about each other.
– Alex Stepanov

Iterators and sequences — operations on iterators — iterator traits — iterator categories — inserters — reverse iterators — stream iterators — checked iterators — exceptions and algorithms — allocators — the standard ***allocator*** — user-defined allocators — low-level memory functions — advice — exercises.

19.1 Introduction [iter.intro]

Iterators are the glue that holds containers and algorithms together. They provide an abstract view of data so that the writer of an algorithm need not be concerned with concrete details of a myriad of data structures. Conversely, the standard model of data access provided by iterators relieves containers from having to provide a more extensive set of access operations. Similarly, allocators are used to insulate container implementations from details of access to memory.

Iterators support an abstract model of data as sequences of objects (§19.2). Allocators provide a mapping from a lower-level model of data as arrays of bytes into the higher-level object model (§19.4). The most common lower-level memory model is itself supported by a few standard functions (§19.4.4).

Iterators are a concept with which every programmer should be familiar. In contrast, allocators are a support mechanism that a programmer rarely needs to worry about and few programmers will ever need to write a new allocator.

19.2 Iterators and Sequences [iter.iter]

An iterator is a pure abstraction. That is, anything that behaves like an iterator is an iterator (§3.8.2). An iterator is an abstraction of the notion of a pointer to an element of a sequence. Its key concepts are

- ‘‘the element currently pointed to’’ (dereferencing, represented by operators `*` and `->`),
- ‘‘point to next element’’ (increment, represented by operator `++`), and
- equality (represented by operator `==`).

For example, the built-in type ***int**** is an iterator for an ***int*`[]` and the class ***list<int>::iterator*** is an iterator for a ***list*** class.

A sequence is an abstraction of the notion ‘‘something where we can get from the beginning to the end by using a next-element operation:’’

Examples of such sequences are arrays (§5.2), vectors (§16.3), singly-linked lists (§17.8[17]), doubly-linked lists (§17.2.2), trees (§17.4.1), input (§21.3.1), and output (§21.2.1). Each has its own appropriate kind of iterator.

The iterator classes and functions are declared in namespace ***std*** and found in ***<iterator>***.

An iterator is *not* a general pointer. Rather, it is an abstraction of the notion of a pointer into an array. There is no concept of a ‘‘null iterator.’’ The test to determine whether an iterator points to an element or not is conventionally done by comparing it against the *end* of its sequence (rather than comparing it against a ***null*** element). This notion simplifies many algorithms by removing the need for a special end case and generalizes nicely to sequences of arbitrary types.

An iterator that points to an element is said to be *valid* and can be dereferenced (using `*`, `[]`, or `->` appropriately). An iterator can be invalid either because it hasn’t been initialized, because it pointed into a container that was explicitly or implicitly resized (§16.3.6, §16.3.8), because the container into which it pointed was destroyed, or because it denotes the end of a sequence (§18.2). The end of a sequence can be thought of as an iterator pointing to a hypothetical element position one-past-the-last element of a sequence.

19.2.1 Iterator Operations [iter.oper]

Not every kind of iterator supports exactly the same set of operations. For example, reading requires different operations from writing, and a ***vector*** allows convenient and efficient random access in a way that would be prohibitively expensive to provide for a ***list*** or an ***istream***. Consequently, we classify iterators into five categories according to the operations they are capable of providing efficiently (that is, in constant time; §17.1):

Iterator Operations and Categories					
Category:	output	input	forward	bidirectional	random-access
Abbreviation:	***Out***	***In***	***For***	***Bi***	***Ran***
Read:		`=*p`	`=*p`	`=*p`	`=*p`
Access:		`->`	`->`	`->`	`-> []`
Write:	`*p=`		`*p=`	`*p=`	`*p=`
Iteration:	`++`	`++`	`++`	`++ --`	`++ -- + - += -=`
Comparison:		`== !=`	`== !=`	`== !=`	`== != < > >= <=`

Both read and write are through the iterator dereferenced by `*`:

```
*p = x;     // write x through p
x = *p;     // read through p into x
```

To be an iterator type, a type must provide an appropriate set of operations. These operations must have their conventional meanings. That is, each operation must have the same effect it has on an ordinary pointer.

Independently of its category, an iterator can allow ***const*** or non-***const*** access to the object it points to. You cannot write to an element using an iterator to ***const*** – whatever its category. An iterator provides a set of operators, but the type of the element pointed to is the final arbiter of what can be done to that element.

Reads and writes copy objects, so element types must have the conventional copy semantics (§17.1.4).

Only random-access iterators can have an integer added or subtracted for relative addressing. However, except for output iterators, the distance between two iterators can always be found by iterating through the elements, so a ***distance***`()` function is provided:

```
template<class In> typename iterator_traits<In>::difference_type distance(In first, In last)
{
    typename iterator_traits<In>::difference_type d = 0;
    while (first++!=last) d++;
    return d;
}
```

An ***iterator_traits<In>::difference_type*** is defined for every iterator ***In*** to hold distances between elements (§19.2.2).

This function is called ***distance***`()` rather than ***operator-***`()` because it can be expensive and the operators provided for an iterator all operate in constant time (§17.1). Counting elements one by one is not the kind of operation I would like to invoke unwittingly for a large sequence. The library also provides a far more efficient implementation of ***distance***`()` for a random-access iterator.

Similarly, ***advance***`()` is provided as a potentially slow `+=`:

```
template <class In, class Dist> void advance(In i, Dist n);     // i+=n
```

19.2.2 Iterator Traits [iter.traits]

We use iterators to gain information about the objects they point to and the sequences they point into. For example, we can dereference an iterator and manipulate the resulting object and we can find the number of elements in a sequence, given the iterators that describe it. To express such operations, we must be able to refer to types related to an iterator such as ''the type of the object referred to by an iterator'' and ''the type of the distance between two iterators.'' The related types of an iterator are described by a small set of declarations in an ***iterator_traits*** template class:

```
template<class Iter> struct iterator_traits {
    typedef typename Iter::iterator_category iterator_category;     // §19.2.3
    typedef typename Iter::value_type value_type;                   // type of element
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::pointer pointer;                         // return type of operator->()
    typedef typename Iter::reference reference;                     // return type of operator*()
};
```

The ***difference_type*** is the type used to represent the difference between two iterators, and the ***iterator_category*** is a type indicating what operations the iterator supports. For ordinary pointers, specializations (§13.5) for ***<T*>*** and ***<const T*>*** are provided. In particular:

```
template<class T> struct iterator_traits<T*> {          // specialization for pointers
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

That is, the difference between two pointers is represented by the standard library type ***ptrdiff_t*** from ***<cstddef>*** (§6.2.1) and a pointer provides random access (§19.2.3). Given ***iterator_traits***, we can write code that depends on properties of an iterator parameter. The ***count*( )** algorithm is the classical example:

```
template<class In, class T>
typename iterator_traits<In>::difference_type count(In first, In last, const T& val)
{
    typename iterator_traits<In>::difference_type res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

Here, the type of the result is expressed in terms of the ***iterator_traits*** of the input. This technique is necessary because there is no language primitive for expressing an arbitrary type in terms of another.

Instead of using ***iterator_traits***, we might have specialized ***count*( )** for pointers:

```
template<class In, class T>
typename In::difference_type count(In first, In last, const T& val);

template<class In, class T> ptrdiff_t count<T*,T>(T* first, T* last, const T& val);
```

However, this would have solved the problem for ***count*( )** only. Had we used this technique for a dozen algorithms, the information about distance types would have been replicated a dozen times. In general, it is better to represent a design decision in one place (§23.4.2). In that way, the decision can – if necessary – be changed in one place.

Because ***iterator_traits<Iterator>*** is defined for every iterator, we implicitly define an ***iterator_traits*** whenever we design a new iterator type. If the default traits generated from the general ***iterator_traits*** template are not right for our new iterator type, we provide a specialization in a way similar to what the standard library does for pointer types. The ***iterator_traits*** that are implicitly generated assume that the iterator is a class with the member types ***difference_type***, ***value_type***, etc. In ***<iterator>***, the library provides a base type that can be used to define those member types:

```
template<class Cat, class T, class Dist = ptrdiff_t, class Ptr = T*, class Ref = T&>
struct iterator {
    typedef Cat iterator_category;   // §19.2.3
    typedef T value_type;            // type of element
    typedef Dist difference_type;    // type of iterator difference
    typedef Ptr pointer;             // return type for ->
    typedef Ref reference;           // return type for *
};
```

Note that ***reference*** and ***pointer*** are not iterators. They are intended to be the return types of ***operator** ( )** and ***operator*->( )**, respectively, for some iterator.

The ***iterator_traits*** are the key to the simplicity of many interfaces that rely on iterators and to the efficient implementation of many algorithms.

19.2.3 Iterator Categories [iter.cat]

The different kinds of iterators – usually referred to as iterator categories – fit into a hierarchical ordering:

This is not a class inheritance diagram. An iterator category is a classification of a type based on the operations it provides. Many otherwise unrelated types can belong to the same iterator category. For example, both ordinary pointers (§19.2.2) and ***Checked_iter***s (§19.3) are random-access iterators.

As noted in Chapter 18, different algorithms require different kinds of iterators as arguments. Also, the same algorithm can sometimes be implemented with different efficiencies for different kinds of iterators. To support overload resolution based on iterator categories, the standard library provides five classes representing the five iterator categories:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

Looking at the operations supported by input and forward iterators (§19.2.1), we would expect ***forward_iterator_tag*** to be derived from ***output_iterator_tag*** as well as from ***input_iterator_tag***. The reasons that it is not are obscure and probably invalid. However, I have yet to see an example in which that derivation would have simplified real code.

The inheritance of tags is useful (only) to save us from defining separate versions of a function where several – but not all – kinds of iterators can use the same algorithms. Consider how to implement ***distance***:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last);
```

There are two obvious alternatives:

[1] If ***In*** is a random-access iterator, we can subtract ***first*** from ***last***.

[2] Otherwise, we must increment an iterator from ***first*** to ***last*** and count the distance.

We can express these two alternatives as a pair of helper functions:

```
template<class In>
typename iterator_traits<In>::difference_type
dist_helper(In first, In last, input_iterator_tag)
{
    typename iterator_traits<In>::difference_type d = 0;
    while (first++!=last) d++;                    // use increment only
    return d;
}

template<class Ran>
typename iterator_traits<Ran>::difference_type
dist_helper(Ran first, Ran last, random_access_iterator_tag)
{
    return last-first;     // rely on random access
}
```

The iterator category tag arguments make it explicit what kind of iterator is expected. The iterator tag is used exclusively for overload resolution; the tag takes no part in the actual computation. It is a purely compile-time selection mechanism. In addition to automatic selection of a helper function, this technique provides immediate type checking (§13.2.5).

It is now trivial to define ***distance*()** by calling the appropriate helper function:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last)
{
    return dist_helper(first,last,iterator_traits<In>::iterator_category());
}
```

For a *dist_helper*( ) to be called, the *iterator_traits<In>::iterator_category* used must be a *input_iterator_tag* or a *random_access_iterator_tag*. However, there is no need for separate versions of *dist_helper*( ) for forward or bidirectional iterators. Thanks to tag inheritance, those cases are handled by the *dist_helper*( ) which takes an *input_iterator_tag*. The absence of a version for *output_iterator_tag* reflects the fact that *distance*( ) is not meaningful for output iterators:

```
void f(vector<int>& vi,
       list<double>& ld,
       istream_iterator<string>& is1, istream_iterator<string>& is2,
       ostream_iterator<char>& os1, ostream_iterator<char>& os2)
{
       distance(vi.begin(),vi.end());          // use subtraction algorithm
       distance(ld.begin(),ld.end());          // use increment algorithm
       distance(is1,is2);                      // use increment algorithm
       distance(os1,os2);   // error: wrong iterator category, dist_helper() argument type mismatch
}
```

Calling *distance*( ) for an *istream_iterator* probably doesn't make much sense in a real program, though. The effect would be to read the input, throw it away, and return the number of values thrown away.

Using *iterator_traits<T>::iterator_category* allows a programmer to provide alternative implementations so that a user who cares nothing about the implementation of algorithms automatically gets the most appropriate implementation for each data structure used. In other words, it allows us to hide an implementation detail behind a convenient interface. Inlining can be used to ensure that this elegance is not bought at the cost of run-time efficiency.

19.2.4 Inserters [iter.insert]

Producing output through an iterator into a container implies that elements following the one pointed to by the iterator can be overwritten. This implies the possibility of overflow and consequent memory corruption. For example:

```
void f(vector<int>& vi)
{
       fill_n(vi.begin(),200,7);        // assign 7 to vi[0]..[199]
}
```

If *vi* has fewer than *200* elements, we are in trouble.

In *<iterator>*, the standard library provides three iterator template classes to deal with this problem, plus three functions to make it convenient to use those iterators:

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);
template <class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);
template <class Cont, class Out> insert_iterator<Cont> inserter(Cont& c, Out p);
```

The *back_inserter*( ) causes elements to be added to the end of the container, *front_inserter*( ) causes elements to be added to the front, and ''plain'' *inserter*( ) causes elements to be added before its iterator argument. For *inserter*(*c*,*p*), *p* must be a valid iterator for *c*. Naturally, a container grows each time a value is written to it through an insert iterator.

When written to, an inserter inserts a new element into a sequence using ***push_back*( )**, ***push_front*( )**, or ***insert*( )** (§16.3.6) rather than overwriting an existing element. For example:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi),200,7);    // add 200 7s to the end of vi
}
```

Inserters are as simple and efficient as they are useful. For example:

```
template <class Cont>
class insert_iterator : public iterator<output_iterator_tag,void,void,void,void> {
protected:
    Cont& container;                    // container to insert into
    typename Cont::iterator iter;       // points into the container
public:
    explicit insert_iterator(Cont& x, typename Cont::iterator i)
        : container(x), iter(i) {}

    insert_iterator& operator=(const typename Cont::value_type& val)
    {
        iter = container.insert(iter,val);
        ++iter;
        return *this;
    }

    insert_iterator& operator*( ) { return *this; }
    insert_iterator& operator++( ) { return *this; }        // prefix ++
    insert_iterator operator++(int) { return *this; }       // postfix ++
};
```

Clearly, inserters are output iterators.

An ***insert_iterator*** is a special case of an output sequence. In parallel to the ***iseq*** from §18.3.1, we might define:

```
template<class Cont>
insert_iterator<Cont>
oseq(Cont& c, typename Cont::iterator first, typename Cont::iterator last)
{
    return insert_iterator<Cont>(c,c.erase(first,last)); // erase is explained in §16.3.6
}
```

In other words, an output sequence removes its old elements and replaces them with the output. For example:

```
void f(list<int>& li,vector<int>& vi)  // replace second half of vi by a copy of li
{
    copy(li.begin( ),li.end( ),oseq(vi,vi+vi.size( )/2,vi.end( )));
}
```

The container needs to be an argument to an ***oseq*** because it is not possible to decrease the size of a container, given only iterators into it (§18.6, §18.6.3).

19.2.5 Reverse Iterators [iter.reverse]

The standard containers provide ***rbegin*()** and ***rend*()** for iterating through elements in reverse order (§16.3.2). These member functions return ***reverse_iterator***s:

```
template <class Iter>
class reverse_iterator : public iterator<iterator_traits<Iter>::iterator_category,
                                         iterator_traits<Iter>::value_type,
                                         iterator_traits<Iter>::difference_type,
                                         iterator_traits<Iter>::pointer,
                                         iterator_traits<Iter>::reference> {
protected:
    Iter current;    // current points to the element after the one *this refers to.
public:
    typedef Iter iterator_type;

    reverse_iterator() : current() { }
    explicit reverse_iterator(Iter x) : current(x) { }
    template<class U> reverse_iterator(const reverse_iterator<U>& x) : current(x.base()) { }

    Iter base() const { return current; } // current iterator value

    reference operator*() const { Iter tmp = current; return *--tmp; }
    pointer operator->() const;
    reference operator[](difference_type n) const;

    reverse_iterator& operator++() { --current; return *this; }      // note: not ++
    reverse_iterator operator++(int) { reverse_iterator t = current; --current;  return t; }
    reverse_iterator& operator--() { ++current; return *this; }      // note: not --
    reverse_iterator operator--(int) { reverse_iterator t = current; ++current;  return t; }

    reverse_iterator operator+(difference_type n) const;
    reverse_iterator& operator+=(difference_type n);
    reverse_iterator operator-(difference_type n) const;
    reverse_iterator& operator-=(difference_type n);
};
```

A ***reverse_iterator*** is implemented using an ***iterator*** called ***current***. That ***iterator*** can (only) point to the elements of its sequence plus its one-past-the-end element. However, the ***reverse_iterator***'s one-past-the-end element is the original sequence's (inaccessible) one-before-the-beginning element. Thus, to avoid access violations, ***current*** points to the element after the one the ***reverse_iterator*** refers to. This implies that `*` returns the value `*(current-1)` and that `++` is implemented using `--` on ***current***.

A ***reverse_iterator*** supports the operations that its initializer supports (only). For example:

```
void f(vector<int>& v, list<char>& lst)
{
    reverse_iterator(v.end())[3] = 7;          // ok: random-access iterator
    reverse_iterator(lst.end())[3] = '4';      // error: bidirectional iterator doesn't support []
    *(++++++reverse_iterator(lst.end())) = '4'; // ok!
}
```

In addition, the library provides `==`, `!=`, `<`, `<=`, `>`, `>=`, `+` and `-` for ***reverse_iterator***s.

19.2.6 Stream Iterators [iter.stream]

Ordinarily, I/O is done using the streams library (Chapter 21), a graphical user-interface system (not covered by the C++ standard), or the C I/O functions (§21.8). These I/O interfaces are primarily aimed at reading and writing individual values of a variety of types. The standard library provides four iterator types to fit stream I/O into the general framework of containers and algorithms:

- ***ostream_iterator***: for writing to an ***ostream*** (§3.4, §21.2.1).
- ***istream_iterator***: for reading from an ***istream*** (§3.6, §21.3.1).
- ***ostreambuf_iterator***: for writing to a stream buffer (§21.6.1).
- ***istreambuf_iterator***: for reading from a stream buffer (§21.6.2).

The idea is simply to present input and output of collections as sequences:

```
template <class T, class Ch = char, class Tr = char_traits<Ch> >
class ostream_iterator : public iterator<output_iterator_tag, void, void, void, void> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef basic_ostream<Ch, Tr> ostream_type;

    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const Ch* delim); // write delim after each output value
    ostream_iterator(const ostream_iterator&);
    ~ostream_iterator();

    ostream_iterator& operator=(const T& val);          // write val to output

    ostream_iterator& operator*();
    ostream_iterator& operator++();
    ostream_iterator& operator++(int);
};
```

This iterator accepts the usual write and increment operations of an output iterator and converts them into output operations on an ***ostream***. For example:

```
void f()
{
    ostream_iterator<int> os(cout);     // write ints to cout through os
    *os = 7;        // output 7
    ++os;           // get ready for next output
    *os = 79;       // output 79
}
```

The ++ operation might trigger an actual output operation, or it might have no effect. Different implementations will use different implementation strategies. Consequently, for code to be portable a ++ must occur between every two assignments to an ***ostream_iterator***. Naturally, every standard algorithm is written that way – or it would not work for a ***vector***. This is why ***ostream_iterator*** is defined this way.

An implementation of ***ostream_iterator*** is trivial and is left as an exercise (§19.6[4]). The standard I/O supports different character types; ***char_traits*** (§20.2) describes the aspects of a character type that can be important for I/O and ***string***s.

An input iterator for ***istream***s is defined analogously:

```
template <class T, class Ch = char, class Tr = char_traits<Ch>, class Dist = ptrdiff_t>
class istream_iterator : public iterator<input_iterator_tag, T, Dist, const T*, const T&> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef basic_istream<Ch,Tr> istream_type;

    istream_iterator();                          // end of input
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator&);
    ~istream_iterator();

    const T& operator*() const;
    const T* operator->() const;
    istream_iterator& operator++();
    istream_iterator operator++(int);
};
```

This iterator is specified so that what would be conventional use for a container triggers input from an ***istream***. For example:

```
void f()
{
    istream_iterator<int> is(cin);          // read ints from cin through is
    int i1 = *is;           // read an int
    ++is;                   // get ready for next input
    int i2 = *is;           // read an int
}
```

The default ***istream_iterator*** represents the end of input so that we can specify an input sequence:

```
void f(vector<int>& v)
{
    copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
}
```

To make this work, the standard library supplies `==` and `!=` for ***istream_iterator***s.

An implementation of ***istream_iterator*** is less trivial than an ***ostream_iterator*** implementation, but it is still simple. Implementing an ***istream_iterator*** is also left as an exercise (§19.6[5]).

19.2.6.1 Stream Buffers [iter.streambuf]

As described in §21.6, stream I/O is based on the idea of ***ostream***s and ***istream***s filling and emptying buffers from and to which the low-level physical I/O is done. It is possible to bypass the standard iostreams formatting and operate directly on the stream buffers (§21.6.4). That ability is also provided to algorithms through the notion of ***istreambuf_iterator***s and ***ostreambuf_iterator***s:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator
        : public iterator<input_iterator_tag, Ch, typename Tr::off_type, Ch*, Ch&> {
public:
        typedef Ch char_type;
        typedef Tr traits_type;
        typedef typename Tr::int_type int_type;
        typedef basic_streambuf<Ch, Tr> streambuf_type;
        typedef basic_istream<Ch, Tr> istream_type;

        class proxy;                                    // helper type

        istreambuf_iterator() throw();                  // end of buffer
        istreambuf_iterator(istream_type& is) throw();  // read from is's streambuf
        istreambuf_iterator(streambuf_type*) throw();
        istreambuf_iterator(const proxy& p) throw();    // read from p's streambuf

        Ch operator*() const;
        istreambuf_iterator& operator++();              // prefix
        proxy operator++(int);                          // postfix

        bool equal(istreambuf_iterator&);               // both or neither streambuf at eof
};
```

In addition, == and != are supplied.

Reading from a ***streambuf*** is a lower-level operation than reading from an ***istream***. Consequently, the ***istreambuf_iterator*** interface is messier than the ***istream_iterator*** interface. However, once the ***istreambuf_iterator*** is properly initialized, *, ++, and = have their usual meanings when used in the usual way.

The ***proxy*** type is an implementation-defined helper type that allows the postfix ++ to be implemented without imposing constraints on the ***streambuf*** implementation. A ***proxy*** holds the result value while the iterator is incremented:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator<Ch, Tr>::proxy {
        Ch val;
        basic_istreambuf<Ch, Tr>* buf;

        proxy(Ch v, basic_istreambuf<Ch, Tr>* b) :val(v), buf(b) { }
public:
        Ch operator*() { return val; }
};
```

An ***ostreambuf_iterator*** is defined similarly:

```
template <class Ch, class Tr = char_traits<Ch> >
class ostreambuf_iterator : public iterator<output_iterator_tag, void, void, void, void>{
public:
        typedef Ch char_type;
        typedef Tr traits_type;
```

```
    typedef basic_streambuf<Ch, Tr> streambuf_type;
    typedef basic_ostream<Ch, Tr> ostream_type;

    ostreambuf_iterator(ostream_type& os) throw();        // write to os's streambuf
    ostreambuf_iterator(streambuf_type*) throw();
    ostreambuf_iterator& operator=(Ch);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);

    bool failed() const throw();                          // true if Tr::eof() seen
};
```

19.3 Checked Iterators [iter.checked]

A programmer can provide iterators in addition to those provided by the standard library. This is often necessary when providing a new kind of container, and sometimes a new kind of iterator is a good way to support a different way of using existing containers. As an example, I here describe an iterator that range checks access to its container.

Using standard containers reduces the amount of explicit memory management. Using standard algorithms reduces the amount of explicit addressing of elements in containers. Using the standard library together with language facilities that maintain type safety dramatically reduces run-time errors compared to traditional C coding styles. However, the standard library still relies on the programmer to avoid access beyond the limits of a container. If by accident element `x[x.size()+7]` of some container `x` is accessed, then unpredictable – and usually bad – things happen. Using a range-checked *vector*, such as *Vec* (§3.7.1), helps in some cases. More cases can be handled by checking every access through an iterator.

To achieve this degree of checking without placing a serious notational burden on the programmer, we need checked iterators and a convenient way of attaching them to containers. To make a *Checked_iter*, we need a container and an iterator into that container. As for binders (§18.4.4.1), inserters (§19.2.4), etc., I provide functions for making a *Checked_iter*:

```
template<class Cont, class Iter> Checked_iter<Cont, Iter> make_checked(Cont& c, Iter i)
{
    return Checked_iter<Cont, Iter>(c, i);
}

template<class Cont> Checked_iter<Cont, typename Cont::iterator> make_checked(Cont& c)
{
    return Checked_iter<Cont, typename Cont::iterator>(c, c.begin());
}
```

These functions offer the notational convenience of deducing the types from arguments rather than stating those types explicitly. For example:

```
void f(vector<int>& v, const vector<int>& vc)
{
    typedef Checked_iter<vector<int>,vector<int>::iterator> CI;
    CI p1 = make_checked(v,v.begin()+3);
    CI p2 = make_checked(v);            // by default: point to first element

    typedef Checked_iter<const vector<int>,vector<int>::const_iterator> CIC;
    CIC p3 = make_checked(vc,vc.begin()+3);
    CIC p4 = make_checked(vc);

    const vector<int>& vv = v;
    CIC p5 = make_checked(v,vv.begin());
}
```

By default, ***const*** containers have ***const*** iterators, so their ***Checked_iter***s must also be constant iterators. The iterator ***p5*** shows one way of getting a ***const*** iterator for a non-***const*** iterator.

This demonstrates why ***Checked_iter*** needs two template parameters: one for the container type and one to express the ***const***/non-***const*** distinction.

The names of these ***Checked_iter*** types become fairly long and unwieldy, but that doesn't matter when iterators are used as arguments to a generic algorithm. For example:

```
template<class Iter> void mysort(Iter first, Iter last);

void f(vector<int>& c)
{
    try {
        mysort(make_checked(c), make_checked(c,c.end()));
    }
    catch (out_of_bounds) {
        cerr<<"oops: bug in mysort()\n";
        abort();
    }
}
```

An early version of such an algorithm is exactly where I would most suspect a range error so that using checked iterators would make sense.

The representation of a ***Checked_iter*** is a pointer to a container plus an iterator pointing into that container:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    Iter curr; // iterator for current position
    Cont* c;   // pointer to current container

    // ...
};
```

Deriving from ***iterator_traits*** is one technique for defining the desired ***typedef***s. The obvious alternative – deriving from ***iterator*** – would be verbose in this case (as it was for ***reverse_iterator***; §19.2.5). Just as there is no requirement that an iterator should be a class, there is no requirement that iterators that are classes should be derived from ***iterator***.

The *Checked_iter* operations are all fairly trivial:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    // ...
public:
    void valid(Iter p)
    {
        if (c->end() == p) return;
        for (Iter pp = c->begin(); pp!=c->end(); ++pp) if (pp == p) return;
        throw out_of_bounds()
    }

    friend bool operator==(const Checked_iter& i, const Checked_iter& j)
    {
        return i.c==j.c && i.curr==j.curr;
    }

    // no default initializer.

    // use default copy constructor and copy assignment.

    Checked_iter(Cont& x, Iter p) : c(&x), curr(p) { valid(p); }

    reference_type operator*()
    {
        if (curr==c->end()) throw out_of_bounds();
        return *curr;
    }

    pointer_type operator->()
    {
        return &*curr;                          // checked by *
    }

    Checked_iter operator+(Dist d)              // for random-access iterators only
    {
        if (c->end()-curr<=d) throw out_of_bounds();
        return Checked_iter(c,curr+d);
    }

    reference_type operator[](Dist d)           // for random-access iterators only
    {
        if (c->end()-curr<=d) throw out_of_bounds();
        return c[d];
    }

    Checked_iter& operator++()                  // prefix ++
    {
        if (curr == c->end()) throw out_of_bounds();
        ++curr;
        return *this;
    }
```

```
        Checked_iter  operator++(int)                    // postfix ++
        {
            Checked_iter  tmp = *this;
            ++*this;                                     // checked by prefix ++
            return  tmp;
        }

        Checked_iter& operator--()                       // prefix --
        {
            if (curr == c->begin()) throw  out_of_bounds();
            --curr;
            return *this;
        }

        Checked_iter  operator--(int)                    // postfix --
        {
            Checked_iter  tmp = *this;
            --*this;                                     // checked by prefix --
            return  tmp;
        }

        difference_type  index() { return  curr-c.begin(); }  // random-access only

        Iter  unchecked() { return  curr; }

        // +, -, < , etc. (§19.6[6])
    };
```

A *Checked_iter* can be initialized only for a particular iterator pointing into a particular container. In a full-blown implementation, a more efficient version of *valid*( ) should be provided for random-access iterators (§19.6[6]). Once a *Checked_iter* is initialized, every operation that changes its position is checked to make sure the iterator still points into the container. An attempt to make the iterator point outside the container causes an *out_of_bounds* exception to be thrown. For example:

```
    void  f(list<string>& ls)
    {
        int  count = 0;
        try {
            Checked_iter< list<string> > p(ls, ls.begin());
            while (true) {
                ++p;        // sooner or later this will reach the end
                ++count;
            }
        }
        catch(out_of_bounds) {
            cout << "overrun  after " << count << " tries\n";
        }
    }
```

A *Checked_iter* knows which container it is pointing into. This allows it to catch some, but not all, cases in which iterators into a container have been invalidated by an operation on it (§16.3.8). To

protect against all such cases, a different and more expensive iterator design would be needed (see §19.6[7]).

Note that postincrement (postfix ++) involves a temporary and preincrement does not. For this reason, it is best to prefer ++*p* over *p*++ for iterators.

Because a ***Checked_iter*** keeps a pointer to a container, it cannot be used for a built-in array directly. When necessary, a ***c_array*** (§17.5.4) can be used.

To complete the notion of checked iterators, we must make them trivial to use. There are two basic approaches:

[1] Define a checked container type that behaves like other containers, except that it provides only a limited set of constructors and its ***begin***( ), ***end***( ), etc., supply ***Checked_iter***s rather than ordinary iterators.

[2] Define a handle that can be initialized by an arbitrary container and that provides checked access functions to its container (§19.6[8]).

The following template attaches checked iterators to a container:

```
template<class C> class Checked : public C {
public:
    explicit Checked(size_t n) :C(n) { }
    Checked( ) :C( ) { }

    typedef Checked_iter<C> iterator;
    typedef Checked_iter<C, C::const_iterator> const_iterator;

    typename C::iterator begin( ) { return iterator(*this, C::begin( )); }
    typename C::iterator end( ) { return iterator(*this, C::end( )); }

    typename C::const_iterator begin( ) const { return const_iterator(*this, C::begin( )); }
    typename C::const_iterator end( ) const { return const_iterator(*this, C::end( )); }

    typename C::reference_type operator[](size_t n) { return Checked_iter<C>(*this)[n]; }

    C& base( ) { return static_cast<C&>(*this); }   // get hold of the base container
};
```

This allows us to write:

```
Checked< vector<int> > vec(10);
Checked< list<double> > lst;

void f( )
{
    int v1 = vec[5];            // ok
    int v2 = vec[15];           // throws out_of_bounds
    // ...
    lst.push_back(v2);
    mysort(vec.begin( ), vec.end( ));
    copy(vec.begin( ), vec.end( ), lst.begin( ), lst.end( ));
}
```

If a container is resized, iterators – including ***Checked_iter***s – into it may become invalid. In that case, the ***Checked_iter*** can be re-initialized:

```
void g()
{
    Checked_iter<int> p(vi);
    // ..
    int i = p.index();                          // get current position
    vi.resize(100);                             // p becomes invalid
    p = Checked_iter<int>(vi, vi.begin()+i);    // restore current position
}
```

The old – and invalid – current position is lost. I provided ***index*()** as a means of storing and restoring a ***Checked_iter***. If necessary, a reference to the container used as the base of the ***Checked*** container can be extracted using ***base*()**.

19.3.1 Exceptions, Containers, and Algorithms [iter.except]

You could argue that using both standard algorithms and checked iterators is like wearing both belt and suspenders: either should keep you safe. However, experience shows that for many people and for many applications a dose of paranoia is reasonable – especially during times when a program goes through frequent changes that involve several people.

One way of using run-time checks is to keep them in the code only while debugging. The checks are then removed before the program is shipped. This practice has been compared to wearing a life jacket while paddling around close to the shore and then removing it before setting out onto the open sea. However, some uses of run-time checks do impose significant time and space overheads, so insisting on such checks at all times is not realistic. In any case, it is unwise to optimize without measurements, so before removing checks, do an experiment to see if worthwhile improvements actually emerge from doing so. To do such an experiment, we must be able to remove run-time checks easily (see §24.3.7.1). Once measurements have been done, we could remove the run-time testing from the most run-time critical – and hopefully most thoroughly tested – code and leave the rest of the code checked as a relatively cheap form of insurance.

Using a ***Checked_iter*** allows us to detect many mistakes. It does not, however, make it easy to recover from these errors. People rarely write code that is 100% robust against every `++`, `--`, `*`, `[]`, `->`, and `=` potentially throwing an exception. This leaves us with two obvious strategies:

[1] Catch exceptions close to the point from which they are thrown so that the writer of the exception handler has a decent chance of knowing what went wrong and can take appropriate action.

[2] Catch the exception at a high level of a program, abandon a significant portion of a computation, and consider all data structures written to during the failed computation suspect (maybe there are no such data structures or maybe they can be sanity checked).

It is irresponsible to catch an exception from some unknown part of a program and proceed under the assumption that no data structure is left in an undesirable state, unless there is a further level of error handling that will catch subsequent errors. A simple example of this is when a final check (by computer or human) is done before the results are accepted. In such cases, it can be simpler and cheaper to proceed blithely rather than to try to catch every error at a low level. This would be an example of a simplification made possible by a multilevel error recovery scheme (§14.9).

19.4 Allocators [iter.alloc]

An *allocator* is used to insulate implementers of algorithms and containers that must allocate memory from the details of physical memory. An allocator provides standard ways of allocating and deallocating memory and standard names of types used as pointers and references. Like an iterator, an allocator is a pure abstraction. Any type that behaves like an allocator is an allocator.

The standard library provides a standard allocator intended to serve most users of a given implementation well. In addition, users can provide allocators that represent alternative views of memory. For example, we can write allocators that use shared memory, garbage-collected memory, memory from preallocated pools of objects (§19.4.2), etc.

The standard containers and algorithms obtain and access memory through the facilities provided by an allocator. Thus, by providing a new allocator we provide the standard containers with a way of using a new and different kind of memory.

19.4.1 The Standard Allocator [iter.alloc.std]

The standard *allocator* template from *<memory>* allocates memory using *operator new*( ) (§6.2.6) and is by default used by all standard containers:

```
template <class T> class allocator {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef T* pointer;
    typedef const T* const_pointer;

    typedef T& reference;
    typedef const T& const_reference;

    pointer address(reference r) const { return &r; }
    const_pointer address(const_reference r) const { return &r; }

    allocator() throw();
    template <class U> allocator(const allocator<U>&) throw();
    ~allocator() throw();

    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0); // space for n Ts
    void deallocate(pointer p, size_type n);   // deallocate n Ts, don't destroy

    void construct(pointer p, const T& val) { new(p) T(val); }     // initialize *p by val
    void destroy(pointer p) { p->~T(); }            // destroy *p but don't deallocate

    size_type max_size() const throw();

    template <class U>
    struct rebind { typedef allocator<U> other; };   // in effect: typedef allocator<U> other
};

template<class T> bool operator==(const allocator<T>&, const allocator<T>&) throw();
template<class T> bool operator!=(const allocator<T>&, const allocator<T>&) throw();
```

An *allocate*(*n*) operation allocates space for *n* objects that can be deallocated by a corresponding call of *deallocate*(*p*,*n*). Note that *deallocate*() also takes a number-of-elements argument *n*. This allows for close-to-optimal allocators that maintain only minimal information about allocated memory. On the other hand, such allocators require that the user always provide the right *n* when they *deallocate*().

The default *allocator* uses *operator new*(*size_t*) to obtain memory and *operator delete*(*void**) to free it. This implies that the *new_handler*() might be called and *out_of_memory* might be thrown in case of memory exhaustion (§6.2.6.2).

Note that *allocate*() is not obliged to call a lower-level allocator each time. Often, a better strategy is for the allocator to maintain a free list of space ready to hand out with minimal time overhead (§19.4.2).

The optional *hint* argument to *allocate*() is completely implementation-dependent. However, it is intended as a help to allocators for systems where locality is important. For example, an allocator might try to allocate space for related objects on the same page in a paging system. The type of the *hint* argument is the *pointer* from the ultra-simplified specialization:

```
template <> class allocator<void> {
public:
    typedef void* pointer;
    typedef const void* const_pointer;
    // note: no reference
    typedef void value_type;
    template <class U>
    struct rebind { typedef allocator<U> other; }; // in effect: typedef allocator<U> other
};
```

The *allocator<void>::pointer* type acts as a universal pointer type and is *const void** for all standard allocators.

Unless the documentation for an allocator says otherwise, the user has two reasonable choices when calling *allocate*():

[1] Don't give a hint.

[2] Use a pointer to an object that is frequently used together with the new object as the hint; for example, the previous element in a sequence.

Allocators are intended to save implementers of containers from having to deal with raw memory directly. As an example, consider how a *vector* implementation might use memory:

```
template <class T, class A = allocator<T> > class vector {
public:
    typedef typename A::pointer iterator;
    // ...
private:
    A alloc;          // allocator object
    iterator v;       // pointer to elements
    // ...
```

```
public:
    explicit vector(size_type n, const T& val = T(), const A& a = A())
        : alloc(a)
    {
        v = alloc.allocate(n);
        for(iterator p = v; p<v+n; ++p) alloc.construct(p, val);
        // ...
    }

    void reserve(size_type n)
    {
        if (n<=capacity()) return;

        iterator p = alloc.allocate(n);
        iterator q = v;

        while (q<v+size()) {                    // copy existing elements
            alloc.construct(p++, *q);
            alloc.destroy(q++);
        }
        alloc.deallocate(v, capacity());        // free old space
        v = p;
        // ...
    }

    // ...
};
```

The ***allocator*** operations are expressed in terms of ***pointer*** and ***reference typedef***s to give the user a chance to supply alternative types for accessing memory. This is very hard to do in general. For example, it is not possible to define a perfect reference type within the C++ language. However, language and library implementers can use these ***typedef***s to support types that couldn't be provided by an ordinary user. An example would be an allocator that provided access to a persistent store. Another example would be a ''long'' pointer type for accessing main memory beyond what a default pointer (usually 32 bits) could address.

The ordinary user can supply an unusual pointer type to an allocator for specific uses. The equivalent cannot be done for references, but that may be an acceptable constraint for an experiment or a specialized system.

An allocator is designed to make it easy to handle objects of the type specified by its template parameter. However, most container implementations require objects of additional types. For example, the implementer of a ***list*** will need to allocate ***Link*** objects. Usually, such ***Link***s must be allocated using their ***list***'s allocator.

The curious ***rebind*** type is provided to allow an allocator to allocate objects of arbitrary type. Consider:

```
typedef typename A::rebind<Link>::other Link_alloc;
```

If ***A*** is an ***allocator***, then ***rebind<Link>::other*** is ***typedef***'d to mean ***allocator<Link>***, so the previous ***typedef*** is an indirect way of saying:

```
typedef allocator<Link> Link_alloc;
```

The indirection frees us from having to mention ***allocator*** directly. It expresses the ***Link_alloc*** type in terms of a template parameter ***A***. For example:

```
template <class T, class A = allocator<T> > class list {
private:
    class Link { /* ... */ };

    typedef typename A::rebind<Link>::other Link_alloc;    // allocator<Link>

    Link_alloc a;   // link allocator
    A alloc;        // list allocator
    // ...
public:
    typedef typename A::pointer iterator;
    // ...

    iterator insert(iterator pos, const T& x )
    {
        Link_alloc::pointer p = a.allocate(1);    // get a Link
        // ...
    }
    // ...
};
```

Because ***Link*** is a member of ***list***, it is parameterized by an allocator. Consequently, ***Link***s from ***list***s with different allocators are of different types, just like the ***list***s themselves (§17.3.3).

19.4.2 A User-Defined Allocator [iter.alloc.user]

Implementers of containers often ***allocate***( ) and ***deallocate***( ) objects one at a time. For a naive implementation of ***allocate***( ), this implies lots of calls of operator ***new***, and not all implementations of operator ***new*** are efficient when used like that. As an example of a user-defined allocator, I present a scheme for using pools of fixed-sized pieces of memory from which the allocator can ***allocate***( ) more efficiently than can a conventional and more general ***operator new***( ).

I happen to have a pool allocator that does approximately the right thing, but it has the wrong interface (because it was designed years before allocators were invented). This ***Pool*** class implements the notion of a pool of fixed-sized elements from which a user can do fast allocations and deallocations. It is a low-level type that deals with memory directly and worries about alignment:

```
class Pool {
    struct Link { Link* next; };

    struct Chunk {
        enum { size = 8*1024-16 };
        Chunk* next;
        char mem[size];
    };
    Chunk* chunks;

    const unsigned int esize;
    Link* head;
```

```
        Pool(Pool&);                    // copy protection
        void operator=(Pool&);          // copy protection
        void grow();                    // make pool larger
public:
        Pool(unsigned int n);           // n is the size of elements
        ~Pool();

        void* alloc();                  // allocate one element
        void free(void* b);             // put an element back into the pool
};

inline void* Pool::alloc()
{
        if (head==0) grow();
        Link* p = head;                 // return first element
        head = p->next;
        return p;
}

inline void Pool::free(void* b)
{
        Link* p = static_cast<Link*>(b);
        p->next = head;                 // put b back as first element
        head = p;
}

Pool::Pool(unsigned int sz)
        : esize(sz<sizeof(Link*)?sizeof(Link*):sz)
{
        head = 0;
        chunks = 0;
}

Pool::~Pool() // free all chunks
{
        Chunk* n = chunks;
        while (n) {
                Chunk* p = n;
                n = n->next;
                delete p;
        }
}

void Pool::grow()   // allocate new 'chunk,' organize it as a linked list of elements of size 'esize'
{
        Chunk* n = new Chunk;
        n->next = chunks;
        chunks = n;

        const int nelem = Chunk::size/esize;
        char* start = n->mem;
        char* last = &start[(nelem-1)*esize];
```

```
        for (char* p = start; p<last; p+=esize)          // assume sizeof(Link)<=esize
            reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);
        reinterpret_cast<Link*>(last)->next = 0;
        head = reinterpret_cast<Link*>(start);
    }
```

To add a touch of realism, I'll use ***Pool*** unchanged as part of the implementation of my allocator, rather than rewrite it to give it the right interface. The pool allocator is intended for fast allocation and deallocation of single elements and that is what my ***Pool*** class supports. Extending this implementation to handle allocations of arbitrary numbers of objects and to objects of arbitrary size (as required by ***rebind*( )**) is left as an exercise (§19.6[9]).

Given ***Pool***, the definition of ***Pool_alloc*** is trivial;

```
template <class T> class Pool_alloc {
private:
    static Pool mem;    // pool of elements of sizeof(T)
public:
    // like the standard allocator (§19.4.1)
};

template <class T> Pool Pool_alloc<T>::mem(sizeof(T));

template <class T> Pool_alloc<T>::Pool_alloc() { }

template <class T>
T* Pool_alloc<T>::allocate(size_type n, void* = 0)
{
    if (n == 1) return static_cast<T*>(mem.alloc());
    // ...
}

template <class T>
void Pool_alloc<T>::deallocate(pointer p, size_type n)
{
    if (n == 1) {
        mem.free(p);
        return;
    }
    // ...
}
```

This allocator can now be used in the obvious way:

```
vector<int,Pool_alloc> v;
map<string,number,Pool_alloc> m;

// use exactly as usual

vector<int> v2 = v;    // error: different allocator parameters
```

I chose to make the ***Pool*** for a ***Pool_alloc*** static because of a restriction that the standard library imposes on allocators used by the standard containers: the implementation of a standard container is allowed to treat every object of its allocator type as equivalent. This can lead to significant

performance advantages. For example, because of this restriction, memory need not be set aside for allocators in ***Link*** objects (which are typically parameterized by the allocator of the container for which they are ***Link***s; §19.4.1), and operations that may access elements of two sequences (such as ***swap***( )) need not check whether the objects manipulated all have the same allocator. However, the restriction does imply that such allocators cannot use per-object data.

Before applying this kind of optimization, make sure that it is necessary. I expect that many default ***allocator***s will implement exactly this kind of classic C++ optimization – thus saving you the bother.

19.4.3 Generalized Allocators [iter.general]

An ***allocator*** is a simplified and optimized variant of the idea of passing information to a container through a template parameter (§13.4.1, §16.2.3). For example, it makes sense to require that every element in a container is allocated by the container's allocator. However, if two ***list***s of the same type were allowed to have different allocators, then ***splice***( ) (§17.2.2.1) couldn't be implemented through relinking. Instead, ***splice***( ) would have to be defined in terms of copying of elements to protect against the rare cases in which we want to splice elements from a ***list*** with one allocator into another with a different allocator of the same allocator type. Similarly, if allocators were allowed to be perfectly general, the ***rebind*** mechanism that allows an allocator to allocate elements of arbitrary types would have to be more elaborate. Consequently, a standard allocator is assumed to hold no per-object data and an implementation of a standard may take advantage of that.

Surprisingly, the apparently Draconian restriction against per-object information in allocators is not particularly serious. Most allocators do not need per-object data and can be made to run faster without such data. Allocators can still hold data on a per-allocator-type basis. If separate data is needed, separate allocator types can be used. For example:

```
template<class T, class D> class My_alloc {     // allocator for T implemented using D
    D d;        // data needed for My_alloc<T,D>
    // ...
};

typedef My_alloc<int,Persistent_info> Persistent;
typedef My_alloc<int,Shared_info> Shared;
typedef My_alloc<int,Default_info> Default;

list<int,Persistent> lst1;
list<int,Shared> lst2;
list<int,Default> lst3;
```

The lists ***lst1***, ***lst2***, and ***lst3*** are of different types. Therefore, we must use general algorithms (Chapter 18) when operating on two of these lists rather than specialized list operations (§17.2.2.1). This implies that copying rather than relinking is done, so having different allocators poses no problems.

The restriction against per-object data in allocators is imposed because of the stringent demands on the run-time and space efficiency of the standard library. For example, the space overhead of allocator data for a list probably wouldn't be significant. However, it could be serious if each link of a list suffered overhead.

Consider how the allocator technique could be used when the efficiency constraints of the standard library don't apply. This would be the case for a nonstandard library that wasn't meant to deliver high performance for essentially every data structure and every type in a program and for some special-purpose implementations of the standard library. In such cases, an allocator can be used to carry the kind of information that often inhabits universal base classes (§16.2.2). For example, an allocator could be designed to answer requests about where its objects are allocated, present data representing object layout, and answer questions such as ''is this element in this container?'' It could also provide controls for a container that acts as a cache for memory in permanent storage, provide association between the container and other objects, etc.

In this way, arbitrary services can be provided transparently to the ordinary container operations. However, it is best to distinguish between issues relating to storage of data and issues of the use of data. The latter do not belong in a generalized allocator, but they could be provided through a separate template argument.

19.4.4 Uninitialized Memory [iter.memory]

In addition to the standard ***allocator***, the ***<memory>*** header provides a few functions for dealing with uninitialized memory. They share the dangerous and occasionally essential property of using a type name ***T*** to refer to space sufficient to hold an object of type ***T*** rather than to a properly constructed object of type ***T***.

The library provides three ways to copy values into uninitialized space:

```
template <class In, class For>
For uninitialized_copy(In first, In last, For res)     // copy into res
{
    typedef typename iterator_traits<For>::value_type V;

    while (first != last)
        new (static_cast<void*>(&*res++)) V(*first++);     // construct in res (§10.4.11)
    return res;
}

template <class For, class T>
void uninitialized_fill(For first, For last, const T& val)
{
    typedef typename iterator_traits<For>::value_type V;

    while (first != last) new (static_cast<void*>(&*first++)) V(val);     // construct in first
}

template <class For, class Size, class T>
void uninitialized_fill_n(For first, Size n, const T& val)
{
    typedef typename iterator_traits<For>::value_type V;

    while (n--) new (static_cast<void*>(&*first++)) V(val); // construct in first
}
```

These functions are intended primarily for implementers of containers and algorithms. For example, ***reserve()*** and ***resize()*** (§16.3.8) are most easily implemented using these functions

(§19.6[10]). It would clearly be most unfortunate if an uninitialized object escaped from the internals of a container into the hands of general users.

Algorithms often require temporary space to perform acceptably. Often, such temporary space is best allocated in one operation but not initialized until a particular location is actually needed. Consequently, the library provides a pair of functions for allocating and deallocating uninitialized space:

```
template <class T> pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t); // allocate, don't initialize
template <class T> void return_temporary_buffer(T*);                   // deallocate, don't destroy
```

A *get_temporary_buffer<X>(n)* operation tries to allocate space for ***n*** or more objects of type ***X***. If it succeeds in allocating some memory, it returns a pointer to the first uninitialized space and the number of objects of type ***X*** that will fit into that space; otherwise, the ***second*** value of the pair is zero. The idea is that a system may keep a number of fixed-sized buffers ready for fast allocation so that requesting space for ***n*** objects may yield space for more than ***n***. It may also yield less, however, so one way of using *get_temporary_buffer()* is to optimistically ask for a lot and then use what happens to be available.

A buffer obtained by *get_temporary_buffer()* must be freed for other use by a call of *return_temporary_buffer()*. Just as *get_temporary_buffer()* allocates without constructing, *return_temporary_buffer()* frees without destroying. Because *get_temporary_buffer()* is low-level and likely to be optimized for managing temporary buffers, it should not be used as an alternative to ***new*** or *allocator::allocate()* for obtaining longer-term storage.

The standard algorithms that write into a sequence assume that the elements of that sequence have been previously initialized. That is, the algorithms use assignment rather than copy construction for writing. Consequently, we cannot use uninitialized memory as the immediate target of an algorithm. This can be unfortunate because assignment can be significantly more expensive than initialization. Besides, we are not interested in the values we are about to overwrite anyway (or we wouldn't be overwriting them). The solution is to use a ***raw_storage_iterator*** from ***<memory>*** that initializes instead of assigns:

```
template <class Out, class T>
class raw_storage_iterator : public iterator<output_iterator_tag,void,void,void,void> {
    Out p;
public:
    explicit raw_storage_iterator(Out pp) : p(pp) { }

    raw_storage_iterator& operator*() { return *p; }
    raw_storage_iterator& operator=(const T& val)
    {
        T* pp = &*p;
        new(pp) T(val);      // place val in pp (§10.4.11)
        return p;
    }
    raw_storage_iterator& operator++() { return ++p; }
    raw_storage_iterator operator++(int) { return p++; }
};
```

For example, we might write a template that copies the contents of a ***vector*** into a buffer:

```
template<class T, class A> T* temporary_dup(vector<T,A>& v)
{
    T* p = get_temporary_buffer<T>(v.size()).first;
    if (p == 0) return 0;
    copy(v.begin(), v.end(), raw_storage_iterator<T*,T>(p));
    return p;
}
```

Had ***new*** been used instead of ***get_temporary_buffer*()**, initialization would have been done. Once initialization is avoided, the ***raw_storage_iterator*** becomes necessary for dealing with the uninitialized space. In this example, the caller of ***temporary_dup*()** is responsible for calling ***destroy_temporary_buffer*()** for the pointer it received.

19.4.5 Dynamic Memory [iter.dynamic]

The functions used to implement the ***new*** and ***delete*** operators are declared in ***<new>*** together with a few related facilities:

```
class bad_alloc : public exception { /* ... */ };

struct nothrow_t {};
extern const nothrow_t nothrow;      // indicator for allocation that doesn't throw exceptions

typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();

void* operator new(size_t, const nothrow_t&) throw();
void operator delete(void*, const nothrow_t&) throw();

void* operator new[](size_t) throw(bad_alloc);
void operator delete[](void*) throw();

void* operator new[](size_t, const nothrow_t&) throw();
void operator delete[](void*, const nothrow_t&) throw();

void* operator new (size_t, void* p) throw() { return p; }      // placement (§10.4.11)
void operator delete (void* p, void*) throw() { }

void* operator new[](size_t, void* p) throw() { return p; }
void operator delete[](void* p, void*) throw() { }
```

The ***nothrow*** versions of ***operator new*()** allocate as usual, but if allocation fails, they return ***0*** rather than throwing ***bad_alloc***. For example:

```
void f()
{
    int* p = new int[100000]; // may throw bad_alloc
```

```
        if (int* q = new(nothrow) int[100000]) {  // will not throw exception
                // allocation succeeded
        }
        else {
                // allocation failed
        }
}
```

This allows us to use pre-exception error-handling strategies for allocation.

19.4.6 C-Style Allocation [iter.c]

From C, C++ inherited a functional interface to dynamic memory. It can be found in *<cstdlib>*:

```
void* malloc(size_t s);              // allocate s bytes
void* calloc(size_t n, size_t s);    // allocate n times s bytes initialized to 0
void free(void* p);                  // free space allocated by malloc() or calloc()
void* realloc(void* p, size_t s);    // change the size of the array pointed to by p to s;
                                     // if that cannot be done, allocate s bytes, copy
                                     // the array pointed to by p to it, and free p
```

These functions should be avoided in favor of *new*, *delete*, and standard containers. These functions deal with uninitialized memory. In particular, *free*( ) does not invoke destructors for the memory it frees. An implementation of *new* and *delete* may use these functions, but there is no guarantee that it does. For example, allocating an object using *new* and deleting it using *free*( ) is asking for trouble. If you feel the need to use *realloc*( ), consider relying on a standard container instead; doing that is usually simpler and just as efficient (§16.3.5).

The library also provides a set of functions intended for efficient manipulation of bytes. Because C originally accessed untyped bytes through *char** pointers, these functions are found in *<cstring>*. The *void** pointers are treated as if they were *char** pointers within these functions:

```
void* memcpy(void* p, const void* q, size_t n);     // copy non-overlapping areas
void* memmove(void* p, const void* q, size_t n);    // copy potentially overlapping areas
```

Like *strcpy*( ) (§20.4.1), these functions copy *n* bytes from *q* to *p* and return *p*. The ranges copied by *memmove*( ) may overlap. However, *memcpy*( ) assumes that the ranges do not overlap and is usually optimized to take advantage of that assumption. Similarly:

```
void* memchr(const void* p, int b, size_t n);  // like strchr() (§20.4.1): find b in p[0]..p[n-1]
int memcmp(const void* p, const void* q, size_t n); // like strcmp(): compare byte sequences
void* memset(void* p, int b, size_t n);        // set n bytes to b, return p
```

Many implementations provide highly optimized versions of these functions.

19.5 Advice [iter.advice]

[1] When writing an algorithm, decide which kind of iterator is needed to provide acceptable efficiency and express the algorithm using the operators supported by that kind of iterator (only); §19.2.1.

[2] Use overloading to provide more-efficient implementations of an algorithm when given as arguments iterators that offer more than minimal support for the algorithm; §19.2.3.
[3] Use ***iterator_traits*** to express suitable algorithms for different iterator categories; §19.2.2.
[4] Remember to use ++ between accesses of ***istream_iterator***s and ***ostream_iterator***s; §19.2.6.
[5] Use inserters to avoid container overflow; §19.2.4.
[6] Use extra checking during debugging and remove checking later only where necessary; §19.3.1.
[7] Prefer ++***p*** to ***p***++; §19.3.
[8] Use uninitialized memory to improve the performance of algorithms that expand data structures; §19.4.4.
[9] Use temporary buffers to improve the performance of algorithms that require temporary data structures; §19.4.4.
[10] Think twice before writing your own allocator; §19.4.
[11] Avoid ***malloc***( ), ***free***( ), ***realloc***( ), etc.; §19.4.6.
[12] You can simulate a ***typedef*** of a template by the technique used for ***rebind***; §19.4.1.

19.6 Exercises [iter.exercises]

1. (∗1.5) Implement ***reverse***( ) from §18.6.7. Hint: See §19.2.3.
2. (∗1.5) Write an output iterator, ***Sink***, that doesn't actually write anywhere. When can ***Sink*** be useful?
3. (∗2) Implement ***reverse_iterator*** (§19.2.5).
4. (∗1.5) Implement ***ostream_iterator*** (§19.2.6).
5. (∗2) Implement ***istream_iterator*** (§19.2.6).
6. (∗2.5) Complete ***Checked_iter*** (§19.3).
7. (∗2.5) Redesign ***Checked_iter*** to check for invalidated iterators.
8. (∗2) Design and implement a handle class that can act as a proxy for a container by providing a complete container interface to its users. Its implementation should consist of a pointer to a container plus implementations of container operations that do range checking.
9. (∗2.5) Complete or reimplement ***Pool_alloc*** (§19.4.2) so that it provides all of the facilities of the standard library ***allocator*** (§19.4.1). Compare the performance of ***allocator*** and ***Pool_alloc*** to see if there is any reason to use a ***Pool_alloc*** on your system.
10. (∗2.5) Implement ***vector*** using allocators rather than ***new*** and ***delete***.

20

Strings

Prefer the standard to the offbeat.
– Strunk & White

Strings — characters — ***char_traits*** — ***basic_string*** — iterators — element access — constructors — error handling — assignment — conversions — comparisons — insertion — concatenation — find and replace — size and capacity — string I/O — C-style strings — character classification — C library functions — advice — exercises.

20.1 Introduction [string.intro]

A string is a sequence of characters. The standard library ***string*** provides string manipulation operations such as subscripting (§20.3.3), assignment (§20.3.6), comparison (§20.3.8), appending (§20.3.9), concatenation (§20.3.10), and searching for substrings (§20.3.11). No general substring facility is provided by the standard, so one is provided here as an example of standard string use (§20.3.11). A standard string can be a string of essentially any kind of character (§20.2).

Experience shows that it is impossible to design the perfect ***string***. People's taste, expectations, and needs differ too much for that. So, the standard library ***string*** isn't ideal. I would have made some design decisions differently, and so would you. However, it serves many needs well, auxiliary functions to serve further needs are easily provided, and ***std::string*** is generally known and available. In most cases, these factors are more important than any minor improvement we could provide. Writing string classes has great educational value (§11.12, §13.2), but for code meant to be widely used, the standard library ***string*** is the one to use.

From C, C++ inherited the notion of strings as zero-terminated arrays of ***char*** and a set of functions for manipulating such C-style strings (§20.4.1).

20.2 Characters [string.char]

''Character'' is itself an interesting concept. Consider the character *C*. The *C* that you see as a curved line on the page (or screen), I typed into my computer many months ago. There, it lives as the numeric value *67* in an 8-bit byte. It is the third letter in the Latin alphabet, the usual abbreviation for the sixth atom (Carbon), and, incidentally, the name of a programming language (§1.6). What matters in the context of programming with strings is that there is a correspondence between squiggles with conventional meaning, called characters, and numeric values. To complicate matters, the same character can have different numeric values in different character sets, not every character set has values for every character, and many different character sets are in common use. A character set is a mapping between a character (some conventional symbol) and an integer value.

C++ programmers usually assume that the standard American character set (ASCII) is available, but C++ makes allowances for the possibility that some characters may be missing in a programmer's environment. For example, in the absence of characters such as `[` and `{`, keywords and digraphs can be used (§C.3.1).

Character sets with characters not in ASCII offer a greater challenge. Languages such as Chinese, Danish, French, Icelandic, and Japanese cannot be written properly using ASCII only. Worse, the character sets used for these languages can be mutually incompatible. For example, the characters used for European languages using Latin alphabets *almost* fit into a 256-character character set. Unfortunately, different sets are still used for different languages and some different characters have ended up with the same integer value. For example, French (using Latin1) doesn't coexist well with Icelandic (which therefore requires Latin2). Ambitious attempts to present every character known to man in a single character set have helped a lot, but even 16-bit character sets – such as Unicode – are not enough to satisfy everyone. The 32-bit character sets that could – as far as I know – hold every character are not widely used.

Basically, the C++ approach is to allow a programmer to use any character set as the character type in strings. An extended character set or a portable numeric encoding can be used (§C.3.3).

20.2.1 Character Traits [string.traits]

As shown in §13.2, a string can, in principle, use any type with proper copy operations as its character type. However, efficiency can be improved and implementations can be simplified for types that don't have user-defined copy operations. Consequently, the standard ***string*** requires that a type used as its character type does not have user-defined copy operations. This also helps to make I/O of strings simple and efficient.

The properties of a character type are defined by its ***char_traits***. A ***char_traits*** is a specialization of the template:

```
template<class Ch> struct char_traits { };
```

All ***char_traits*** are defined in ***std***, and the standard ones are presented in ***<string>***. The general ***char_traits*** itself has no properties; only ***char_traits*** specializations for a particular character type have. Consider ***char_traits<char>***:

```
template<> struct char_traits<char> {
    typedef char char_type;          // type of character
```

```
    static void assign(char_type&, const char_type&);          // = for char_type

    // integer representation of characters:

    typedef int int_type;              // type of integer value of character

    static char_type to_char_type(const int_type&);            // int to char conversion
    static int_type to_int_type(const char_type&);             // char to int conversion
    static bool eq_int_type(const int_type&, const int_type&); // ==

    // char_type comparisons:

    static bool eq(const char_type&, const char_type&);        // ==
    static bool lt(const char_type&, const char_type&);        // <

    // operations on s[n] arrays:

    static char_type* move(char_type* s, const char_type* s2, size_t n);
    static char_type* copy(char_type* s, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static int compare(const char_type* s, const char_type* s2, size_t n);
    static size_t length(const char_type*);
    static const char_type* find(const char_type* s, int n, const char_type&);

    // I/O related:

    typedef streamoff off_type;        // offset in stream
    typedef streampos pos_type;        // position in stream
    typedef mbstate_t state_type;      // multi-byte stream state

    static int_type eof();                          // end-of-file
    static int_type not_eof(const int_type& i);     // i unless i equals eof(); if not any value!=eof()
    static state_type get_state(pos_type p);        // multibyte conversion state of character in p
};
```

The implementation of the standard string template, ***basic_string*** (§20.3), relies on these types and functions. A type used as a character type for ***basic_string*** must provide a ***char_traits*** specialization that supplies them all.

For a type to be a ***char_type***, it must be possible to obtain an integer value corresponding to each character. The type of that integer is ***int_type***, and the conversion between it and the ***char_type*** is done by ***to_char_type***( ) and ***to_int_type***( ). For a ***char***, this conversion is trivial.

Both ***move***(***s***,***s2***,***n***) and ***copy***(***s***,***s2***,***n***) copy ***n*** characters from ***s2*** to ***s*** using ***assign***(***s***[***i***],***s2***[***i***]). The difference is that ***move***( ) works correctly even if ***s2*** is in the [***s***,***s***+***n***[range. Thus, ***copy***( ) can be faster. This mirrors the standard C library functions ***memcpy***( ) and ***memmove***( ) (§19.4.6). A call ***assign***(***s***,***n***,***x***) assigns ***n*** copies of ***x*** into ***s*** using ***assign***(***s***[***i***],***x***).

The ***compare***( ) function uses ***lt***( ) and ***eq***( ) to compare characters. It returns an ***int***, where ***0*** represents an exact match, a negative number means that its first argument comes lexicographically before the second, and a positive number means that its first argument comes after its second. This mirrors the standard C library function ***strcmp***( ) (§20.4.1).

The I/O-related functions are used by the implementation of low-level I/O (§21.6.4).

A wide character – that is, an object of type ***wchar_t*** (§4.3) – is like a ***char***, except that it takes up two or more bytes. The properties of a ***wchar_t*** are described by ***char_traits<wchar_t>***:

```
template<> struct char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    typedef wstreamoff off_type;
    typedef wstreampos pos_type;

    // like char_traits<char>
};
```

A ***wchar_t*** is typically used to hold characters of a 16-bit character set such as Unicode.

20.3 Basic_string [string.string]

The standard library string facilities are based on the template ***basic_string*** that provides member types and operations similar to those provided by standard containers (§16.3):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class std::basic_string {
public:
    // ...
};
```

This template and its associated facilities are defined in namespace ***std*** and presented by ***<string>***.

Two ***typedef***s provide conventional names for common string types:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

The ***basic_string*** is similar to ***vector*** (§16.3), except that ***basic_string*** provides some typical string operations, such as searching for substrings, instead of the complete set of operations offered by ***vector***. A ***string*** is unlikely to be implemented by a simple array or ***vector***. Many common uses of strings are better served by implementations that minimize copying, use no free store for short strings, allow for simple modification of longer strings, etc. (see §20.6[12]). The number of ***string*** functions reflects the importance of string manipulation and also the fact that some machines provide specialized hardware instructions for string manipulation. Such functions are most easily utilized by a library implementer if there is a standard library function with similar semantics.

Like other standard library types, a ***basic_string<T>*** is a concrete type (§2.5.3, §10.3) without virtual functions. It can be used as a member when designing more sophisticated text manipulation classes, but it is not intended to be a base for derived classes (§25.2.1; see also §20.6[10]).

20.3.1 Types [string.types]

Like ***vector***, ***basic_string*** makes its related types available through a set of member type names:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // types (much like vector, list, etc.: §16.3.1):
```

```
    typedef Tr traits_type;          // specific to basic_string

    typedef typename Tr::char_type value_type;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;

    typedef implementation_defined iterator;
    typedef implementation_defined const_iterator;

    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // ...
};
```

The ***basic_string*** notion supports strings of many kinds of characters in addition to the simple ***basic_string<char>*** known as ***string***. For example:

```
typedef basic_string<unsigned char> Ustring;

struct Jchar { /* ... */ };        // Japanese character type
typedef basic_string<Jchar> Jstring;
```

Strings of such characters can be used just like strings of ***char*** as far as the semantics of the characters allows. For example:

```
Ustring first_word(const Ustring& us)
{
    Ustring::size_type pos = us.find(' ');     // see §20.3.11
    return Ustring(us,0,pos);                  // see §20.3.4
}

Jstring first_word(const Jstring& js)
{
    Jstring::size_type pos = js.find(' ');     // see §20.3.11
    return Jstring(js,0,pos);                  // see §20.3.4
}
```

Naturally, templates that take string arguments can also be used:

```
template<class S> S first_word(const S& s)
{
    typename S::size_type pos = s.find(' '); // see §20.3.11
    return S(s,0,pos);                       // see §20.3.4
}
```

A ***basic_string<Ch>*** can contain any character of the set ***Ch***. In particular, ***string*** can contain a ***0*** (zero).

20.3.2 Iterators [string.begin]

Like other containers, a ***string*** provides iterators for ordinary and reverse iteration:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // iterators (like vector, list, etc.: §16.3.2):

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // ...
};
```

Because ***string*** has the required member types and the functions for obtaining iterators, ***string***s can be used together with the standard algorithms (Chapter 18). For example:

```
void f(string& s)
{
    string::iterator p = find(s.begin(), s.end(), 'a');
    // ...
}
```

The most common operations on ***string***s are supplied directly by ***string***. Hopefully, these versions will be optimized for ***string***s beyond what would be easy to do for general algorithms.

The standard algorithms (Chapter 18) are not as useful for strings as one might think. General algorithms tend to assume that the elements of a container are meaningful in isolation. This is typically not the case for a string. The meaning of a string is encoded in its exact sequence of characters. Thus, sorting a string (that is, sorting the characters in a string) destroys its meaning, whereas sorting a general container typically makes it more useful.

The ***string*** iterators are not range checked.

20.3.3 Element Access [string.elem]

Individual characters of a ***string*** can be accessed through subscripting:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // element access (like vector: §16.3.3):
```

```
        const_reference operator[](size_type n) const; // unchecked access
        reference operator[](size_type n);

        const_reference at(size_type n) const;                // checked access
        reference at(size_type n);

        // ...
    };
```

Out-of-range access causes *at*( ) to throw an *out_of_range*.

Compared to *vector*, *string* lacks *front*( ) and *back*( ). To refer to the first and the last character of a *string*, we must say *s*[*0*] and *s*[*s*.*length*( )-*1*], respectively. The pointer/array equivalence (§5.3) doesn't hold for *strings*. If *s* is a *string*, &*s*[*0*] is not the same as *s*.

20.3.4 Constructors [string.ctor]

The set of initialization and copy operations for a *string* differs from what is provided for other containers (§16.3.4) in many details:

```
    template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
    class basic_string {
    public:
        // ...
        // constructors, etc. (a bit like vector and list: §16.3.4):

        explicit basic_string(const A& a = A());
        basic_string(const basic_string& s,
                     size_type pos = 0, size_type n = npos, const A& a = A());
        basic_string(const Ch* p, size_type n, const A& a = A());
        basic_string(const Ch* p, const A& a = A());
        basic_string(size_type n, Ch c, const A& a = A());
        template<class In> basic_string(In first, In last, const A& a = A());

        ~basic_string();

        static const size_type npos;      // ''all characters'' marker

        // ...
    };
```

A *string* can be initialized by a C-style string, by another *string*, by part of a C-style string, by part of a *string*, or from a sequence of characters. However, a *string* cannot be initialized by a character or an integer:

```
    void f(char* p, vector<char>&v)
    {
        string s0;                  // the empty string
        string s00 = "";            // also the empty string

        string s1 = 'a';            // error: no conversion from char to string
        string s2 = 7;              // error: no conversion from int to string
        string s3(7);               // error: no constructor taking one int argument
```

```
        string s4(7,´a´);          // 7 copies of 'a'; that is "aaaaaaa"

        string s5 = "Frodo";       // copy of "Frodo"
        string s6 = s5;            // copy of s5

        string s7(s5,3,2);         // s5[3] and s5[4]; that is "do"
        string s8(p+7,3);          // p[7], p[8], and p[9]
        string s9(p,7,3);          // string(string(p),7,3), possibly expensive

        string s10(v.begin(),v.end());     // copy all characters from v
    }
```

Characters are numbered starting at position ***0*** so that a string is a sequence of characters numbered ***0*** to ***length*( )-*1***.

The ***length*( )** of a string is simply a synonym for its ***size*( )**; both functions return the number of characters in the string. Note that they do not count a C-string-style, zero-terminator character (§20.4.1). An implementation of ***basic_string*** stores its length rather than relying on a terminator.

Substrings are expressed as a character position plus a number of characters. The default value ***npos*** is initialized to the largest possible value and used to mean ''all of the elements.''

There is no constructor for creating a string of ***n*** unspecified characters. The closest we come to that is the constructor that makes a string of ***n*** copies of a given character. The length of a string is determined by the number of characters it holds at any give time. This allows the compiler to save the programmer from silly mistakes such as the definitions of ***s2*** and ***s3*** in the previous example.

The copy constructor is the constructor taking four arguments. Three of those arguments have defaults. For efficiency, that constructor could be implemented as two separate constructors. The user wouldn't be able to tell without actually looking at the generated code.

The constructor that is a template member is the most general. It allows a string to be initialized with values from an arbitrary sequence. In particular, it allows a string to be initialized with elements of a different character type as long as a conversion exists. For example:

```
    void f(string s)
    {
        wstring ws(s.begin(),s.end());     // copy all characters from s
        // ...
    }
```

Each ***wchar_t*** in ***ws*** is initialized by its corresponding ***char*** from ***s***.

20.3.5 Errors [string.error]

Often, strings are simply read, written, printed, stored, compared, copied, etc. This causes no problems, or, at worst, performance problems. However, once we start manipulating individual substrings and characters to compose new string values from existing ones, we sooner or later make mistakes that could cause us to write beyond the end of a string.

For explicit access to individual characters, ***at*( )** checks and throws ***out_of_range*( )** if we try to access beyond the end of the string; **[ ]** does not.

Most string operations take a character position plus a number of characters. A position larger than the size of the string throws an ***out_of_range*** exception. A ''too large'' character count is simply taken to be equivalent to ''the rest'' of the characters. For example:

```
void f()
{
    string s = "Snobol4";
    string s2(s,100,2); // character position beyond end of string: throw out_of_range()
    string s3(s,2,100); // character count too large: equivalent to s3(s,2,s.size()-2)
    string s4(s,2,string::npos);    // the characters starting from s[2]
}
```

Thus, ‘‘too large’’ positions are to be avoided, but ‘‘too large’’ character counts are useful. In fact, ***npos*** is really just the largest possible value for ***size_type***.

We could try to give a negative position or character count:

```
void g(string& s)
{
    string s5(s,-2,3);  // large position!: throw out_of_range()
    string s6(s,3,-2);  // large character count!: ok
}
```

However, the ***size_type*** used to represent positions and counts is an ***unsigned*** type, so a negative number is simply a confusing way of specifying a large positive number (§16.3.4).

Note that the functions used to find substrings of a ***string*** (§20.3.11) return ***npos*** if they don’t find anything. Thus, they don’t throw exceptions. However, later using ***npos*** as a character position does.

A pair of iterators is another way of specifying a substring. The first iterator identifies a position, and the difference between two iterators is a character count. As usual, iterators are not range checked.

Where a C-style string is used, range checking is harder. When given a C-style string (a pointer to ***char***) as an argument, ***basic_string*** functions assume the pointer is not ***0***. When given character positions for C-style strings, they assume that the C-style string is long enough for the position to be valid. Be careful! In this case, being careful means being paranoid, except when using character literals.

All strings have ***length()<npos***. In a few cases, such as inserting one string into another (§20.3.9), it is possible (although not likely) to construct a string that is too long to be represented. In that case, a ***length_error*** is thrown. For example:

```
string s(string::npos,´a´);    // throw length_error()
```

20.3.6 Assignment [string.assign]

Naturally, assignment is provided for strings:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // assignment (a bit like vector and list: §16.3.4):
```

```
        basic_string& operator=(const basic_string& s);
        basic_string& operator=(const Ch* p);
        basic_string& operator=(Ch c);

        basic_string& assign(const basic_string&);
        basic_string& assign(const basic_string& s, size_type pos, size_type n);
        basic_string& assign(const Ch* p, size_type n);
        basic_string& assign(const Ch* p);
        basic_string& assign(size_type n, Ch c);
        template<class In> basic_string& assign(In first, In last);

        // ...
    };
```

Like other standard containers, ***string***s have value semantics. That is, when one string is assigned to another, the assigned string is copied and two separate strings with the same value exist after the assignment. For example:

```
    void g()
    {
        string s1 = "Knold";
        string s2 = "Tot";

        s1 = s2;          // two copies of "Tot"
        s2[1] = 'u';      // s2 is "Tut", s1 is still "Tot"
    }
```

Assignment with a single character to a string is supported even though initialization by a single character isn't:

```
    void f()
    {
        string s = 'a'; // error: initialization by char
        s = 'a';        // ok: assignment
        s = "a";
        s = s;
    }
```

Being able to assign a ***char*** to a ***string*** isn't much use and could even be considered error-prone. However, appending a ***char*** using **+=** is at times essential (§20.3.9), and it would be odd to be able to say ***s+='c'*** but not ***s=s+'c'***.

The name ***assign()*** is used for the assignments, which are the counterparts to multiple argument constructors (§16.3.4, §20.3.4).

As mentioned in §11.12, it is possible to optimize a ***string*** so that copying doesn't actually take place until two copies of a ***string*** are needed. The design of the standard ***string*** encourages implementations that minimize actual copying. This makes read-only uses of strings and passing of strings as function arguments much cheaper than one could naively have assumed. However, it would be equally naive for programmers not to check their implementations before writing code that relied on ***string*** copy being optimized (§20.6[13]).

20.3.7 Conversion to C-Style Strings [string.conv]

As shown in §20.3.4, a ***string*** can be initialized by a C-style string and C-style strings can be assigned to ***string***s. Conversely, it is possible to place a copy of the characters of a ***string*** into an array:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // conversion to C-style string:

    const Ch* c_str() const;
    const Ch* data() const;
    size_type copy(Ch* p, size_type n, size_type pos = 0) const;

    // ...
};
```

The ***data***( ) function writes the characters of the string into an array and returns a pointer to that array. The array is owned by the ***string***, and the user should not try to delete it. The user also cannot rely on its value after a subsequent call on a non-***const*** function on the string. The ***c_str***( ) function is like ***data***( ), except that it adds a 0 (zero) at the end as a C-string-style terminator. For example:

```
void f()
{
    string s = "equinox";          // s.length()==7
    const char* p1 = s.data();     // p1 points to seven characters
    printf("p1 = %s\n",p1);        // bad: missing terminator
    p1[2] = 'a';                   // error: p1 points to a const array
    s[2] = 'a';
    char c = p1[1];                // bad: access of s.data() after modification of s

    const char* p2 = s.c_str();    // p2 points to eight characters
    printf("p2 = %s\n",p2);        // ok: c_str() adds terminator
}
```

In other words, ***data***( ) produces an array of characters, whereas ***c_str***( ) produces a C-style string. These functions are primarily intended to allow simple use of functions that take C-style strings. Consequently, ***c_str***( ) tends to be more useful than ***data***( ). For example:

```
void f(string s)
{
    int i = atoi(s.c_str());    // get int value of digits in string (§20.4.1)
    // ...
}
```

Typically, it is best to leave characters in a ***string*** until you need them. However, if you can't use the characters immediately, you can copy them into an array rather than leave them in the buffer allocated by ***c_str***( ) or ***data***( ). The ***copy***( ) function is provided for that. For example:

```
char* c_string(const string& s)
{
    char* p = new char[s.length()+1];  // note: +1
    s.copy(p,string::npos);
    p[s.length()] = 0;                 // note: add terminator
    return p;
}
```

A call `s.copy(p,n,m)` copies at most `n` characters to `p` starting with `s[m]`. If there are fewer than `n` characters in `s` to copy, `copy()` simply copies all the characters there are.

Note that a `string` can contain the `0` character. Functions manipulating C-style strings will interprete such as `0` as a terminator. Be careful to put `0`s into a string only if you don't apply C-style functions to it or if you put the `0` there exactly to be a terminator.

Conversion to a C-style string could have been provided by an `operator const char*()` rather than `c_str()`. This would have provided the convenience of an implicit conversion at the cost of surprises in cases in which such a conversion was unexpected.

If you find `c_str()` appearing in your program with great frequency, it is probably because you rely heavily on C-style interfaces. Often, an interface that relies on `string`s rather than C-style strings is available and can be used to eliminate the conversions. Alternatively, you can avoid most of the explicit calls of `c_str()` by providing additional definitions of the functions that caused you to write the `c_str()` calls:

```
extern "C" int atoi(const char*);

int atoi(const string& s)
{
    return atoi(s.c_str());
}
```

20.3.8 Comparisons [string.compare]

Strings can be compared to strings of their own type and to arrays of characters with the same character type:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...

    int compare(const basic_string& s) const; // combined > and ==
    int compare(const Ch* p) const;

    int compare(size_type pos, size_type n, const basic_string& s) const;
    int compare(size_type pos, size_type n,
                const basic_string& s, size_type pos2, size_type n2) const;
    int compare(size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;

    // ...
};
```

When an argument ***n*** is supplied, only the ***n*** first characters will be compared. The comparison criterion used is ***char_traits<Ch>***'s ***compare*( )** (§20.2.1). Thus, ***s.compare*(*s2*)** returns ***0*** if the strings have the same value, a negative number if ***s*** is lexicographically before ***s2***, and a positive number otherwise.

A user cannot supply a comparison criterion the way it was done in §13.4. When that degree of flexibility is needed, we can use ***lexicographical_compare*( )** (§18.9), define a function like the one in §13.4, or write an explicit loop. For example, the ***toupper*( )** function (§20.4.2) allows us to write case-insensitive comparisons:

```
int cmp_nocase(const string& s, const string& s2)
{
    string::const_iterator p = s.begin();
    string::const_iterator p2 = s2.begin();

    while (p!=s.end() && p2!=s2.end()) {
        if (toupper(*p)!=toupper(*p2)) return (toupper(*p)<toupper(*p2)) ? -1 : 1;
        ++p;
        ++p2;
    }

    return (s2.size()==s.size()) ? 0 : (s.size()<s2.size()) ? -1 : 1; // size is unsigned
}

void f(const string& s, const string& s2)
{
    if (s == s2) {                      // case sensitive compare of s and s2
        // ...
    }

    if (cmp_nocase(s,s2)== 0) {         // case insensitive compare of s and s2
        // ...
    }

    // ...
}
```

The usual comparison operators ==, !=, >, <, >=, and <= are provided for ***basic_string***s:

```
template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch,Tr,A>&, const Ch*);

// similar declarations for !=, >, <, >=, and <=
```

Comparison operators are nonmember functions so that conversions can be applied in the same way to both operands (§11.2.3). The versions taking C-style strings are provided to optimize comparisons against string literals. For example:

```
void f(const string& name)
{
    if (name =="Obelix" || "Asterix"==name) {    // use optimized ==
        // ...
    }
}
```

20.3.9 Insert [string.insert]

Once created, a string can be manipulated in many ways. Of the operations that modify the value of a string, is one of the most common is appending to it – that is, adding characters to the end. Insertion at other points of a string is rarer:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // add characters after (*this)[length()– 1]:

    basic_string& operator+=(const basic_string& s);
    basic_string& operator+=(const Ch* p);
    basic_string& operator+=(Ch c);
    void push_back(Ch c);

    basic_string& append(const basic_string& s);
    basic_string& append(const basic_string& s, size_type pos, size_type n);
    basic_string& append(const Ch* p, size_type n);
    basic_string& append(const Ch* p);
    basic_string& append(size_type n, Ch c);
    template<class In> basic_string& append(In first, In last);

    // insert characters before (*this)[pos]:

    basic_string& insert(size_type pos, const basic_string& s);
    basic_string& insert(size_type pos, const basic_string& s, size_type pos2, size_type n);
    basic_string& insert(size_type pos, const Ch* p, size_type n);
    basic_string& insert(size_type pos, const Ch* p);
    basic_string& insert(size_type pos, size_type n, Ch c);

    // insert characters before p:

    iterator insert(iterator p, Ch c);
    void insert(iterator p, size_type n, Ch c);
    template<class In> void insert(iterator p, In first, In last);

    // ...
};
```

Basically, the variety of operations provided for initializing a string and assigning to a string is also available for appending and for inserting characters before some character position.

The += operator is provided as the conventional notation for the most common forms of append. For example:

```
string complete_name(const string& first_name, const string& family_name)
{
    string s = first_name;
    s += ´ ´;
    s += family_name;
    return s;
}
```

Appending to the end can be noticeably more efficient than inserting into other positions. For example:

```
string complete_name2(const string& first_name, const string& family_name) // poor algorithm
{
    string s = family_name;
    s.insert(s.begin(), ´ ´);
    return s.insert(0, first_name);
}
```

Insertion usually forces the ***string*** implementation to do extra memory management and to move characters around.

Because ***string*** has a ***push_back()*** operation (§16.3.5), a ***back_inserter*** can be used for a ***string*** exactly as for general containers.

20.3.10 Concatenation [string.cat]

Appending is a special form of concatenation. *Concatenation* – constructing a string out of two strings by placing one after the other – is provided by the + operator:

```
template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A>
operator+(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const Ch*, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(Ch, const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, const Ch*);

template<class Ch, class Tr, class A>
basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, Ch);
```

As usual, + is defined as a nonmember function. For templates with several template parameters, this implies a notational disadvantage, since the template parameters are mentioned repeatedly.

On the other hand, use of concatenation is obvious and convenient. For example:

```
string complete_name3(const string& first_name, const string& family_name)
{
    return first_name + ´ ´ + family_name;
}
```

This notational convenience may be bought at the cost of some run-time overhead compared to *complete_name*( ). One extra temporary (§11.3.2) is needed in *complete_name3*( ). In my experience, this is rarely important, but it is worth remembering when writing an inner loop of a program where performance matters. In that case, we might even consider avoiding a function call by making *complete_name*( ) inline and composing the result string in place using lower-level operations (§20.6[14]).

20.3.11 Find [string.find]

There is a bewildering variety of functions for finding substrings:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // find subsequence (like search() §18.5.5):

    size_type find(const basic_string& s, size_type i = 0) const;
    size_type find(const Ch* p, size_type i, size_type n) const;
    size_type find(const Ch* p, size_type i = 0) const;
    size_type find(Ch c, size_type i = 0) const;

    // find subsequence searching backwards from the end (like find_end(), §18.5.5):

    size_type rfind(const basic_string& s, size_type i = npos) const;
    size_type rfind(const Ch* p, size_type i, size_type n) const;
    size_type rfind(const Ch* p, size_type i = npos) const;
    size_type rfind(Ch c, size_type i = npos) const;

    // find character (like find_first_of() in §18.5.2):

    size_type find_first_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_of(const Ch* p, size_type i = 0) const;
    size_type find_first_of(Ch c, size_type i = 0) const;

    // find character from argument searching backwards from the end:

    size_type find_last_of(const basic_string& s, size_type i = npos) const;
    size_type find_last_of(const Ch* p, size_type i, size_type n) const;
    size_type find_last_of(const Ch* p, size_type i = npos) const;
    size_type find_last_of(Ch c, size_type i = npos) const;

    // find character not in argument:

    size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_not_of(const Ch* p, size_type i = 0) const;
    size_type find_first_not_of(Ch c, size_type i = 0) const;

    // find character not in argument searching backwards from the end:
```

```
        size_type find_last_not_of(const basic_string& s, size_type i = npos) const;
        size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
        size_type find_last_not_of(const Ch* p, size_type i = npos) const;
        size_type find_last_not_of(Ch c, size_type i = npos) const;
        // ...
};
```

These are all ***const*** members. That is, they exist to locate a substring for some use, but they do not change the value of the string to which they are applied.

The meaning of the ***basic_string::find*** functions can be understood from their general algorithm equivalents. Consider an example:

```
void f()
{
        string s = "accdcde";
        string::size_type i1 = s.find("cd");                    // i1 = 2   s[2]=='c' && s[3]=='d'
        string::size_type i2 = s.rfind("cd");                   // i2 = 4   s[4]=='c' && s[5]=='d'
        string::size_type i3 = s.find_first_of("cd");           // i3 = 1   s[1] == 'c'
        string::size_type i4 = s.find_last_of("cd");            // i4 = 5   s[5] == 'd'
        string::size_type i5 = s.find_first_not_of("cd");       // i5 = 0   s[0]!='c' && s[0]!='d'
        string::size_type i6 = s.find_last_not_of("cd");        // i6 = 6   s[6]!='c' && s[6]!='d'
}
```

If a ***find()*** function fails to find anything, it returns ***npos***, which represents an illegal character position. If ***npos*** is used as a character position, ***range_error*** will be thrown (§20.3.5).

Note that result of a ***find()*** is an ***unsigned*** value.

20.3.12 Replace [string.replace]

Once a position in a string is identified, the value of individual character positions can be changed using subscripting or whole substrings can be replaced with new characters using ***replace()***:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        // ...
        // replace [ (*this)[i], (*this)[i+n] [ with other characters:

        basic_string& replace(size_type i, size_type n, const basic_string& s);
        basic_string& replace(size_type i, size_type n,
                              const basic_string& s, size_type i2, size_type n2);
        basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2);
        basic_string& replace(size_type i, size_type n, const Ch* p);
        basic_string& replace(size_type i, size_type n, size_type n2, Ch c);

        basic_string& replace(iterator i, iterator i2, const basic_string& s);
        basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n);
        basic_string& replace(iterator i, iterator i2, const Ch* p);
        basic_string& replace(iterator i, iterator i2, size_type n, Ch c);
        template<class In> basic_string& replace(iterator i, iterator i2, In j, In j2);
```

```
        // remove characters from string (''replace with nothing''):

        basic_string& erase(size_type i = 0, size_type n = npos);
        iterator erase(iterator i);
        iterator erase(iterator first, iterator last);

        // ...
};
```

Note that the number of new characters need not be the same as the number of characters previously in the string. The size of the string is changed to accommodate the new substring. In particular, *erase*( ) simply removes a substring and adjusts its size accordingly. For example:

```
void f()
{
        string s = "but I have heard it works even if you don´t believe in it";
        s.erase(0,4);                          // erase initial "but "
        s.replace(s.find("even"),4,"only");
        s.replace(s.find("don´t"),5,"");   // erase by replacing with ""
}
```

The simple call *erase*( ), with no argument, makes the string into an empty string. This is the operation that is called *clear*( ) for general containers (§16.3.6).

The variety of *replace*( ) functions matches that of assignment. After all, *replace*( ) is an assignment to a substring.

20.3.13 Substrings [string.sub]

The *substr*( ) function lets you specify a substring as a position plus a length:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
        // ...
        // address substring:

        basic_string substr(size_type i = 0, size_type n = npos) const;
        // ...
};
```

The *substr*( ) function is simply a way of reading a part of a string. On the other hand, *replace*( ) lets you write to a substring. Both rely on the low-level position plus number of characters notation. However, *find*( ) lets us find substrings by value. Together, they allow us to define a substring that can be used for both reading and writing:

```
template<class Ch> class Basic_substring {
public:
        typedef typename basic_string<Ch>::size_type size_type;

        Basic_substring(basic_string<Ch>& s, size_type i, size_type n);        // s[i]..s[i+n–1]
        Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2);  // s2 in s
        Basic_substring(basic_string<Ch>& s, const Ch* p);                  // *p in s
```

```
        Basic_substring& operator=(const basic_string<Ch>&);          // write through to *ps
        Basic_substring& operator=(const Basic_substring<Ch>&);
        Basic_substring& operator=(const Ch*);
        Basic_substring& operator=(Ch);

        operator basic_string<Ch>() const;                            // read from *ps
        operator Ch* () const;
    private:
        basic_string<Ch>* ps;
        size_type pos;
        size_type n;
    };
```

The implementation is largely trivial. For example:

```
    template<class Ch>
    Basic_substring<Ch>::Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2)
        :ps(&s), n(s2.length())
    {
        pos = s.find(s2);
    }

    template<class Ch>
    Basic_substring<Ch>& Basic_substring<Ch>::operator=(const basic_string<Ch>& s)
    {
        ps->replace(pos,n,s);     // write through to *ps
        return *this;
    }

    template<class Ch> Basic_substring<Ch>::operator basic_string<Ch>() const
    {
        return basic_string<Ch>(*ps,pos,n);      // copy from *ps
    }
```

If *s2* isn't found in *s*, *pos* will be *npos*. Attempts to read or write it will throw *range_error* (§20.3.5).

This *Basic_substring* can be used like this:

```
    typedef Basic_substring<char> Substring;

    void f()
    {
        string s = "Mary had a little lamb";
        Substring(s,"lamb") = "fun";
        Substring(s,"a little") = "no";
        string s2 = "Joe" + Substring(s,s.find(´ ´),string::npos);
    }
```

Naturally, this would be much more interesting if *Substring* could do some pattern matching (§20.6[7]).

20.3.14 Size and Capacity [string.capacity]

Memory-related issues are handled much as they are for *vector* (§16.3.8):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // size, capacity, etc. (like §16.3.8):

    size_type size() const;                          // number of characters (§20.3.4)
    size_type max_size() const;                      // largest possible string
    size_type length() const { return size(); }
    bool empty() const { return size()==0; }

    void resize(size_type n, Ch c);
    void resize(size_type n) { resize(n,Ch()); }

    size_type capacity() const;                      // like vector: §16.3.8
    void reserve(size_type res_arg = 0);             // like vector: §16.3.8

    allocator_type get_allocator() const;
};
```

A call *reserve*(*res_arg*) throws *length_error* if *res_arg*>*max_size*().

20.3.15 I/O Operations [string.io]

One of the main uses of *string*s is as the target of input and as the source of output:

```
template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&,const basic_string<Ch,Tr,A>&);

template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&, Ch eol);

template<class Ch, class Tr, class A>
basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&, basic_string<Ch,Tr,A>&);
```

The << operator writes a string to an *ostream* (§21.2.1). The >> operator reads a whitespace-terminated word (§3.6, §21.3.1) to its string, expanding the string as needed to hold the word. Initial whitespace is skipped, and the terminating whitespace character is not entered into the string.

The *getline*() function reads a line terminated by *eol* to its string, expanding string as needed to hold the line (§3.6). If no *eol* argument is provided, a newline ´*\n*´ is used as the delimiter. The line terminator is removed from the stream but not entered into the string. Because a *string* expands to hold the input, there is no reason to leave the terminator in the stream or to provide a count of characters read in the way *get*() and *getline*() do for character arrays (§21.3.4).

20.3.16 Swap [string.swap]

As for *vector*s (§16.3.9), a *swap*( ) function for strings can be much more efficient than the general algorithm, so a specific version is provided:

```
template<class Ch, class Tr, class A>
void swap(basic_string<Ch,Tr,A>&, basic_string<Ch,Tr,A>&);
```

20.4 The C Standard Library [string.cstd]

The C++ standard library inherited the C-style string functions from the C standard library. This section lists some of the most useful C string functions. The description is not meant to be exhaustive; for further information, check your reference manual. Beware that implementers often add their own nonstandard functions to the standard header files, so it is easy to get confused about which functions are guaranteed to be available on every implementation.

The headers presenting the standard C library facilities are listed in §16.1.2. Memory management functions can be found in §19.4.6, C I/O functions in §21.8, and the C math library in §22.3. The functions concerned with startup and termination are described in §3.2 and §9.4.1.1, and the facilities for reading unspecified function arguments are presented in §7.6. C-style functions for wide character strings are found in *<cwchar>* and *<wchar.h>*.

20.4.1 C-Style Strings [string.c]

Functions for manipulating C-style strings are found in *<string.h>* and *<cstring>*:

```
char* strcpy(char* p, const char* q);          // copy from q into p (incl. terminator)
char* strcat(char* p, const char* q);          // append from q to p (incl. terminator)
char* strncpy(char* p, const char* q, int n);  // copy n char from q into p
char* strncat(char* p, const char* q, int n);  // append n char from q to p

size_t strlen(const char* p);    // length of p (not counting the terminator)

int strcmp(const char* p, const char* q);          // compare: p and q
int strncmp(const char* p, const char* q, int n);  // compare first n char

char* strchr(char* p, int c);                      // find first c in p
const char* strchr(const char* p, int c);
char* strrchr(char* p, int c);                     // find last c in p
const char* strrchr(const char* p, int c);
char* strstr(char* p, const char* q);              // find first q in p
const char* strstr(const char* p, const char* q);

char* strpbrk(char* p, const char* q);             // find first char from q in p
const char* strpbrk(const char* p, const char* q);

size_t strspn(const char* p, const char* q);    // number of char in p before any char in q
size_t strcspn(const char* p, const char* q);   // number of char in p before a char not in q
```

A pointer is assumed to be nonzero, and the array of *char* that it points to is assumed to be terminated by *0*. The *strn*-functions pad with *0* if there are not *n* characters to copy. String comparisons

return *0* if the strings are equal, a negative number if the first argument is lexicographically before the second, and a positive number otherwise.

Naturally, C doesn't provide the pairs of overloaded functions. However, they are needed in C++ for *const* safety. For example:

```
void f(const char* pcc, char* pc)    // C++
{
    *strchr(pcc,´a´) = ´b´;  // error: cannot assign to const char
    *strchr(pc,´a´) = ´b´;   // ok, but sloppy: there might not be an 'a' in pc
}
```

The C++ *strchr*( ) does not allow you to write to a *const*. However, a C program may ''take advantage'' of the weaker type checking in the C *strchr*( ):

```
char* strchr(const char* p, int c);  /* C standard library function, not C++ */

void g(const char* pcc, char* pc)   /* C, will not compile in C++ */
{
    *strchr(pcc,´a´) = ´b´;  /* converts const to non-const: ok in C, error in C++ */
    *strchr(pc,´a´) = ´b´;   /* ok in C and C++ */
}
```

Whenever possible, C-style strings are best avoided in favor of *string*s. C-style strings and their associated standard functions can be used to produce very efficient code, but even experienced C and C++ programmers are prone to make uncaught ''silly errors'' when using them. However, no C++ programmer can avoid seeing some of these functions in old code. Here is a nonsense example illustrating the most common functions:

```
void f(char* p, char* q)
{
    if (p==q) return;            // pointers are equal
    if (strcmp(p,q)==0) {        // string values are equal
        int i = strlen(p);       // number of characters (not counting the terminator)
        // ...
    }
    char buf[200];
    strcpy(buf,p);               // copy p into buf (including the terminator)
                                 // sloppy: will overflow some day.
    strncpy(buf,p,200);          // copy 200 char from p into buf
                                 // sloppy: will fail to copy the terminator some day.
    // ...
}
```

Input and output of C-style strings are usually done using the *printf* family of functions (§21.8).

In *<stdlib.h>* and *<cstdlib>*, the standard library provides useful functions for converting strings representing numeric values into numeric values:

```
double atof(const char* p);    // convert p to double
int atoi(const char* p);       // convert p to int
long atol(const char* p);      // convert p to long
```

Leading whitespace is ignored. If the string doesn't represent a number, zero is returned. For

example, the value of *atoi*("*seven*") is *0*. If the string represents a number that cannot be represented in the intended result type, *errno* (§16.1.2, §22.3) is set to *ERANGE* and an appropriately huge or tiny value is returned.

20.4.2 Character Classification [string.isalpha]

In <*ctype.h*> and <*cctype*>, the standard library provides a set of useful functions for dealing with ASCII and similar character sets:

```
int isalpha(int);      // letter: 'a'..'z' 'A'..'Z' in C locale (§20.2.1, §21.7)
int isupper(int);      // upper case letter: 'A'..'Z' in C locale (§20.2.1, §21.7)
int islower(int);      // lower case letter: 'a'..'z' in C locale (§20.2.1, §21.7)
int isdigit(int);      // '0'..'9'
int isxdigit(int);     // '0'..'9' or letter
int isspace(int);      // ' ' '\t' '\v' return newline formfeed
int iscntrl(int);      // control character (ASCII 0..31 and 127)
int ispunct(int);      // punctuation: none of the above
int isalnum(int);      // isalpha() | isdigit()
int isprint(int);      // printable: ascii ' '..'~'
int isgraph(int);      // isalpha() | isdigit() | ispunct()

int toupper(int c);    // uppercase equivalent to c
int tolower(int c);    // lowercase equivalent to c
```

All are usually implemented by a simple lookup, using the character as an index into a table of character attributes. This means that constructs such as:

```
if ( ( 'a' <= c && c <= 'z' ) || ( 'A' <= c && c <= 'Z' ) ) {      // alphabetic
        // ...
}
```

are inefficient in addition to being tedious to write and error-prone (on a machine with the EBCDIC character set, this will accept nonalphabetic characters).

These functions take *int* arguments, and the integer passed must be representable as an *unsigned char* or *EOF* (which is most often *-1*). This can be a problem on systems where *char* is signed (see §20.6[11]).

Equivalent functions for wide characters are found in <*cwtype*> and <*wtype.h*>.

20.5 Advice [string.advice]

[1] Prefer *string* operations to C-style string functions; §20.4.1.
[2] Use *string*s as variables and members, rather than as base classes; §20.3, §25.2.1.
[3] You can pass *string*s as value arguments and return them by value to let the system take care of memory management; §20.3.6.
[4] Use *at*( ) rather than iterators or [ ] when you want range checking; §20.3.2, §20.3.5.
[5] Use iterators and [ ] rather than *at*( ) when you want to optimize speed; §20.3.2, §20.3.5.
[6] Directly or indirectly, use *substr*( ) to read substrings and *replace*( ) to write substrings; §20.3.12, §20.3.13.

[7] Use the *find*( ) operations to localize values in a *string* (rather than writing an explicit loop); §20.3.11.
[8] Append to a *string* when you need to add characters efficiently; §20.3.9.
[9] Use *string*s as targets of non-time-critical character input; §20.3.15.
[10] Use *string*::*npos* to indicate ''the rest of the *string*;'' §20.3.5.
[11] If necessary, implement heavily-used *string*s using low-level operations (rather than using low-level data structures everywhere); §20.3.10.
[12] If you use *string*s, catch *range_error* and *out_of_range* somewhere; §20.3.5.
[13] Be careful not to pass a *char** with the value *0* to a string function; §20.3.7.
[14] Use *c_str* rather to produce a C-style string representation of a *string* only when you have to; §20.3.7.
[15] Use *isalpha*( ), *isdigit*( ), etc., when you need to know the classification of a character rather that writing your own tests on character values; §20.4.2.

20.6 Exercises [string.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

1. (∗2) Write a function that takes two *string*s and returns a *string* that is the concatenation of the strings with a dot in the middle. For example, given *file* and *write*, the function returns *file*.*write*. Do the same exercise with C-style strings using only C facilities such as *malloc*( ) and *strlen*( ). Compare the two functions. What are reasonable criteria for a comparison?
2. (∗2) Make a list of differences between *vector* and *basic_string*. Which differences are important?
3. (∗2) The string facilities are not perfectly regular. For example, you can assign a *char* to a string, but you cannot initialize a *string* with a *char*. Make a list of such irregularities. Which could have been eliminated without complicating the use of strings? What other irregularities would this introduce?
4. (∗1.5) Class *basic_string* has a lot of members. Which could be made nonmember functions without loss of efficiency or notational convenience?
5. (∗1.5) Write a version of *back_inserter*( ) (§19.2.4) that works for *basic_string*.
6. (∗2) Complete *Basic_substring* from §20.3.13 and integrate it with a *String* type that overloads ( ) to mean ''substring of'' and otherwise acts like *string*.
7. (∗2.5) Write a *find*( ) function that finds the first match for a simple regular expression in a *string*. Use ? to mean ''any character,'' * to mean any number of characters not matching the next part of the regular expression, and [*abc*] to mean any character from the set specified between the square braces (here *a*, *b*, and *c*). Other characters match themselves. For example, *find*(*s*,"*name*:") returns a pointer to the first occurrence of *name*: in *s*; *find*(*s*,"[*nN*]*ame*:") returns a pointer to the first occurrence of *name*: or *Name*: in *s*; and *find*(*s*,"[*nN*]*ame*(*)") returns a pointer to the first occurence of *Name* or *name* followed by a (possibly empty) parenthesized sequences of characters in *s*.
8. (∗2.5) What operations do you find missing from the simple regular expression function from

§20.6[7]? Specify and add them. Compare the expressiveness of your regular expression matcher to that of a widely distributed one. Compare the performance of your regular expression matcher to that of a widely distributed one.

9. (∗2.5) Use a regular expression library to implement pattern-matching operations on a ***String*** class that has an associated ***Substring*** class.
10. (∗2.5) Consider writing an ''ideal'' class for general text processing. Call it ***Text***. What facilities should it have? What implementation constraints and overheads are imposed by your set of ''ideal'' facilities?
11. (∗1.5) Define a set of overloaded versions for ***isalpha***( ), ***isdigit***( ), etc., so that these functions work correctly for ***char***, ***unsigned char***, and ***signed char***.
12. (∗2.5) Write a ***String*** class optimized for strings having no more than eight characters. Compare its performance to that of the ***String*** from §11.12 and your implementation's version of the standard library ***string***. Is it possible to design a string that combines the advantages of a string optimized for very short strings with the advantages of a perfectly general string?
13. (∗2) Measure the performance of copying of ***string***s. Does your implementation's implementation of ***string*** adequately optimize copying?
14. (∗2.5) Compare the performance of the three ***complete_name***( ) functions from §20.3.9 and §20.3.10. Try to write a version of ***complete_name***( ) that runs as fast as possible. Keep a record of mistakes found during its implementation and testing.
15. (∗2.5) Imagine that reading medium-long strings (most are 5 to 25 characters long) from ***cin*** is the bottleneck in your system. Write an input function that reads such strings as fast as you can think of. You can choose the interface to that function to optimize for speed rather than for convenience. Compare the result to your implementation's >> for ***string***s.
16. (∗1.5) Write a function ***itos***(***int***) that returns a ***string*** representing its ***int*** argument.

.

21

Streams

What you see is all you get.
– Brian Kernighan

Input and output — ***ostream***s — output of built-in types — output of user-defined types — virtual output functions — ***istream***s — input of built-in types — unformatted input — stream state — input of user-defined types — I/O exceptions — tying of streams — sentries — formatting integer and floating-point output — fields and adjustments — manipulators — standard manipulators — user-defined manipulators — file streams — closing streams — string streams — stream buffers — locale — stream callbacks — ***printf*( ) — advice — exercises.

21.1 Introduction [io.intro]

Designing and implementing a general input/output facility for a programming language is notoriously difficult. Traditionally, I/O facilities have been designed exclusively to handle a few built-in data types. However, a nontrivial C++ program uses many user-defined types, and the input and output of values of those types must be handled. An I/O facility should be easy, convenient, and safe to use; efficient and flexible; and, above all, complete. Nobody has come up with a solution that pleases everyone. It should therefore be possible for a user to provide alternative I/O facilities and to extend the standard I/O facilities to cope with special applications.

C++ was designed to enable a user to define new types that are as efficient and convenient to use as built-in types. It is therefore a reasonable requirement that an I/O facility for C++ should be provided in C++ using only facilities available to every programmer. The stream I/O facilities presented here are the result of an effort to meet this challenge:

§21.2 *Output:* What the application programmer thinks of as output is really the conversion of objects of types, such as ***int***, ***char****, and ***Employee_record***, into sequences of characters. The facilities for writing built-in and user-defined types to output are described.

§21.3 *Input:* The facilities for requesting input of characters, strings, and values of other built-in and user-defined types are presented.

§21.4 *Formatting:* There are often specific requirements for the layout of the output. For example, ***int***s may have to be printed in decimal and pointers in hexadecimal or floating-point numbers must appear with exactly specified precision. Formatting controls and the programming techniques used to provide them are discussed.

§21.5 *Files and Streams:* By default, every C++ program can use standard streams, such as standard output (***cout***), standard input (***cin***), and error output (***cerr***). To use other devices or files, streams must be created and attached to those files or devices. The mechanisms for opening and closing files and for attaching streams to files and ***string***s are described.

§21.6 *Buffering:* To make I/O efficient, we must use a buffering strategy that is suitable for both the data written (read) and the destination it is written to (read from). The basic techniques for buffering streams are presented.

§21.7 *Locale:* A ***locale*** is an object that specifies how numbers are printed, what characters are considered letters, etc. It encapsulates many cultural differences. Locales are implicitly used by the I/O system and are only briefly described here.

§21.8 *C I/O:* The ***printf*( )** function from the C ***<stdio.h>*** library and the C library's relation to the C++ ***<iostream>*** library are discussed.

Knowledge of the techniques used to implement the stream library is not needed to use the library. Also, the techniques used for different implementations will differ. However, implementing I/O is a challenging task. An implementation contains examples of techniques that can be applied to many other programming and design tasks. Therefore, the techniques used to implement I/O are worthy of study.

This chapter discusses the stream I/O system to the point where you should be able to appreciate its structure, to use it for most common kinds of I/O, and to extend it to handle new user-defined types. If you need to implement the standard streams, provide a new kind of stream, or provide a new locale, you need a copy of the standard, a good systems manual, and/or examples of working code in addition to what is presented here.

The key components of the stream I/O systems can be represented graphically like this:

The dotted arrow from ***basic_iostream<>*** indicates that ***basic_ios<>*** is a virtual base class; the solid arrows represent pointers. The classes marked with <> are templates parameterized by a character type and containing a ***locale***.

The streams concept and the general notation it provides can be applied to a large class of communication problems. Streams have been used for transmitting objects between machines (§25.4.1), for encrypting message streams (§21.10[22]), for data compression, for persistent storage of objects, and much more. However, the discussion here is restricted to simple character-oriented input and output.

Declarations of stream I/O classes and templates (sufficient to refer to them but not to apply operations to them) and standard ***typedef***s are presented in ***<iosfwd>***. This header is occasionally needed when you want to include some but not all of the I/O headers.

21.2 Output [io.out]

Type-safe and uniform treatment of both built-in and user-defined types can be achieved by using a single overloaded function name for a set of output functions. For example:

```
put(cerr,"x = "); // cerr is the error output stream
put(cerr,x);
put(cerr,´\n´);
```

The type of the argument determines which ***put*** function will be invoked for each argument. This solution is used in several languages. However, it is repetitive. Overloading the operator << to mean ‘‘put to’’ gives a better notation and lets the programmer output a sequence of objects in a single statement. For example:

```
cerr << "x = " << x << ´\n´;
```

If ***x*** is an ***int*** with the value ***123***, this statement would print

```
x = 123
```

followed by a newline onto the standard error output stream, ***cerr***. Similarly, if ***x*** is of type ***complex*** (§22.5) with the value ***(1,2.4)***, the statement will print

```
x = (1,2.4)
```

on ***cerr***. This style can be used as long as ***x*** is of a type for which operator << is defined and a user can trivially define operator << for a new type.

An output operator is needed to avoid the verbosity that would have resulted from using an output function. But why <<? It is not possible to invent a new lexical token (§11.2). The assignment operator was a candidate for both input and output, but most people seemed to prefer to use different operators for input and output. Furthermore, = binds the wrong way; that is, ***cout=a=b*** means ***cout=(a=b)*** rather than ***(cout=a)=b*** (§6.2). I tried the operators < and >, but the meanings ‘‘less than’’ and ‘‘greater than’’ were so firmly implanted in people’s minds that the new I/O statements were for all practical purposes unreadable.

The operators << and >> are not used frequently enough for built-in types to cause that problem. They are symmetric in a way that can be used to suggest ‘‘to’’ and ‘‘from.’’ When they are

used for I/O, I refer to << as *put to* and to >> as *get from*. People who prefer more technical-sounding names call them *inserters* and *extractors*, respectively. The precedence of << is low enough to allow arithmetic expressions as operands without using parentheses. For example:

```
cout << "a*b+c=" << a*b+c << '\n';
```

Parentheses must be used to write expressions containing operators with precedence lower than <<'s. For example:

```
cout << "a^b|c=" << (a^b|c) << '\n';
```

The left shift operator (§6.2.4) can be used in an output statement, but of course it, too, must appear within parentheses:

```
cout << "a<<b=" << (a<<b) << '\n';
```

21.2.1 Output Streams [io.ostream]

An ***ostream*** is a mechanism for converting values of various types into sequences of characters. Usually, these characters are then output using lower-level output operations. There are many kinds of characters (§20.2) that can be characterized by ***char_traits*** (§20.2.1). Consequently, an ***ostream*** is a specialization for a particular kind of character of a general ***basic_ostream*** template:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
    virtual ~basic_ostream();
    // ...
};
```

This template and its associated output operations are defined in namespace ***std*** and presented by ***<ostream>***, which contains the output-related parts of ***<iostream>***.

The ***basic_ostream*** template parameters control the type of characters that is used by the implementation; they do not affect the types of values that can be output. Streams implemented using ordinary ***char***s and streams implemented using wide characters are directly supported by every implementation:

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
```

On many systems, it is possible to optimize writing of wide characters through ***wostream*** to an extent that is hard to match for streams using bytes as the unit of output.

It is possible to define streams for which the physical I/O is not done in terms of characters. However, such streams are beyond the scope of the C++ standard and beyond the scope of this book (§21.10[15]).

The ***basic_ios*** base class is presented in ***<ios>***. It controls formatting (§21.4), locale (§21.7), and access to buffers (§21.6). It also defines a few types for notational convenience:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type;    // type of integer value of character
    typedef typename Tr::pos_type pos_type;  // position in buffer
    typedef typename Tr::off_type off_type;    // offset in buffer

    // ... see also §21.3.3, §21.3.7, §21.4.4, §21.6.3, and §21.7.1 ...
};
```

The *ios_base* base class contains information and operations that are independent of the character type used, such as the precision used for floating-point output. It therefore doesn't need to be a template.

In addition to the *typedef*s in *ios_base*, the stream I/O library uses a signed integral type *streamsize* to represent the number of characters transferred in an I/O operation and the size of I/O buffers. Similarly, a *typedef* called *streamoff* is supplied for expressing offsets in streams and buffers.

Several standard streams are declared in *<iostream>*:

```
ostream cout;        // standard output stream of char
ostream cerr;        // standard unbuffered output stream for error messages
ostream clog;        // standard output stream for error messages

wostream wcout;      // wide stream corresponding to cout
wostream wcerr;      // wide stream corresponding to cerr
wostream wclog;      // wide stream corresponding to clog
```

The *cerr* and *clog* streams refer to the same output destination; they simply differ in the buffering they provide. The *cout* writes to the same destination as C's *stdout* (§21.8), while *cerr* and *clog* write to the same destination as C's *stderr*. The programmer can create more streams as needed (see §21.5).

21.2.2 Output of Built-In Types [io.out.builtin]

The class *ostream* is defined with the operator << (''put to'') to handle output of the built-in types:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
    // ...

    basic_ostream& operator<<(short n);
    basic_ostream& operator<<(int n);
    basic_ostream& operator<<(long n);

    basic_ostream& operator<<(unsigned short n);
    basic_ostream& operator<<(unsigned int n);
    basic_ostream& operator<<(unsigned long n);
```

```
        basic_ostream& operator<<(float f);
        basic_ostream& operator<<(double f);
        basic_ostream& operator<<(long double f);

        basic_ostream& operator<<(bool n);
        basic_ostream& operator<<(const void* p);          // write pointer value

        basic_ostream& put(Ch c);      // write c
        basic_ostream& write(const Ch* p, streamsize n);   // p[0]..p[n-1]

        // ...
    };
```

An ***operator<<( )*** returns a reference to the ***ostream*** for which it was called so that another ***operator<<( )*** can be applied to it. For example,

```
cerr << "x = " << x;
```

where ***x*** is an ***int***, will be interpreted as:

```
(cerr.operator<<("x = ")).operator<<(x);
```

In particular, this implies that when several items are printed by a single output statement, they will be printed in the expected order: left to right. For example:

```
void val(char c)
{
    cout << "int(´" << c << "´) = " << int(c) << ´\n´;
}

int main()
{
    val(´A´);
    val(´Z´);
}
```

On an implementation using ASCII characters, this will print:

```
int(´A´) = 65
int(´Z´) = 90
```

Note that a character literal has type ***char*** (§4.3.1) so that ***cout<<´Z´*** will print the letter ***Z*** and not the integer value ***90***.

A ***bool*** value will be output as ***0*** or ***1*** by default. If you don't like that, you can set the formatting flag ***boolalpha*** from ***<iomanip>*** (§21.4.6.2) and get ***true*** or ***false***. For example:

```
int main()
{
    cout << true << ´ ´ << false << ´\n´;
    cout << boolalpha;                    // use symbolic representation for true and false
    cout << true << ´ ´ << false << ´\n´;
}
```

This prints:

```
1 0
true false
```

More precisely, ***boolalpha*** ensures that we get a locale-dependent representation of ***bool*** values. By setting my locale (§21.7) just right, I can get:

```
1 0
sandt falsk
```

Formatting floating-point numbers, the base used for integers, etc., are discussed in §21.4.

The function ***ostream::operator<<(const void*)*** prints a pointer value in a form appropriate to the architecture of the machine used. For example,

```
int main()
{
    int i = 0;
    int* p = new int;
    cout << "local " << &i << ", free store " << p << '\n';
}
```

printed

```
local 0x7fffead0, free store 0x500c
```

on my machine. Other systems have different conventions for printing pointer values.

The ***put()*** and ***write()*** functions simply write characters. Consequently, the **<<** for outputting characters need not be a member. The ***operator<<()*** functions that take a character operand can be implemented as nonmembers using ***put()***:

```
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, Ch);
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, char);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, char);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, signed char);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, unsigned char);
```

Similarly, **<<** is provided for writing out zero-terminated character arrays:

```
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const Ch*);
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const char*);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const char*);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const signed char*);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const unsigned char*);
```

21.2.3 Output of User-Defined Types [io.out.udt]

Consider a user-defined type ***complex*** (§11.3):

```
class complex {
public:
    double real() const { return re; }
    double imag() const { return im; }
    // ...
};
```

Operator << can be defined for the new type ***complex*** like this:

```
ostream& operator<<(ostream&s, complex z)
{
    return s << '(' << z.real() << ',' << z.imag() << ')';
}
```

This << can then be used exactly like << for a built-in type. For example,

```
int main()
{
    complex x(1,2);
    cout << "x = " << x << '\n';
}
```

produces

```
x = (1,2)
```

Defining an output operation for a user-defined type does not require modification of the declaration of class ***ostream***. This is fortunate because ***ostream*** is defined in ***<iostream>***, which users cannot and should not modify. Not allowing additions to ***ostream*** also provides protection against accidental corruption of that data structure and makes it possible to change the implementation of an ***ostream*** without affecting user programs.

21.2.3.1 Virtual Output Functions [io.virtual]

The ***ostream*** members are not ***virtual***. The output operations that a programmer can add are not members, so they cannot be ***virtual*** either. One reason for this is to achieve close to optimal performance for simple operations such as putting a character into a buffer. This is a place where run-time efficiency is crucial and where inlining is a must. Virtual functions are used to achieve flexibility for the operations dealing with buffer overflow and underflow only (§21.6.4).

However, a programmer sometimes wants to output an object for which only a base class is known. Since the exact type isn't known, correct output cannot be achieved simply by defining a << for each new type. Instead, a virtual output function can be provided in the abstract base:

```
class My_base {
public:
        // ...

        virtual ostream& put(ostream& s) const = 0;        // write *this to s
};

ostream& operator<<(ostream& s, const My_base& r)
{
        return r.put(s);        // use the right put()
}
```

That is, *put*( ) is a virtual function that ensures that the right output operation is used in <<.

Given that, we can write:

```
class Sometype : public My_base {
public:
        // ...

        ostream& put(ostream& s) const;        // the real output function: override My_base::put()
};

void f(const My_base& r, Sometype& s)     // use << which calls the right put()
{
        cout << r << s;
}
```

This integrates the virtual *put*( ) into the framework provided by *ostream* and <<. The technique is generally useful to provide operations that act like virtual functions, but with the run-time selection based on their second argument.

21.3 Input [io.in]

Input is handled similarly to output. There is a class *istream* that provides an input operator >> (''get from'') for a small set of standard types. An *operator*>>( ) can then be defined for a user-defined type.

21.3.1 Input Streams [io.istream]

In parallel to *basic_ostream* (§21.2.1), *basic_istream* is defined in <*istream*>, which contains the input-related parts of <*iostream*>, like this:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_istream : virtual public basic_ios<Ch, Tr> {
public:
        virtual ~basic_istream( );

        // ...
};
```

The base class *basic_ios* is described in §21.2.1.

Two standard input streams ***cin*** and ***wcin*** are provided in ***<iostream>***:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

istream cin;      // standard input stream of char
wistream wcin;    // standard input stream of wchar_t
```

The ***cin*** stream reads from the same source as C's ***stdin*** (§21.8).

21.3.2 Input of Built-In Types [io.in.builtin]

An ***istream*** provides operator **>>** for the built-in types:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
        // ...
        // formatted input:

        basic_istream& operator>>(short& n);              // read into n
        basic_istream& operator>>(int& n);
        basic_istream& operator>>(long& n);

        basic_istream& operator>>(unsigned short& u);     // read into u
        basic_istream& operator>>(unsigned int& u);
        basic_istream& operator>>(unsigned long& u);

        basic_istream& operator>>(float& f);              // read into f
        basic_istream& operator>>(double& f);
        basic_istream& operator>>(long double& f);

        basic_istream& operator>>(bool& b);               // read into b
        basic_istream& operator>>(void*& p);              // read pointer value into p

        // ...
};
```

The ***operator>>( )*** input functions are defined in this style:

```
istream& istream::operator>>(T& tvar)     // T is a type for which istream::operator>> is declared
{
        // skip whitespace, then somehow read a T into 'tvar'
        return *this;
}
```

Because **>>** skips whitespace, you can read a sequence of whitespace-separated integers like this:

```
int read_ints(vector<int>& v)     // fill v, return number of ints read
{
        int i = 0;
        while (i<v.size() && cin>>v[i]) i++;
        return i;
}
```

A non-*int* on the input will cause the input operation to fail and thus terminate the input loop. For example, the input:

```
1 2 3 4 5.6 7 8.
```

will have *read_ints*( ) read in the five integers

```
1 2 3 4 5
```

and leave the dot as the next character to be read from input. Whitespace is defined as the standard C whitespace (blank, tab, newline, formfeed, and carriage return) by a call to *isspace*( ) as defined in *<cctype>* (§20.4.2).

The most common mistake when using *istream*s is to fail to notice that input didn't happen as expected because the input wasn't of the expected format. One should either check the state of an input stream (§21.3.3) before relying on values supposedly read in or use exceptions (§21.3.6).

The format expected for input is specified by the current locale (§21.7). By default, the *bool* values *true* and *false* are represented by *1* and *0*, respectively. Integers must be decimal and floating-point numbers of the form used to write them in a C++ program. By setting *base_field* (§21.4.2), it is possible to read *0123* as an octal number with the decimal value *83* and *0xff* as a hexadecimal number with the decimal value *255*. The format used to read pointers is completely implementation-dependent (have a look to see what your implementation does).

Surprisingly, there is no member >> for reading a character. The reason is simply that >> for characters can be implemented using the *get*( ) character input operations (§21.3.4), so it doesn't need to be a member. From a stream, we can read a character into the stream's character type. If that character type is *char*, we can also read into a *signed char* and *unsigned char*:

```
template<class Ch, class Tr>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, Ch&);

template<class Tr>
basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, unsigned char&);

template<class Tr>
basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, signed char&);
```

From a user's point of view, it does not matter whether a >> is a member.

Like the other >> operators, these functions first skip whitespace. For example:

```
void f()
{
    char c;
    cin >> c;
    // ...
}
```

This places the first non-whitespace character from *cin* into *c*.

In addition, we can read into an array of characters:

```
template<class Ch, class Tr>
    basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, Ch*);
template<class Tr>
    basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, unsigned char*);
template<class Tr>
    basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, signed char*);
```

These operations first skip whitespace. Then they read into their array operand until they encounter a whitespace character or end-of-file. Finally, they terminate the string with a *0*. Clearly, this offers ample opportunity for overflow, so reading into a *string* (§20.3.15) is usually better. However, you can specify a maximum for the number of characters to be read by >>: *is.width(n)* specifies that the next >> on *is* will read at most *n-1* characters into an array. For example:

```
void g()
{
    char v[4];
    cin.width(4);
    cin >> v;
    cout << "v = " << v << endl;
}
```

This will read at most three characters into *v* and add a terminating *0*.

Setting *width()* for an *istream* affects only the immediately following >> into an array and does not affect reading into other types of variables.

21.3.3 Stream State [io.state]

Every stream (*istream* or *ostream*) has a *state* associated with it. Errors and nonstandard conditions are handled by setting and testing this state appropriately.

The stream state is found in *basic_istream*'s base *basic_ios* from <*ios*>:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    bool good() const;   // next operation might succeed
    bool eof() const;    // end of input seen
    bool fail() const;   // next operation will fail
    bool bad() const;    // stream is corrupted

    iostate rdstate() const;                        // get io state flags
    void clear(iostate f = goodbit);                // set io state flags
    void setstate(iostate f) { clear(rdstate()|f); } // add f to io state flags

    operator void*() const;                         // nonzero if !fail()
    bool operator!() const { return fail(); }

    // ...
};
```

If the state is *good()* the previous input operation succeeded. If the state is *good()*, the next input

operation might succeed; otherwise, it will fail. Applying an input operation to a stream that is not in the *good*( ) state is a null operation. If we try to read into a variable *v* and the operation fails, the value of *v* should be unchanged (it is unchanged if *v* is a variable of one of the types handled by *istream* or *ostream* member functions). The difference between the states *fail*( ) and *bad*( ) is subtle. When the state is *fail*( ) but not also *bad*( ), it is assumed that the stream is uncorrupted and that no characters have been lost. When the state is *bad*( ), all bets are off.

The state of a stream is represented as a set of flags. Like most constants used to express the behavior of streams, these flags are defined in *basic_ios*' base *ios_base*:

```
class ios_base {
public:
    // ...

    typedef implementation_defined2 iostate;
    static const iostate badbit,      // stream is corrupted
                         eofbit,      // end-of-file seen
                         failbit,     // next operation will fail
                         goodbit;     // goodbit==0

    // ...
};
```

The I/O state flags can be directly manipulated. For example:

```
void f()
{
    ios_base::iostate s = cin.rdstate();  // returns a set of iostate bits

    if (s & ios_base::badbit) {
        // cin characters possibly lost
    }
    // ...
    cin.setstate(ios_base::failbit);
    // ...
}
```

When a stream is used as a condition, the state of the stream is tested by *operator void**( ) or *operator*!( ). The test succeeds only if the state is *good*( ). For example, a general copy function can be written like this:

```
template<class T> void iocopy(istream& is, ostream& os)
{
    T buf;
    while (is>>buf) os << buf << '\n';
}
```

The *is>>buf* returns a reference to *is*, which is tested by a call of *is*::*operator void**( ). For example:

```
void f(istream& i1, istream& i2, istream& i3, istream& i4)
{
    iocopy<complex>(i1,cout);      // copy complex numbers
    iocopy<double>(i2,cout);       // copy doubles
    iocopy<char>(i3,cout);         // copy chars
    iocopy<string>(i4,cout);       // copy whitespace-separated words
}
```

21.3.4 Input of Characters [io.in.unformatted]

The >> operator is intended for formatted input; that is, reading objects of an expected type and format. Where this is not desirable and we want to read characters as characters and then examine them, we use the *get*( ) functions:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
    // ...
    // unformatted input:

    streamsize gcount() const;      // number of char read by last get()

    int_type get();                 // read one Ch (or Tr::eof())

    basic_istream& get(Ch& c);      // read one Ch into c

    basic_istream& get(Ch* p, streamsize n);        // newline is terminator
    basic_istream& get(Ch* p, streamsize n, Ch term);

    basic_istream& getline(Ch* p, streamsize n);    // newline is terminator
    basic_istream& getline(Ch* p, streamsize n, Ch term);

    basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof());
    basic_istream& read(Ch* p, streamsize n);       // read at most n char

    // ...
};
```

The *get*( ) and *getline*( ) functions treat whitespace characters exactly like other characters. They are intended for input operations, where one doesn't make assumptions about the meanings of the characters read.

The function *istream*::*get*(*char*&) reads a single character into its argument. For example, a character-by-character copy program can be written like this:

```
int main()
{
    char c;
    while(cin.get(c)) cout.put(c);
}
```

The three-argument *s*.*get*(*p*,*n*,*term*) reads at at most *n-1* characters into *p*[*0*]..*p*[*n-2*]. A call of *get*( ) will always place a *0* at the end of the characters (if any) it placed in the buffer, so *p*

must point to an array of at least *n* characters. The third argument, *term*, specifies a terminator. A typical use of the three-argument *get*( ) is to read a ''line'' into a fixed-sized buffer for further analysis. For example:

```
void f()
{
    char buf[100];
    cin >> buf;                 // suspect: will overflow some day
    cin.get(buf, 100, '\n');    // safe
    // ...
}
```

If the terminator is found, it is left as the first unread character on the stream. Never call *get*( ) twice without removing the terminator. For example:

```
void subtle_infinite_loop()
{
    char buf[256];

    while (cin) {
        cin.get(buf, 256);      // read a line
        cout << buf;            // print a line. Oops: forgot to remove '\n' from cin
    }
}
```

This example is a good reason to prefer *getline*( ) over *get*( ). A *getline*( ) behaves like its corresponding *get*( ), except that it removes its terminator from the *istream*. For example:

```
void f()
{
    char word[100][MAX];
    int i = 0;
    while(cin.getline(word[i++], 100, '\n') && i<MAX);
    // ...
}
```

When efficiency isn't paramount, it is better to read into a *string* (§3.6, §20.3.15). In that way, the most common allocation and overflow problems cannot occur. However, the *get*( ), *getline*( ), and *read*( ) functions are needed to implement such higher-level facilities. The relatively messy interface is the price we pay for speed, for not having to re-scan the input to figure out what terminated the input operation, for being able to reliably limit the number of characters read, etc.

A call *read*(*p*, *n*) reads at most *n* characters into *p*[*0*]..*p*[*n-1*]. The read function does not rely on a terminator, and it doesn't put a terminating *0* into its target. Consequently, it really can read *n* characters (rather than just *n-1*). In other words, it simply reads characters and doesn't try to make its target into a C-style string.

The *ignore*( ) function reads characters like *read*( ), but it doesn't store them anywhere. Like *read*( ), it really can read *n* characters (rather than *n-1*). The default number of characters read by *ignore*( ) is *1*, so a call of *ignore*( ) without an argument means ''throw the next character away.'' Like *getline*( ), it optionally takes a terminator and removes that terminator from the input stream if it gets to it. Note that *ignore*( )'s default terminator is end-of-file.

For all of these functions, it is not immediately obvious what terminated the read – and it can be hard even to remember which function has what termination criterion. However, we can always inquire whether we reached end-of-file (§21.3.3). Also, *gcount*( ) gives the number of characters read from the stream by the most recent, unformatted input function call. For example:

```
void read_a_line(int max)
{
    // ...
    if (cin.fail()) {        // Oops: bad input format
        cin.clear();                 // clear the input flags (§21.3.3)
        cin.ignore(max, ';');        // skip to semicolon

        if (!cin) {
            // oops: we reached the end of the stream
        }
        else if (cin.gcount()==max) {
            // oops: read max characters
        }
        else {
            // found and discarded the semicolon
        }
    }
}
```

Unfortunately, if the maximum number of characters are read there is no way of knowing whether the terminator was found (as the last character).

The *get*( ) that doesn't take an argument is the *<iostream>* version of the *<cstdio> getchar*( ) (§21.8). It simply reads a character and returns the character's numeric value. In that way, it avoids making assumptions about the character type used. If there is no input character to return, *get*( ) returns a suitable ''end-of-file'' marker (that is, the stream's *traits_type*::*eof*( )) and sets the *istream* into *eof*-state (§21.3.3). For example:

```
void f(unsigned char* p)
{
    int i;
    while((i = cin.get()) && i!=EOF) {
        *p++ = i;
        // ...
    }
}
```

EOF is the value of *eof*( ) from the usual *char_traits* for *char*. *EOF* is presented in *<iostream>*. Thus, this loop could have been written *read*(*p*, *MAX_INT*), but presumably we wrote an explicit loop because we wanted to look at each character as it came in. It has been said that C's greatest strength is its ability to read a character and decide to do nothing with it – and to do this fast. It is indeed an important and underrated strength, and one that C++ aims to preserve.

The standard header *<cctype>* defines several functions that can be useful when processing input (§20.4.2). For example, an *eatwhite*( ) function that reads whitespace characters from a stream could be defined like this:

```
istream& eatwhite(istream& is)
{
    char c;
    while (is.get(c)) {
        if (!isspace(c)) {    // is c a whitespace character?
            is.putback(c);  // put c back into the input buffer
            break;
        }
    }
    return is;
}
```

The call *is*.*putback*(*c*) makes *c* be the next character read from the stream *is* (§21.6.4).

21.3.5 Input of User-Defined Types [io.in.udt]

An input operation can be defined for a user-defined type exactly as an output operation was. However, for an input operation, it is essential that the second argument be of a non-*const* reference type. For example:

```
istream& operator>>(istream& s, complex& a)
/*
    input formats for a complex ("f" indicates a floating-point number):
        f
        (f)
        (f,f)
*/
{
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(ios_base::badbit);  // set state
    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}
```

Despite the scarcity of error-handling code, this will actually handle most kinds of errors. The local variable *c* is initialized to avoid having its value accidentally be '(' after a failed first >> operation. The final check of the stream state ensures that the value of the argument *a* is changed only if everything went well.

The operation for setting a stream state is called *clear*( ) because its most common use is to reset the state of a stream to *good*( ); *ios_base*::*goodbit* is the default argument value for *ios_base*::*clear*( ) (§21.3.3).

21.3.6 Exceptions [io.except]

It is not convenient to test for errors after each I/O operation, so a common cause of error is failing to do so where it matters. In particular, output operations are typically unchecked, but they do occasionally fail.

The only function that directly changes the state of a stream is *clear*( ). Thus, an obvious way of getting notified by a state change is to ask *clear*( ) to throw an exception. The *ios_base* member *exceptions*( ) does just that:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    class failure;    // exception class (see §14.10)

    iostate exceptions( ) const;            // get exception state
    void exceptions(iostate except);        // set exception state

    // ...
};
```

For example,

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit);
```

requests that *clear*( ) should throw an *ios_base*::*failure* exception if *cout* goes into states *bad*, *fail*, or *eof* – in other words, if any output operation on *cout* doesn't perform flawlessly. Similarly,

```
cin.exceptions(ios_base::badbit|ios_base::failbit);
```

allows us to catch the not-too-uncommon case in which the input is not in the format we expected, so an input operation didn't return a value from the stream.

A call of *exceptions*( ) with no arguments returns the set of I/O state flags that triggers an exception. For example:

```
void print_exceptions(ios_base& ios)
{
    ios_base::iostate s = ios.exceptions( );
    if (s&ios_base::badbit) cout << "throws for bad";
    if (s&ios_base::failbit) cout << "throws for fail";
    if (s&ios_base::eofbit) cout << "throws for eof";
    if (s == 0) cout << "doesn´t throw";
}
```

The primary use of I/O exceptions is to catch unlikely – and therefore often forgotten – errors. Another is to control I/O. For example:

```
void readints(vector<int>& s)          // not my favorite style!
{
    ios_base::iostate old_state = cin.exceptions();  // save exception state
    cin.exceptions(ios_base::eofbit);                // throw for eof

    for (;;)
        try {
            int i;
            cin>>i;
            s.push_back(i);
        }
        catch(ios_base::eof) {
            // ok: end of file reached
        }

    cin.exceptions(old_state);        // reset exception state
}
```

The question to ask about this use of exceptions is, ‘‘Is that an error?’’ or ‘‘Is that really exceptional?’’ (§14.5). Usually, I find that the answer to either question is no. Consequently, I prefer to deal with the stream state directly. What can be handled with local control structures within a function is rarely improved by the use of exceptions.

21.3.7 Tying of Streams [io.tie]

The *basic_ios* function *tie()* is used to set up and break connections between an *istream* and an *ostream*:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
    // ...

    basic_ostream<Ch,Tr>* tie() const;                      // get pointer to tied stream
    basic_ostream<Ch,Tr>* tie(basic_ostream<Ch,Tr>* s);     // tie *this to s

    // ...
};
```

Consider:

```
string get_passwd()
{
    string s;
    cout << "Password: ";
    cin >> s;
    // ...
}
```

How can we be sure that *Password:* appears on the screen before the read operation is executed? The output on *cout* is buffered, so if *cin* and *cout* had been independent *Password:* would not have appeared on the screen until the output buffer was full. The answer is that *cout* is tied to *cin* by the operation *cin.tie(&cout)*.

When an *ostream* is tied to an *istream*, the *ostream* is flushed whenever an input operation on the *istream* causes underflow; that is, whenever new characters are needed from the ultimate input source to complete the input operation. Thus,

```
cout << "Password: ";
cin >> s;
```

is equivalent to:

```
cout << "Password: ";
cout.flush();
cin >> s;
```

A stream can have at most one *ostream* at a time tied to it. A call *s.tie(0)* unties the stream *s* from the stream it was tied to, if any. Like most other stream functions that set a value, *tie(s)* returns the previous value; that is, it returns the previously tied stream or *0*. A call without an argument, *tie()*, returns the current value without changing it.

Of the standard streams, *cout* is tied to *cin* and *wcout* is tied to *wcin*. The *cerr* streams need not be tied because they are unbuffered, while the *clog* streams are not meant for user interaction.

21.3.8 Sentries [io.sentry]

When I wrote operators << and >> for *complex*, I did not worry about tied streams (§21.3.7) or whether changing stream state would cause exceptions (§21.3.6). I assumed – correctly – that the library-provided functions would take care of that for me. But how? There are a couple of dozen such functions. If we had to write intricate code to handle tied streams, *locale*s (§21.7), exceptions, etc., in each, then the code could get rather messy.

The approach taken is to provide the common code through a *sentry* class. Code that needs to be executed first (the ''prefix code'') – such as flushing a tied stream – is provided as the *sentry*'s constructor. Code that needs to be executed last (the ''suffix code'') – such as throwing exceptions caused by state changes – is provided as the *sentry*'s destructor:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
    // ...
    class sentry;
    // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream<Ch,Tr>::sentry {
public:
    explicit sentry(basic_ostream<Ch,Tr>& s);
    ~sentry();
    operator bool();

    // ...
};
```

Thus, common code is factored out and an individual function can be written like this:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch, Tr>& basic_ostream<Ch, Tr>::operator<<(int i)
{
    sentry s(*this);
    if (!s) {   // check whether all is well for output to start
        setstate(failbit);
        return *this;
    }

    // output the int
    return *this;
}
```

This technique of using constructors and destructors to provide common prefix and suffix code through a class is useful in many contexts.

Naturally, ***basic_istream*** has a similar ***sentry*** member class.

21.4 Formatting [io.format]

The examples in §21.2 were all of what is commonly called *unformatted output*. That is, an object was turned into a sequence of characters according to default rules. Often, the programmer needs more detailed control. For example, we need to be able to control the amount of space used for an output operation and the format used for output of numbers. Similarly, some aspects of input can be explicitly controlled.

Control of I/O formatting resides in class ***basic_ios*** and its base ***ios_base***. For example, class ***basic_ios*** holds the information about the base (octal, decimal, or hexadecimal) to be used when integers are written or read, the precision of floating-point numbers written or read, etc. It also holds the functions to set and examine these per-stream control variables.

Class ***basic_ios*** is a base of ***basic_istream*** and ***basic_ostream***, so format control is on a per-stream basis.

21.4.1 Format State [io.format.state]

Formatting of I/O is controlled by a set of flags and integer values in the stream's ***ios_base***:

```
class ios_base {
public:
    // ...
    // names of format flags:

    typedef implementation_defined1 fmtflags;
    static const fmtflags
        skipws,             // skip whitespace on input

        left,               // field adjustment: pad after value
        right,              // pad before value
        internal,           // pad between sign and value
```

```
        boolalpha,          // use symbolic representation of true and false

        dec,                // integer base: base 10 output (decimal)
        hex,                // base 16 output (hexadecimal)
        oct,                // base 8 output (octal)

        scientific,         // floating-point notation: d.dddddddEdd
        fixed,              // dddd.dd

        showbase,           // on output prefix oct by 0 and hex by 0x
        showpoint,          // print trailing zeros
        showpos,            // explicit '+' for positive ints
        uppercase,          // 'E', 'X' rather than 'e', 'x'

        adjustfield,        // flags related to field adjustment (§21.4.5)
        basefield,          // flags related to integer base (§21.4.2)
        floatfield;         // flags related to floating-point output (§21.4.3)

        fmtflags unitbuf;   // flush output after each output operation

    fmtflags flags() const;         // read flags
    fmtflags flags(fmtflags f);     // set flags

    fmtflags setf(fmtflags f) { return flags(flags()|f); }                  // add flag
    fmtflags setf(fmtflags f, fmtflags mask) { return flags(flags()|(f&mask)); }// add flag
    void unsetf(fmtflags mask) { flags(flags()&~mask); }                    // clear flags

    // ...
};
```

The values of the flags are implementation-defined. Use the symbolic names exclusively, rather than specific numeric values, even if those values happen to be correct on your implementation today.

Defining an interface as a set of flags, and providing operations for setting and clearing those flags is a time-honored if somewhat old-fashioned technique. Its main virtue is that a user can compose a set of options. For example:

```
const ios_base::fmtflags my_opt = ios_base::left|ios_base::oct|ios_base::fixed;
```

This allows us to pass options around and install them where needed. For example:

```
void your_function(ios_base::fmtflags opt)
{
    ios_base::fmtflags old_options = cout.flags(opt);     // save old_options and set new ones
    // ...
    cout.flags(old_options);   // reset options
}

void my_function()
{
    your_function(my_opt);
    // ...
}
```

The *flags*( ) function returns the old option set.

Being able to read and set all options allows us to set an individual flag. For example:

```
myostream.flags(myostream.flags() | ios_base::showpos);
```

This makes *mystream* display an explicit + in front of positive numbers without affecting other options. The old options are read, and *showpos* is set by or-ing it into the set. The function *setf*( ) does exactly that, so the example could equivalently have been written:

```
myostream.setf(ios_base::showpos);
```

Once set, a flag retains its value until it is unset.

Controlling I/O options by explicitly setting and clearing flags is crude and error-prone. For simple cases, manipulators (§21.4.6) provide a cleaner interface. Using flags to control stream state is a better study in implementation technique than in interface design.

21.4.1.1 Copying Format State [io.copyfmt]

The complete format state of a stream can be copied by *copyfmt*( ):

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    basic_ios& copyfmt(const basic_ios& f);
    // ...
};
```

The stream's buffer (§21.6) and the state of that buffer isn't copied by *copyfmt*( ). However, all of the rest of the state is, including the requested exceptions (§21.3.6) and any user-supplied additions to that state (§21.7.1).

21.4.2 Integer Output [io.out.int]

The technique of or-ing in a new option with *flags*( ) or *setf*( ) works only when a single bit controls a feature. This is not the case for options such as the base used for printing integers and the style of floating-point output. For such options, the value that specifies a style is not necessarily represented by a single bit or as a set of independent single bits.

The solution adopted in *<iostream>* is to provide a version of *setf*( ) that takes a second ''pseudo argument'' that indicates which kind of option we want to set in addition to the new value. For example,

```
cout.setf(ios_base::oct, ios_base::basefield);   // octal
cout.setf(ios_base::dec, ios_base::basefield);   // decimal
cout.setf(ios_base::hex, ios_base::basefield);   // hexadecimal
```

sets the base of integers without side effects on other parts of the stream state. Once set, a base is used until reset. For example,

```
cout << 1234 << ´ ´ << 1234 << ´ ´;          //default: decimal
```

```
cout.setf(ios_base::oct, ios_base::basefield);   // octal
cout << 1234 << ´ ´ << 1234 << ´ ´;

cout.setf(ios_base::hex, ios_base::basefield);   // hexadecimal
cout << 1234 << ´ ´ << 1234 << ´ ´;
```

produces *1234 1234 2322 2322 4d2 4d2*.

If we need to be able to tell which base was used for each number, we can set ***showbase***. Thus, adding

```
cout.setf(ios_base::showbase);
```

before the previous operations, we get *1234 1234 02322 02322 0x4d2 0x4d2*. The standard manipulators (§21.4.6.2) provide a more elegant way of specifying the base of integer output.

21.4.3 Floating-Point Output [io.out.float]

Floating-point output is controlled by a *format* and a *precision*:

- The *general* format lets the implementation choose a format that presents a value in the style that best preserves the value in the space available. The precision specifies the maximum number of digits. It corresponds to ***printf*( )**'s ***%g*** (§21.8).
- The *scientific* format presents a value with one digit before a decimal point and an exponent. The precision specifies the maximum number of digits after the decimal point. It corresponds to ***printf*( )**'s ***%e***.
- The *fixed* format presents a value as an integer part followed by a decimal point and a fractional part. The precision specifies the maximum number of digits after the decimal point. It corresponds to ***printf*( )**'s ***%f***.

We control the floating-point output format through the state manipulation functions. In particular, we can set the notation used for printing floating-point values without side effects on other parts of the stream state. For example,

```
cout << "default:\t" << 1234.56789 << ´\n´;

cout.setf(ios_base::scientific, ios_base::floatfield);   // use scientific format
cout << "scientific:\t" << 1234.56789 << ´\n´;

cout.setf(ios_base::fixed, ios_base::floatfield);        // use fixed-point format
cout << "fixed:\t" << 1234.56789 << ´\n´;

cout.setf(0, ios_base::floatfield);                      // reset to default (that is, general format)
cout << "default:\t" << 1234.56789 << ´\n´;
```

produces

```
default:    1234.57
scientific: 1.234568e+03
fixed:      1234.567890
default:    1234.57
```

The default precision (for all formats) is ***6***. The precision is controlled by an ***ios_base*** member function:

```
class ios_base {
public:
    // ...
    streamsize precision() const;          // get precision
    streamsize precision(streamsize n);    // set precision (and get old precision)
    // ...
};
```

A call of *precision*( ) affects all floating-point I/O operations for a stream up until the next call of *precision*( ). Thus,

```
cout.precision(8);
cout << 1234.56789 << ´ ´ << 1234.56789 << ´ ´ << 123456 << ´\n´;

cout.precision(4);
cout << 1234.56789 << ´ ´ << 1234.56789 << ´ ´ << 123456 << ´\n´;
```

produces

```
1234.5679 1234.5679 123456
1235 1235 123456
```

Note that floating-point values are rounded rather than just truncated and that *precision*( ) doesn't affect integer output.

The *uppercase* flag (§21.4.1) determines whether *e* or *E* is used to indicate the exponents in the scientific format.

Manipulators provide a more elegant way of specifying output format for floating-point output (§21.4.6.2).

21.4.4 Output Fields [io.fields]

Often, we want to fill a specific space on an output line with text. We want to use exactly *n* characters and not fewer (and more only if the text does not fit). To do this, we specify a field width and a character to be used if padding is needed:

```
class ios_base {
public:
    // ...
    streamsize width() const;              // get field width
    streamsize width(streamsize wide);     // set field width
    // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    Ch fill() const;                       // get filler character
    Ch fill(Ch ch);                        // set filler character
    // ...
};
```

The *width*( ) function specifies the minimum number of characters to be used for the next standard library << output operation of a numeric value, *bool*, C-style string, character, pointer (§21.2.1), *string* (§20.3.15), and *bitfield* (§17.5.3.3). For example,

```
cout.width(4);
cout << 12;
```

will print *12* preceded by two spaces.

The ''padding'' or ''filler'' character can be specified by the *fill*( ) function. For example,

```
cout.width(4);
cout.fill('#');
cout << "ab";
```

gives the output *##ab*.

The default fill character is the space character and the default field size is *0*, meaning ''as many characters as needed.'' The field size can be reset to its default value like this:

```
cout.width(0); // ''as many characters as needed''
```

A call *width*(*n*) function sets the minimum number of characters to *n*. If more characters are provided, they will all be printed. For example,

```
cout.width(4);
cout << "abcdef";
```

produces *abcdef* rather than just *abcd*. It is usually better to get the right output looking ugly than to get the wrong output looking just fine (see also §21.10[21]).

A *width*(*n*) call affects only the immediately following << output operation:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

This produces *##12:13*, rather than *##12###:##13*, as would have been the case had *width*(*4*) applied to subsequent operations. Had all subsequent output operations been affected by *width*( ), we would have had to explicitly specify *width*( ) for essentially all values.

The standard manipulators (§21.4.6.2) provide a more elegant way of specifying the width of an output field.

21.4.5 Field Adjustment [io.field.adjust]

The adjustment of characters within a field can be controlled by *setf*( ) calls:

```
cout.setf(ios_base::left, ios_base::adjustfield);      // left
cout.setf(ios_base::right, ios_base::adjustfield);     // right
cout.setf(ios_base::internal, ios_base::adjustfield);  // internal
```

This sets the adjustment of output within an output field defined by *ios_base*::*width*( ) without side effects on other parts of the stream state.

Adjustment can be specified like this:

```
cout.fill('#');

cout << '(';
cout.width(4);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::left, ios_base::adjustfield);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::internal, ios_base::adjustfield);
cout << -12 << ")";
```

This produces: `(#-12), (-12#), (-#12)`. Internal adjustment places fill characters between the sign and the value. As shown, right adjustment is the default.

21.4.6 Manipulators [io.manipulators]

To save the programmer from having to deal with the state of a stream in terms of flags, the standard library provides a set of functions for manipulating that state. The key idea is to insert an operation that modifies the state in between the objects being read or written. For example, we can explicitly request that an output buffer be flushed:

```
cout << x << flush << y << flush;
```

Here, `cout.flush()` is called at the appropriate times. This is done by a version of `<<` that takes a pointer to function argument and invokes it:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
    // ...

    basic_ostream& operator<<(basic_ostream& (*f)(basic_ostream&)) { return f(*this); }
    basic_ostream& operator<<(ios_base& (*f)(ios_base&));
    basic_ostream& operator<<(basic_ios<Ch,Tr>& (*f)(basic_ios<Ch,Tr>&));

    // ...
};
```

For this to work, a function must be a nonmember or static-member function with the right type. In particular, `flush()` is defined like this:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>& s)
{
    return s.flush();    // call ostream's member flush()
}
```

These declarations ensure that

```
cout << flush;
```

is resolved as

```
cout.operator<<(flush);
```

which calls

```
flush(cout);
```

which then invokes

```
cout.flush();
```

The whole rigmarole is done (at compile time) to allow **basic_ostream::flush()** to be called using the **cout<<flush** notation.

There is a wide variety of operations we might like to perform just before or just after an input or output operation. For example:

```
cout << x;
cout.flush();
cout << y;


cin.noskipws();        // don't skip whitespace
cin >> x;
```

When the operations are written as separate statements, the logical connections between the operations are not obvious. Once the logical connection is lost, the code gets harder to understand. The notion of manipulators allows operations such as **flush()** and **noskipws()** to be inserted directly in the list of input or output operations. For example:

```
cout << x << flush << y << flush;
cin >> noskipws >> x;
```

Naturally, class **basic_istream** provides >> operators for invoking manipulators in a way similar to class **basic_ostream**:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
    // ...

    basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
    basic_istream& operator>>(basic_ios<Ch,Tr>& (*pf)(basic_ios<Ch,Tr>&));
    basic_istream& operator>>(ios_base& (*pf)(ios_base&));

    // ...
};
```

21.4.6.1 Manipulators Taking Arguments [io.manip.arg]

Manipulators that take arguments can also be useful. For example, we might want to write

```
cout << setprecision(4) << angle;
```

to print the value of the floating-point variable *angle* with four digits.

To do this, *setprecision* must return an object that is initialized by *4* and that calls *cout*::*setprecision*(*4*) when invoked. Such a manipulator is a function object that is invoked by << rather than by ( ). The exact type of that function object is implementation-defined, but it might be defined like this:

```
struct smanip {
    ios_base& (*f)(ios_base&, int);        // function to be called
    int i;

    smanip(ios_base& (*ff)(ios_base&, int), int ii) : f(ff), i(ii) { }
};

template<cladd Ch, class Tr>
ostream<Ch, Tr>& operator<<(ostream<Ch, Tr>& os, smanip& m)
{
    return m.f(os, m.i);
}
```

The *smanip* constructor stores its arguments in *f* and *i*, and *operator*<< calls *f*(*i*). We can now define *setprecision*( ) like this:

```
ios_base& set_precision(ios_base& s, int n)        // helper
{
    return s.setprecision(n);  // call the member function
}

inline smanip setprecision(int n)
{
    return smanip(set_precision, n);        // make the function object
}
```

We can now write:

```
cout << setprecision(4) << angle ;
```

A programmer can define new manipulators in the style of *smanip* as needed (§21.10[22]). Doing this does not require modification of the definitions of standard library templates and classes such as *basic_istream*, *basic_ostream*, *basic_ios*, and *ios_base*.

21.4.6.2 Standard I/O Manipulators [io.std.manipulators]

The standard library provides manipulators corresponding to the various format states and state changes. The standard manipulators are defined in namespace *std*. Manipulators taking *io_base*, *istream*, and *ostream*arguments are presented in *<ios>*, *<ostream>*, and *<iostream>*, respectively. The rest of the standard manipulators are presented in *<iomanip>*.

```
ios_base& boolalpha(ios_base&);        // symbolic representation of true and false (input and output)
ios_base& noboolalpha(ios_base& s);    // s.unsetf(ios_base::boolalpha)

ios_base& showbase(ios_base&);         // on output prefix oct by 0 and hex by 0x
ios_base& noshowbase(ios_base& s);     // s.unsetf(ios_base::showbase)

ios_base& showpoint(ios_base&);
ios_base& noshowpoint(ios_base& s);    // s.unsetf(ios_base::showpoint)

ios_base& showpos(ios_base&);
ios_base& noshowpos(ios_base& s);      // s.unsetf(ios_base::showpos)

ios_base& skipws(ios_base&);           // skip whitespace
ios_base& noskipws(ios_base& s);       // s.unsetf(ios_base::skipws)

ios_base& uppercase(ios_base&);        // X and E rather than x and e
ios_base& nouppercase(ios_base&);      // x and e rather than X and E

ios_base& internal(ios_base&);         // adjust §21.4.5
ios_base& left(ios_base&);             // pad after value
ios_base& right(ios_base&);            // pad before value

ios_base& dec(ios_base&);              // integer base is 10 (§21.4.2)
ios_base& hex(ios_base&);              // integer base is 16
ios_base& oct(ios_base&);              // integer base is 8

ios_base& fixed(ios_base&);            // floating-point format dddd.dd (§21.4.3)
ios_base& scientific(ios_base&);       // scientific format d.ddddEdd

template <class Ch, class Tr>
    basic_ostream<Ch,Tr>& endl(basic_ostream<Ch,Tr>&);    // put '\n' and flush
template <class Ch, class Tr>
    basic_ostream<Ch,Tr>& ends(basic_ostream<Ch,Tr>&);    // put '\0' and flush
template <class Ch, class Tr>
    basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>&);   // flush stream

template <class Ch, class Tr>
    basic_istream<Ch,Tr>& ws(basic_istream<Ch,Tr>&);      // eat whitespace

smanip resetiosflags(ios_base::fmtflags f);     // clear flags (§21.4)
smanip setiosflags(ios_base::fmtflags f);       // set flags (§21.4)
smanip setbase(int b);                          // output integers in base b
smanip setfill(int c);                          // make c the fill character
smanip setprecision(int n);                     // n digits after decimal point
smanip setw(int n);                             // next field is n char
```

For example,

```
cout << 1234 << ´,´ << hex << 1234 << ´,´ << oct << 1234 << endl;
```

produces `1234,4d2,2322` and

```
cout << ´(´ << setw(4) << setfill(´#´) << 12 << ") (" << 12 << ")\n";
```

produces `(##12) (12)`.

When using manipulators that do not take arguments, *do not* add parentheses. When using standard manipulators that take arguments, remember to ***#include <iomanip>***. For example:

```
#include <iostream>

int main()
{
    std::cout << setprecision(4)    // error: setprecision undefined (forgot <iomanip>)
        << scientific()             // error: ostream<<ostream& (spurious parentheses)
        << d << endl;
}
```

21.4.6.3 User-Defined Manipulators [io.ud.manipulators]

A programmer can add manipulators in the style of the standard ones. Here, I present an additional style that I have found useful for formatting floating-point numbers.

The ***precision*** used persists for all output operations, but a ***width()*** operation applies to the next numeric output operation only. What I want is something that makes it simple to output a floating-point number in a predefined format without affecting future output operations on the stream. The basic idea is to define a class that represents formats, another that represents a format plus a value to be formatted, and then an operator **<<** that outputs the value to an ***ostream*** according to the format. For example:

```
Form gen4(4); // general format, precision is 4

void f(double d)
{
    Form sci8 = gen4;
    sci8.scientific().precision(8); // scientific format, precision 8

    cout << d << ´ ´ << gen4(d) << ´ ´ << sci8(d) << ´ ´ << d << ´\n´;
}
```

A call *f(1234.56789)* writes

```
1234.57 1235 1.23456789e+03 1234.57
```

Note how the use of a ***Form*** doesn't affect the state of the stream so that the last output of ***d*** has the same default format as the first.

Here is a simplified implementation:

```
class Bound_form;    // Form plus value

class Form {
    friend ostream& operator<<(ostream&, const Bound_form&);

    int prc;    // precision
    int wdt;    // width, 0 means as wide as necessary
    int fmt;    // general, scientific, or fixed (§21.4.3)
    // ...
```

```
public:
    explicit Form(int p = 6) : prc(p)    // default precision is 6
    {
        fmt = 0;    // general format (§21.4.3)
        wdt = 0;    // as wide as necessary
    }

    Bound_form operator()(double d) const; // make a Bound_form for *this and d

    Form& scientific() { fmt = ios_base::scientific; return *this; }
    Form& fixed() { fmt = ios_base::fixed; return *this; }
    Form& general() { fmt = 0; return *this; }

    Form& uppercase();
    Form& lowercase();
    Form& precision(int p) { prc = p; return *this; }

    Form& width(int w) { wdt = w; return *this; }        // applies to all types
    Form& fill(char);

    Form& plus(bool b = true);                           // explicit plus
    Form& trailing_zeros(bool b = true);                 // print trailing zeros
    // ...
};
```

The idea is that a ***Form*** holds all the information needed to format one data item. The default is chosen to be reasonable for many uses, and the various member functions can be used to reset individual aspects of formatting. The `()` operator is used to bind a value with the format to be used to output it. A ***Bound_form*** can then be output to a given stream by a suitable `<<` function:

```
struct Bound_form {
    const Form& f;
    double val;

    Bound_form(const Form& ff, double v) : f(ff), val(v) { }
};

Bound_form Form::operator()(double d) { return Bound_form(*this,d); }

ostream& operator<<(ostream& os, const Bound_form& bf)
{
    ostringstream s;            // string streams are described in §21.5.3
    s.precision(bf.f.prc);
    s.setf(bf.f.fmt,ios_base::floatfield);
    s << bf.val;                // compose string in s
    return os << s.str();       // output s to os
}
```

Writing a less simplistic implementation of `<<` is left as an exercise (§21.10[21]). The ***Form*** and ***Bound_form*** classes are easily extended for formatting integers, strings, etc. (see §21.10[20]).

Note that these declarations make the combination of `<<` and `()` into a ternary operator; *cout<<sci4(d)* collects the ***ostream***, the format, and the value into a single function before doing any real computation.

21.5 File Streams and String Streams [io.files]

When a C++ program starts, ***cout***, ***cerr***, ***clog***, ***cin***, and their wide-character equivalents (§21.2.1) are available for use. These streams are set up by default and their correspondence with I/O devices or files is determined by ''the system.'' In addition, you can create your own streams. In this case, you must specify to what the streams are attached. Attaching a stream to a file or to a ***string*** is common enough so as to be supported directly by the standard library. Here is the hierarchy of standard stream classes:

The classes suffixed by **<>** are templates parameterized on the character type, and their names have a ***basic_*** prefix. A dotted line indicates a virtual base class (§15.2.4).

Files and strings are examples of containers that you can both read from and write to. Consequently, you can have a stream that supports both **<<** and **>>**. Such a stream is called an ***iostream***, which is defined in namespace ***std*** and presented in ***<iostream>***:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_iostream : public basic_istream<Ch,Tr>, public basic_ostream<Ch,Tr> {
public:
    explicit basic_iostream(basic_streambuf<Ch,Tr>* sb);
    virtual ~basic_iostream();
};

typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

Reading and writing from an ***iostream*** is controlled through the put-buffer and get-buffer operations on the ***iostream***'s ***streambuf*** (§21.6.4).

21.5.1 File Streams [io.filestream]

Here is a complete program that copies one file to another. The file names are taken as command-line arguments:

```
#include <fstream>
#include <cstdlib>
```

```
void error(const char* p, const char* p2 = "")
{
    cerr << p << ´ ´ << p2 << ´\n´;
    std::exit(1);
}

int main(int argc, char* argv[])
{
    if (argc != 3) error("wrong number of arguments");

    std::ifstream from(argv[1]);                    // open input file stream
    if (!from) error("cannot open input file",argv[1]);

    std::ofstream to(argv[2]);                      // open output file stream
    if (!to) error("cannot open output file",argv[2]);

    char ch;
    while (from.get(ch)) to.put(ch);

    if (!from.eof() || !to) error("something strange happened");
}
```

A file is opened for input by creating an object of class ***ifstream*** (input file stream) with the file name as the argument. Similarly, a file is opened for output by creating an object of class ***ofstream*** (output file stream) with the file name as the argument. In both cases, we test the state of the created object to see if the file was successfully opened.

A ***basic_ofstream*** is declared like this in ***<fstream>***:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ofstream : public basic_ostream<Ch,Tr> {
public:
    basic_ofstream();
    explicit basic_ofstream(const char* p, openmode m = out);

    basic_filebuf<Ch,Tr>* rdbuf() const;

    bool is_open() const;
    void open(const char* p, openmode m = out);
    void close();
};
```

As usual, ***typedef***s are available for the most common types:

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;

typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<wchar_t> wfstream;
```

An ***ifstream*** is like an ***ofstream***, except that it is derived from ***istream*** and is by default opened for reading. In addition, the standard library offers an ***fstream***, which is like an ***ofstream***, except that it is derived from ***iostream*** and by default can be both read from and written to.

File stream constructors take a second argument specifying alternative modes of opening:

```
class ios_base {
public:
    // ...

    typedef implementation_defined3 openmode;
    static openmode app,      // append
                    ate,      // open and seek to end of file (pronounced ''at end'')
                    binary,   // I/O to be done in binary mode (rather than text mode)
                    in,       // open for reading
                    out,      // open for writing
                    trunc;    // truncate file to 0-length

    // ...
};
```

The actual values of *openmode*s and their meanings are implementation-defined. Please consult your systems and library manual for details – and do experiment. The comments should give some idea of the intended meaning of the modes. For example, we can open a file so that anything written to it is appended to the end:

```
ofstream mystream(name.c_str(), ios_base::app);
```

It is also possible to open a file for both input and output. For example:

```
fstream dictionary("concordance", ios_base::in|ios_base::out);
```

21.5.2 Closing of Streams [io.close]

A file can be explicitly closed by calling *close*( ) on its stream:

```
void f(ostream& mystream)
{
    // ...

    mystream.close();
}
```

However, this is implicitly done by the stream's destructor. So an explicit call of *close*( ) is needed only if the file must be closed before reaching the end of the scope in which its stream was declared.

This raises the question of how an implementation can ensure that the predefined streams *cout*, *cin*, *cerr*, and *clog* are created before their first use and closed (only) after their last use. Naturally, different implementations of the *<iostream>* stream library can use different techniques to achieve this. After all, exactly how it is done is an implementation detail that should not be visible to the user. Here, I present just one technique that is general enough to be used to ensure proper order of construction and destruction of global objects of a variety of types. An implementation may be able to do better by taking advantage of special features of a compiler or linker.

The fundamental idea is to define a helper class that is a counter that keeps track of how many times *<iostream>* has been included in a separately compiled source file:

```
class ios_base::Init {
    static int count;
public:
    Init();
    ~Init();
};

namespace {                         // in <iostream>, one copy in each file #including <iostream>
    ios_base::Init __ioinit;
}

int ios_base::Init::count = 0;      // in some .c file
```

Each translation unit (§9.1) declares its own object called ***__ioinit***. The constructor for the ***__ioinit*** objects uses ***ios_base::Init::count*** as a first-time switch to ensure that actual initialization of the global objects of the stream I/O library is done exactly once:

```
ios_base::Init::Init()
{
    if (count++ == 0) { /* initialize cout, cerr, cin, etc. */ }
}
```

Conversely, the destructor for the ***__ioinit*** objects uses ***ios_base::Init::count*** as a last-time switch to ensure that the streams are closed:

```
ios_base::Init::~Init()
{
    if (--count == 0) { /* clean up cout (flush, etc.), cerr, cin, etc. */ }
}
```

This is a general technique for dealing with libraries that require initialization and cleanup of global objects. In a system in which all code resides in main memory during execution, the technique is almost free. When that is not the case, the overhead of bringing each object file into main memory to execute its initialization function can be noticeable. When possible, it is better to avoid global objects. For a class in which each operation performs significant work, it can be reasonable to test a first-time switch (like ***ios_base::Init::count***) in each operation to ensure initialization. However, that approach would have been prohibitively expensive for streams. The overhead of a first-time switch in the functions that read and write single characters would have been quite noticeable.

21.5.3 String Streams [io.stringstream]

A stream can be attached to a ***string***. That is, we can read from a ***string*** and write to a ***string*** using the formatting facilities provided by streams. Such streams are called a ***stringstream***s. They are defined in ***<sstream>***:

```
template <class Ch, class Tr=char_traits<Ch> >
class basic_stringstream : public basic_iostream<Ch,Tr> {
public:
    explicit basic_stringstream(ios_base::openmode m = out|in);
    explicit basic_stringstream(const basic_string<Ch>& s, openmode m = out|in);
```

```
        basic_string<Ch> str() const;                // get copy of string
        void str(const basic_string<Ch>& s);         // set value to copy of s

        basic_stringbuf<Ch,Tr>* rdbuf() const;
};

typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;

typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

For example, an ***ostringstream*** can be used to format message ***strings***:

```
extern const char* std_message[];

string compose(int n, const string& cs)
{
        ostringstream ost;
        ost << "error(" << n << ") " << std_message[n] << " (user comment: " << cs << ´)´;
        return ost.str();
}
```

There is no need to check for overflow because ***ost*** is expanded as needed. This technique can be most useful for coping with cases in which the formatting required is more complicated than what is common for a line-oriented output device.

An initial value can be provided for an ***ostringstream***, so we could equivalently have written:

```
string compose2(int n, const string& cs)
{
        ostringstream ost("error(");
        ost << n << ") " << std_message[n] << " (user comment: " << cs << ´)´;
        return ost.str();
}
```

An ***istringstream*** is an input stream reading from a ***string***:

```
#include <sstream>

void word_per_line(const string& s)  // prints one word per line
{
        istringstream ist(s);
        string w;
        while (ist>>w) cout << w << ´\n´;
}

int main()
{
        word_per_line("If you think C++ is difficult, try English");
}
```

The initializer ***string*** is copied into the ***istringstream***. The end of the string terminates input.

It is possible to define streams that directly read from and write to arrays of characters (§21.10[26]). This is often useful when dealing with older code, especially since the ***ostrstream*** and ***istrstream*** classes doing that were part of the original streams library.

21.6 Buffering [io.buf]

Conceptually, an output stream puts characters into a buffer. Some time later, the characters are then written to wherever they are supposed to go. Such a buffer is called a ***streambuf*** (§21.6.4). Its definition is found in ***<streambuf>***. Different types of ***streambuf***s implement different buffering strategies. Typically, the ***streambuf*** stores characters in an array until an overflow forces it to write the characters to their real destination. Thus, an ***ostream*** can be represented graphically like this:

The set of template arguments for an ***ostream*** and its ***streambuf*** must be the same and determines the type of character used in the character buffer.

An ***istream*** is similar, except that the characters flow the other way.

Unbuffered I/O is simply I/O where the streambuf immediately transfers each character, rather than holding on to characters until enough have been gathered for efficient transfer.

21.6.1 Output Streams and Buffers [io.ostreambuf]

An ***ostream*** provides operations for converting values of various types into character sequences according to conventions (§21.2.1) and explicit formatting directives (§21.4). In addition, an ***ostream*** provides operations that deal directly with its ***streambuf***:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
    // ...
    explicit basic_ostream(basic_streambuf<Ch,Tr>* b);

    pos_type tellp();                                  // get current position
    basic_ostream& seekp(pos_type);                    // set current position
    basic_ostream& seekp(off_type, ios_base::seekdir); // set current position

    basic_ostream& flush();          // empty buffer (to real destination)
```

```
        basic_ostream& operator<<(basic_streambuf<Ch,Tr>* b);  // write from b
    };
```

An *ostream* is constructed with a *streambuf* argument, which determines how the characters written are handled and where they eventually go. For example, an *ostringstream* (§21.5.3) or an *ofstream* (§21.5.1) are created by initializing an *ostream* with a suitable *streambuf* (§21.6.4).

The *seekp*( ) functions are used to position an *ostream* for writing. The *p* suffix indicates that it is the position used for *putting* characters into the stream. These functions have no effect unless the stream is attached to something for which positioning is meaningful, such as a file. The *pos_type* represents a character position in a file, and the *off_type* represents an offset from a point indicated by an *ios_base*::*seekdir*:

```
    class ios_base {
        // ...

        typedef implementation_defined4 seekdir;
        static const seekdir beg,  // seek from beginning of current file
                             cur,  // seek from current position
                             end;  // seek backwards from end of current file

        // ...
    };
```

Stream positions start at *0*, so we can think of a file as an array of *n* characters. For example:

```
    int f(ofstream& fout)
    {
        fout.seekp(10);
        fout << '#';
        fout.seekp(-2,ios_base::cur);
        fout << '*';
    }
```

This places a # into *file*[*10*] and a * in *file*[*8*]. There is no similar way to do random access on elements of a plain *istream* or *ostream* (see §21.10[13]).

The *flush*( ) operation allows the user to empty the buffer without waiting for an overflow.

It is possible to use << to write a *streambuf* directly into an *ostream*. This is primarily handy for implementers of I/O mechanisms.

21.6.2 Input Streams and Buffers [io.istreambuf]

An *istream* provides operations for reading characters and converting them into values of various types (§21.3.1). In addition, an *istream* provides operations that deal directly with its *streambuf*:

```
    template <class Ch, class Tr = char_traits<Ch> >
    class basic_istream : virtual public basic_ios<Ch,Tr> {
    public:
        // ...

        explicit basic_istream(basic_streambuf<Ch,Tr>* b);
```

```
        pos_type tellg();                                   // get current position
        basic_istream& seekg(pos_type);                     // set current position
        basic_istream& seekg(off_type, ios_base::seekdir);  // set current position

        basic_istream& putback(Ch c);  // put c back into the buffer
        basic_istream& unget();        // putback most recent char read
        int_type peek();               // look at next character to be read

        int sync();      // clear buffer (flush input)

        basic_istream& operator>>(basic_streambuf<Ch,Tr>* b);  // read into b
        basic_istream& get(basic_streambuf<Ch,Tr>& b, Ch t = Tr::newline());

        streamsize readsome(Ch* p, streamsize n);      // read at most n char
    };
```

The positioning functions work like their ***ostream*** counterparts (§21.6.1). The ***g*** suffix indicates that it is the position used for *getting* characters from the stream. The ***p*** and ***g*** suffixes are needed because we can create an ***iostream*** derived from both ***istream*** and ***ostream*** and such a stream needs to keep track of both a get position and a put position.

The ***putback*()** function allows a program to put an unwanted character back to be read some other time, as shown in §21.3.5. The ***unget*()** function puts the most recently read character back. Unfortunately, backing up an input stream is not always possible. For example, trying to back up past the first character read will set ***ios_base*::*failbit***. What is guaranteed is that you can back up one character after a successful read. The ***peek*()** reads the next character but leaves it in the ***streambuf*** so that it can be read again. Thus, ***c=peek*()** is equivalent to **(*c=get*(), *unget*(), *c*)** and to **(*putback*(*c=get*()), *c*)**. Note that setting ***failbit*** might trigger an exception (§21.3.6).

Flushing an ***istream*** is done using ***sync*()**. This cannot always be done right. For some kinds of streams, we would have to reread characters from the real source – and that is not always possible or desirable. Consequently, ***sync*()** returns ***0*** if it succeeded. If it failed, it sets ***ios_base*::*badbit*** (§21.3.3) and returns ***-1***. Again, setting ***badbit*** might trigger an exception (§21.3.6).

The >> and ***get*()** operations that target a ***streambuf*** are primarily useful for implementers of I/O facilities. Only such implementers should manipulate ***streambuf***s directly.

The ***readsome*()** function is a low-level operation that allows a user to peek at a stream to see if there are any characters available to read. This can be most useful when it is undesirable to wait for input, say, from a keyboard. See also ***in_avail*()** (§21.6.4).

21.6.3 Streams and Buffers [io.rdbuf]

The connection between a stream and its buffer is maintained in the stream's ***basic_ios***:

```
    template <class Ch, class Tr = char_traits<Ch> >
    class basic_ios : public ios_base {
    public:
        // ...

        basic_streambuf<charT,traits>* rdbuf() const;                  // get buffer
        basic_streambuf<charT,traits>* rdbuf(basic_streambuf<Ch,Tr>* b);  // set buffer
```

```
        locale imbue(const locale& loc);            // set locale (and get old locale)

        char narrow(char_type c, char d) const;     // make char value from char_type c
        char_type widen(char c) const;              // make char_type value from char c

        // ...

    protected:
        basic_ios();
        void init(basic_streambuf<Ch,Tr>* b);       // set initial buffer
    };
```

In addition to reading and setting the stream's ***streambuf*** (§21.6.4), ***basic_ios*** provides ***imbue*()** to read and re-set the stream's locale (§21.7) by calling ***imbue*()** on its ***ios_base*** (§21.7.1) and ***pubimbue*()** on its buffer (§21.6.4).

The ***narrow*()** and ***widen*()** functions are used to convert ***char***s to and from a buffer's ***char_type***. The second argument of ***narrow*(*c*,*d*)** is the ***char*** returned if there isn't a ***char*** corresponding to the ***char_type*** value ***c***.

21.6.4 Stream Buffers [io.streambuf]

The I/O operations are specified without any mention of file types, but not all devices can be treated identically with respect to buffering strategies. For example, an ***ostream*** bound to a ***string*** (§21.5.3) needs a different kind of buffer than does an ***ostream*** bound to a file (§21.5.1). These problems are handled by providing different buffer types for different streams at the time of initialization. There is only one set of operations on these buffer types, so the ***ostream*** functions do not contain code distinguishing them. The different types of buffers are derived from class ***streambuf***. Class ***streambuf*** provides virtual functions for operations where buffering strategies differ, such as the functions that handle overflow and underflow.

The ***basic_streambuf*** class provides two interfaces. The public interface is aimed primarily at implementers of stream classes such as ***istream***, ***ostream***, ***fstream***, ***stringstream***, etc. In addition, a protected interface is provided for implementers of new buffering strategies and of ***streambuf***s for new input sources and output destinations.

To understand a ***streambuf***, it is useful first to consider the underlying model of a buffer area provided by the protected interface. Assume that the ***streambuf*** has a *put area* into which << writes, and a *get area* from which >> reads. Each area is described by a beginning pointer, current pointer, and one-past-the-end pointer. These pointers are made available through functions:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:

    Ch* eback() const;          // start of get-buffer
    Ch* gptr() const;           // next filled character (next char read comes from here)
    Ch* egptr() const;          // one-past-end of get-buffer

    void gbump(int n);          // add n to gptr()
    void setg(Ch* begin, Ch* next, Ch* end); // set eback(), gptr(), and egptr()
```

```
        Ch* pbase() const;          // start of put-buffer
        Ch* pptr() const;           // next free char (next char written goes here)
        Ch* epptr() const;          // one-past-end of put-buffer
        void pbump(int n);          // add n to pptr()
        void setp(Ch* begin, Ch* end);      // set pbase() and pptr() to begin, and epptr() to end
        // ...
};
```

Given an array of characters, *setg*( ) and *setp*( ) can set up the pointers appropriately. An implementation might access its get area like this:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_streambuf<Ch,Tr>::int_type basic_streambuf<Ch,Tr>::snextc() // read next character
{
        if (gptr()==0) return uflow();              // no input buffering
        gbump(1);                                   // move to next character
        if (gptr()>=egptr()) return underflow(); // re-fill buffer
        return *gptr();                             // return the now current character
}
```

The public interface of a *streambuf* looks like this:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
public:
        // usual typedefs (§21.2.1)

        basic_streambuf();
        virtual ~basic_streambuf();

        locale pubimbue(const locale &loc);                 // set locale (and get old locale)
        locale getloc() const;                              // get locale

        basic_streambuf* pubsetbuf(Ch* p, streamsize n);    // set buffer space

        // position (§21.6.1):
        pos_type pubseekoff(off_type off, ios_base::seekdir way,
                            ios_base::openmode m = ios_base::in|ios_base::out);
        pos_type pubseekpos(pos_type p, ios_base::openmode m = ios_base::in|ios_base::out);

        int pubsync();                                      // sync() input (§21.6.2)


        int_type    snextc();                       // get next character
        int_type    sbumpc();                       // advance gptr() by 1
        int_type    sgetc();                        // get current char
        streamsize  sgetn(Ch* p, streamsize n);     // get into p[0]..p[n-1]

        int_type    sputbackc(Ch c);        // put c back into buffer (§21.6.2)
        int_type    sungetc();              // unget last char

        int_type    sputc(Ch c);                        // put c
        streamsize  sputn(const Ch* p, streamsize n);   // put p[0]..p[n-1]

        streamsize in_avail();              // is input ready?
```

```
        // ...
};
```

The public interface contains functions for inserting characters into the buffer and extracting characters from the buffer. These functions are simple and easily inlined. This is crucial for efficiency.

Functions that implement parts of a specific buffering strategy invoke corresponding functions in the protected interface. For example, ***pubsetbuf*( )** calls ***setbuf*( )**, which is overridden by a derived class to implement that class' notion of getting memory for the buffered characters. Using two functions to implement an operation such as ***setbuf*( )** allows an ***iostream*** implementer to do some ''housekeeping'' before and after the user's code. For example, an implementer might wrap a *try-block* around the call of the virtual function and catch exceptions thrown by the user code. This use of a pair of public and protected functions is yet another general technique that just happens to be useful in the context of I/O.

By default, ***setbuf*(0,0)** means ''unbuffered'' and ***setbuf*(p,n)** means use **p[0]..p[n-1]** to hold buffered characters.

A call to ***in_avail*( )** is used to see how many characters are available in the buffer. This can be used to avoid waiting for input. When reading from a stream connected to a keyboard, ***cin.get*(c)** might wait until the user comes back from lunch. On some systems and for some applications, it can be worthwhile taking that into account when reading. For example:

```
if (cin.rdbuf().in_avail()) {   // get() will not block
        cin.get(c);
        // do something
}
else {                          // get() might block
        // do something else
}
```

In addition to the public interface used by ***basic_istream*** and ***basic_ostream***, ***basic_streambuf*** offers a protected interface to implementers of ***streambuf***s. This is where the virtual functions that determine policy are declared:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:
        // ...

        virtual void imbue(const locale &loc);                  // set locale

        virtual basic_streambuf* setbuf(Ch* p, streamsize n);

        virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                        ios_base::openmode m = ios_base::in|ios_base::out);
        virtual pos_type seekpos(pos_type p,
                        ios_base::openmode m = ios_base::in|ios_base::out);

        virtual int sync();                                     // sync() input (§21.6.2)
```

```
        virtual int showmanyc();
        virtual streamsize xsgetn(Ch* p, streamsize n);          // get n chars
        virtual int_type underflow();                            // get area empty
        virtual int_type uflow();

        virtual int_type pbackfail(int_type c = Tr::eof());      // putback failed

        virtual streamsize xsputn(const Ch* p, streamsize n);    // put n chars
        virtual int_type overflow(int_type c = Tr::eof());       // put area full
    };
```

The ***underflow*()** and ***uflow*()** functions are called to get the next character from the real input source when the buffer is empty. If no more input is available from that source, the stream is set into ***eof*** state (§21.3.3). If doing that doesn't cause an exception, ***traits_type*::*eof*()** is returned. Unbuffered input uses ***uflow*()**; buffered input uses ***underflow*()**. Remember that there typically are more buffers in your system than the ones introduced by the ***iostream*** library, so you can suffer buffering delays even when using unbuffered stream I/O.

The ***overflow*()** function is called to transfer characters to the real output destination when the buffer is full. A call ***overflow*(*c*)** outputs the contents of the buffer plus the character ***c***. If no more output can be written to that target, the stream is put into ***eof*** state (§21.3.3). If doing that doesn't cause an exception, ***traits_type*::*eof*()** is returned.

The ***showmanyc*()** – ''show how many characters'' – function is an odd function intended to allow a user to learn something about the state of a machine's input system. It returns an estimate of how many characters can be read ''soon,'' say, by emptying the operating system's buffers rather than waiting for a disc read. A call to ***showmanyc*()** returns ***-1*** if it cannot promise that any character can be read without encountering end-of-file. This is (necessarily) rather low-level and highly implementation-dependent. Don't use ***showmanyc*()** without a careful reading of your system documentation and a few experiments.

By default, every stream gets the global locale (§21.7). A ***pubimbue*(*loc*)** or ***imbue*(*loc*)** call makes a stream use ***loc*** as its locale.

A ***streambuf*** for a particular kind of stream is derived from ***basic_streambuf***. It provides the constructors and initialization functions that connect the ***streambuf*** to a real source of (target for) characters and overrides the virtual functions that determine the buffering strategy. For example:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_filebuf : public basic_streambuf<Ch, Tr> {
public:
        basic_filebuf();
        virtual ~basic_filebuf();

        bool is_open() const;
        basic_filebuf* open(const char* p, ios_base::openmode mode);
        basic_filebuf* close();

protected:
        virtual int showmanyc();
        virtual int_type underflow();
        virtual int_type uflow();
```

```
        virtual int_type pbackfail(int_type c = Tr::eof());
        virtual int_type overflow(int_type c = Tr::eof());

        virtual basic_streambuf<Ch,Tr>* setbuf(Ch* p, streamsize n);
        virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                    ios_base::openmode m = ios_base::in|ios_base::out);
        virtual pos_type seekpos(pos_type p,
                    ios_base::openmode m = ios_base::in|ios_base::out);
        virtual int sync();
        virtual void imbue(const locale& loc);
    };
```

The functions for manipulating buffers, etc., are inherited unchanged from ***basic_streambuf***. Only functions that affect initialization and buffering policy need to be separately provided.

As usual, the obvious ***typedef***s and their wide stream counterparts are provided:

```
typedef basic_streambuf<char> streambuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_filebuf<char> filebuf;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<wchar_t> wfilebuf;
```

21.7 Locale [io.locale]

A ***locale*** is an object that controls the classification of characters into letters, digits, etc.; the collation order of strings; and the appearance of numeric values on input and output. Most commonly a ***locale*** is used implicitly by the ***iostream***s library to ensure that the usual conventions for some natural language or culture is adhered to. In such cases, a programmer never sees a ***locale*** object. However, by changing a ***stream***'s ***locale***, a programmer can change the way the stream behaves to suit a different set of conventions

A locale is an object of class ***locale*** defined in namespace ***std*** presented in ***<locale>***:

```
class locale {
public:
        // ...

        locale() throw();                          // copy of current global locale
        explicit locale(const char* name);         // construct locale using C locale name
        basic_string<char> name() const;           // give name of this locale

        locale(const locale&) throw();                     // copy locale
        const locale& operator=(const locale& ) throw();   // copy locale

        static locale global(const locale&);       // set the global locale (get the previous locale)
        static const locale& classic();            // get the locale that C defines
};
```

Here, I omitted all of the interesting pieces and left only what is needed to switch from one existing locale to another. For example:

```
void f()
{
    std::locale loc("POSIX");          // standard locale for POSIX

    cin.imbue(loc);                    // let cin use loc
    // ...
    cin.imbue(std::locale::global());  // reset cin to use the default locale
}
```

The *imbue*( ) function is a member of *basic_ios* (§21.7.1).

As shown, some fairly standard locales have character string names. These tend to be shared with C.

It is possible to set the *locale* that is used by all newly constructed streams:

```
void g(const locale& loc = locale())        // use current global locale by default
{
    locale old_global = locale::global(loc);  // make loc the default locale
    // ...
}
```

Setting the global *locale* does not change the behavior of existing streams that are using the previous value of the global *locale*. In particular, *cin*, *cout*, etc., are not affected. If they should be changed, they must be explicitly *imbue*( )d.

Imbuing a stream with a *locale* changes facets of its behavior. It is possible to use members of a *locale* directly, to define new *locale*s, and to extend *locale*s with new facets. For example, a *locale* can also be used explicitly to control the appearance of monetary units, dates, etc., on input and output (§21.10[25]) and conversion between codesets. However, discussion of that is beyond the scope of this book. Please consult your implementation's documentation.

The C-style locale is presented in <*clocale*> and <*locale.h*>.

21.7.1 Stream Callbacks [io.callbacks]

Sometimes, people want to add to the state of a stream. For example, one might want a stream to ''know'' whether a *complex* should be output in polar or Cartesian coordinates. Class *ios_base* provides a function *xalloc*( ) to allocate space for such simple state information. The value returned by *xalloc*( ) identifies a pair of locations that can be accessed by *iword*( ) and *pword*( ):

```
class ios_base {
public:
    // ...

    ~ios_base();

    locale imbue(const locale& loc);     // get and set locale
    locale getloc() const;               // get locale

    static int xalloc();         // get an integer and a pointer (both initialized to 0)
    long& iword(int i);          // access the integer iword(i)
    void*& pword(int i);         // access the pointer pword(i)
```

```
        // callbacks:

        enum event { erase_event, imbue_event, copyfmt_event }; // event type

        typedef void (*event_callback)(event, ios_base&, int i);
        void register_callback(event_callback f, int i);          // attach f to word(i)
    };
```

Sometimes, an implementer or a user needs to be notified about a change in a stream's state. The ***register_callback*( )** function ''registers'' a function to be called when its ''event'' occurs. Thus, a call of ***imbue*( )**, ***copyfmt*( )**, or ***~ios_base*( )** will call a function ''registered'' for an ***imbue_event***, ***copyfmt_event***, or ***erase_event***, respectively. When the the state changes, registered functions are called with the argument ***i*** supplied by their ***register_callback*( )**.

This storage and callback mechanism is fairly obscure. Use it only when you absolutely need to extend the low-level formatting facilities.

21.8 C Input/Output [io.c]

Because C++ and C code are often intermixed, C++ stream I/O is sometimes mixed with the C ***printf*( )** family of I/O functions. The C-style I/O functions are presented by ***<cstdio>*** and ***<stdio.h>***. Also, because C functions can be called from C++ some programmers may prefer to use the more familiar C I/O functions. Even if you prefer stream I/O, you will undoubtedly encounter C-style I/O at some time.

C and C++ I/O can be mixed on a per-character basis. A call of ***sync_with_stdio*( )** before the first stream I/O operation in the execution of a program guarantees that the C-style and C++-style I/O operations share buffers. A call of ***sync_with_stdio*(*false*)** before the first stream I/O operation prevents buffer sharing and can improve I/O performance on some implementations.

```
class ios_base {
    // ...
    static bool sync_with_stdio(bool sync = true);   // get and set
};
```

The general advantage of the stream output functions over the C standard library function ***printf*( )** is that the stream functions are type safe and have a common style for specifying output of objects of built-in and user-defined types.

The general C output functions

```
int printf(const char* format ...);              // write to stdout
int fprintf(FILE*, const char* format ...);      // write to ''file'' (stdout, stderr)
int sprintf(char* p, const char* format ...);    // write to p[0]..
```

produce formatted output of an arbitrary sequence of arguments under control of the format string ***format***. The format string contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next argument. Each conversion specification is introduced by the character **%**. For example:

```
printf("there were %d members present.", no_of_members);
```

Here %*d* specifies that *no_of_members* is to be treated as an *int* and printed as the appropriate sequence of decimal digits. With *no_of_members==127*, the output is

there were 127 members present.

The set of conversion specifications is quite large and provides a great degree of flexibility. Following the %, there may be:

- `-` an optional minus sign that specifies left-adjustment of the converted value in the field;
- `+` an optional plus sign that specifies that a value of a signed type will always begin with a `+` or `-` sign;
- `#` an optional # that specifies that floating-point values will be printed with a decimal point even if no nonzero digits follow, that trailing zeroes will be printed, that octal values will be printed with an initial *0*, and that hexadecimal values will be printed with an initial *0x* or *0X*;
- *d* an optional digit string specifying a field width; if the converted value has fewer characters than the field width, it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- `.` an optional period that serves to separate the field width from the next digit string;
- *d* an optional digit string specifying a precision that specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- `*` a field width or precision may be `*` instead of a digit string. In this case an integer argument supplies the field width or precision;
- `h` an optional character *h*, specifying that a following *d*, *o*, *x*, or *u* corresponds to a short integer argument;
- `l` an optional character *l*, specifying that a following *d*, *o*, *x*, or *u* corresponds to a long integer argument;
- `%` indicating that the character % is to be printed; no argument is used;
- *c* a character that indicates the type of conversion to be applied. The conversion characters and their meanings are:
  - `d` The integer argument is converted to decimal notation;
  - `o` The integer argument is converted to octal notation;
  - `x` The integer argument is converted to hexadecimal notation with an initial *0x*;
  - `X` The integer argument is converted to hexadecimal notation with an initial *0X*;
  - `f` The *float* or *double* argument is converted to decimal notation in the style *[-]ddd.ddd*. The number of *d*'s after the decimal point is equal to the precision for the argument. If necessary, the number is rounded. If the precision is missing, six digits are given; if the precision is explicitly *0* and # isn't specified, no decimal point is printed;
  - `e` The *float* or *double* argument is converted to decimal notation in the scientific style *[-]d.ddde+dd* or *[-]d.ddde-dd*, where there is one digit before the decimal point and the number of digits after the decimal point is equal to the precision specification for the argument. If necessary, the number is rounded. If the precision is missing, six digits are given; if the precision is explicitly *0* and # isn't specified, no digits and no decimal point are printed;

- E As *e*, but with an uppercase *E* used to identify the exponent;
- g The *float* or *double* argument is printed in style d, in style f, or in style e, whichever gives the greatest precision in minimum space;
- G As *g*, but with an uppercase *E* used to identify the exponent.
- c The character argument is printed. Null characters are ignored;
- s The argument is taken to be a string (character pointer), and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is 0 or missing, all characters up to a null are printed.
- p The argument is taken to be a pointer. The representation printed is implementation-dependent.
- u The unsigned integer argument is converted to decimal notation;

In no case does a nonexistent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.

Here is a more elaborate example:

```
char* line_format = "\n#line %d \"%s\"\n";

int main()
{
    int line = 13;
    char* file_name = "C++/main.c";

    printf("int a;\n");
    printf(line_format, line, file_name);
    printf("int b;\n");
}
```

which produces:

```
int a;

#line 13 "C++/main.c"
int b;
```

Using *printf*( ) is unsafe in the sense that type checking is not done. For example, here is a well-known way of getting unpredictable output, a core dump, or worse:

```
char x;
// ...
printf("bad input char: %s", x);          // %s should have been %c
```

The *printf*( ) does, however, provide great flexibility in a form that is familiar to C programmers.

Similarly, *getchar*( ) provides a familiar way of reading characters from input:

```
int i;
while ((i=getchar())!=EOF) {// C character input
    // use i
}
```

Note that to be able to test for end-of-file against the *int* value *EOF*, the value of *getchar*( ) must be put into an *int* rather than into a *char*.

For further details of C I/O, see your C reference manual or Kernighan and Ritchie: *The C Programming Language [Kernighan,1988]*.

21.9 Advice [io.advice]

[1] Define << and >> for user-defined types with values that have meaningful textural representations; §21.2.3, §21.3.5.
[2] Use parentheses when printing expressions containing operators of low precedence; §21.2.
[3] You don't need to modify ***istream*** or ***ostream*** to add new << and >> operators; §21.2.3.
[4] You can define a function so that it behaves as a ***virtual*** function based on its *second* (or subsequent) argument; §21.2.3.1.
[5] Remember that by default >> skips whitespace; §21.3.2.
[6] Use lower-level input functions such as ***get***( ) and ***read***( ) primarily in the implementation of higher-lever input functions; §21.3.4.
[7] Be careful with the termination criteria when using ***get***( ), ***getline***( ), and ***read***( ); §21.3.4.
[8] Prefer manipulators to state flags for controlling I/O; §21.3.3, §21.4, §21.4.6.
[9] Use exceptions to catch rare I/O errors (only); §21.3.6.
[10] Tie streams used for interactive I/O; §21.3.7.
[11] Use sentries to concentrate entry and exit code for many functions in one place; §21.3.8.
[12] Don't use parentheses after a no-argument manipulator; §21.4.6.2.
[13] Remember to ***#include <iomanip>*** when using standard manipulators; §21.4.6.2.
[14] You can achieve the effect (and efficiency) of a ternary operator by defining a simple function object; §21.4.6.3.
[15] Remember that ***width*** specifications apply to the following I/O operation only; §21.4.4.
[16] Remember that ***precision*** specifications apply to all following floating-point output operations; §21.4.3.
[17] Use string streams for in-memory formatting; §21.5.3.
[18] You can specify a mode for a file stream ; §21.5.1.
[19] Distinguish sharply between formatting (***iostream***s) and buffering (***streambuf***s) when extending the I/O system; §21.1, §21.6.
[20] Implement nonstandard ways of transmitting values as stream buffers; §21.6.4.
[21] Implement nonstandard ways of formatting values as stream operations; §21.2.3, §21.3.5.
[22] You can isolate and encapsulate calls of user-defined code by using a pair of functions; §21.6.4.
[23] You can use ***in_avail***( ) to determine whether an input operation will block before reading; §21.6.4.
[24] Distinguish between simple operations that need to be efficient and operations that implement policy (make the former ***inline*** and the latter ***virtual***); §21.6.4.
[25] Use ***locale*** to localize ''cultural differences;'' §21.7.
[26] Use ***sync_with_stdio***(***x***) to mix C-style and C++-style I/O and to disassociate C-style and C++-style I/O; §21.8.
[27] Beware of type errors in C-style I/O; §21.8.

21.10 Exercises [io.exercises]

1. (∗1.5) Read a file of floating-point numbers, make complex numbers out of pairs of numbers read, and write out the complex numbers.
2. (∗1.5) Define a type ***Name_and_address***. Define `<<` and `>>` for it. Copy a stream of ***Name_and_address*** objects.
3. (∗2.5) Copy a stream of ***Name_and_address*** objects in which you have inserted as many errors as you can think of (e.g., format errors and premature end of string). Handle these errors in a way that ensures that the copy function reads most of the correctly formatted ***Name_and_address***es, even when the input is completely messed up.
4. (∗2.5) Redefine the I/O format ***Name_and_address*** to make it more robust in the presence of format errors.
5. (∗2.5) Design some functions for requesting and reading information of various types. Ideas: integer, floating-point number, file name, mail address, date, personal information, etc. Try to make them foolproof.
6. (∗1.5) Write a program that prints (a) all lowercase letters, (b) all letters, (c) all letters and digits, (d) all characters that may appear in a C++ identifier on your system, (e) all punctuation characters, (f) the integer value of all control characters, (g) all whitespace characters, (h) the integer value of all whitespace characters, and finally (i) all printing characters.
7. (∗2) Read a sequence of lines of text into a fixed-sized character buffer. Remove all whitespace characters and replace each alphanumeric character with the next character in the alphabet (replace ***z*** by ***a*** and ***9*** by ***0***). Write out the resulting line.
8. (∗3) Write a ‘‘miniature’’ stream I/O system that provides classes ***istream***, ***ostream***, ***ifstream***, ***ofstream*** providing functions such as ***operator*<<`()` and ***operator*>>`()` for integers and operations such as ***open***`()` and ***close***`()` for files.
9. (∗4) Implement the C standard I/O library (<***stdio.h***>) using the C++ standard I/O library (<***iostream***>).
10. (∗4) Implement the C++ standard I/O library (<***iostream***>) using the C standard I/O library (<***stdio.h***>).
11. (∗4) Implement the C and C++ libraries so that they can be used simultaneously.
12. (∗2) Implement a class for which `[]` is overloaded to implement random reading of characters from a file.
13. (∗3) Repeat §21.10[12] but make `[]` useful for both reading and writing. Hint: Make `[]` return an object of a ‘‘descriptor type’’ for which assignment means ‘‘assign through descriptor to file’’ and implicit conversion to ***char*** ‘‘means read from file through descriptor.’’
14. (∗2) Repeat §21.10[13] but let `[]` index objects of arbitrary types, not just characters.
15. (∗3.5) Implement versions of ***istream*** and ***ostream*** that read and write numbers in their binary form rather than converting them into a character representation. Discuss the advantages and disadvantages of this approach compared to the character-based approach.
16. (∗3.5) Design and implement a pattern-matching input operation. Use ***printf***-style format strings to specify a pattern. It should be possible to try out several patterns against some input to find the actual format. One might derive a pattern-matching input class from ***istream***.
17. (∗4) Invent (and implement) a much better kind of pattern for pattern matching. Be specific about what is better about it.

18. (*2) Define an output manipulator ***based*** that takes two arguments – a base and an ***int*** value – and outputs the integer in the representation specified by the base. For example, ***based***(*2*,*9*) should print *1001*.
19. (*2) Write manipulators that turn character echoing on and off.
20. (*2) Implement ***Bound_form*** from §21.4.6.3 for the usual set of built-in types.
21. (*2) Re-implement ***Bound_form*** from §21.4.6.3 so that an output operation never overflows its ***width***( ). It should be possible for a programmer to ensure that output is never quietly truncated beyond its specified precision.
22. (*3) Implement an ***encrypt***(*k*) manipulator that ensures that output on its ***ostream*** is encrypted using the key *k*. Provide a similar ***decrypt***(*k*) manipulator for an ***istream***. Provide the means for turning the encryption off for a stream so that further I/O is cleartext.
23. (*2) Trace a character's route through your system from the keyboard to the screen for a simple:

```
char c;
cin >> c;
cout << c << endl;
```

24. (*2) Modify ***readints***( ) (§21.3.6) to handle all exceptions. Hint: Resource acquisition is initialization.
25. (*2.5) There is a standard way of reading, writing, and representing dates under control of a ***locale***. Find it in the documentation of your implementation and write a small program that reads and writes dates using this mechanism. Hint: ***struct tm***.
26. (*2.5) Define an ***ostream*** called ***ostrstream*** that can be attached to an array of characters (a C-style string) in a way similar to the way ***ostringstream*** is attached to a ***string***. However, do not copy the array into or out of the ***ostrstream***. The ***ostrstream*** should simply provide a way of writing to its array argument. It might be used for in-memory formatting like this:

```
char buf[message_size];
ostrstream ost(buf, message_size);
do_something(arguments, ost);        // output to buf through ost
cout << buf;                         // ost adds terminating 0
```

An operation such as ***do_something***( ) can write to the stream ***ost***, pass ***ost*** on to its suboperations, etc., using the standard output operations. There is no need to check for overflow because ***ost*** knows its size and will go into ***fail***( ) state when it is full. Finally, a ***display***( ) operation can write the message to a ''real'' output stream. This technique can be most useful for coping with cases in which the final display operation involves writing to something more complicated than a traditional line-oriented output device. For example, the text from ***ost*** could be placed in a fixed-sized area somewhere on a screen. Similarly, define class ***istrstream*** as an input string stream reading from a zero-terminated string of characters. Interpret the terminating zero character as end-of-file. These ***strstream***s were part of the original streams library and can often be found in <***strstream.h***>.

27. (*2.5) Implement a manipulator ***general***( ) that resets a stream to its original (general) format in the same way a ***scientific***( ) (§21.4.6.2) sets a stream to use scientific format.

22

Numerics

The purpose of computing is insight, not numbers.
– R.W. Hamming

... but for the student,
numbers are often the best road to insight.
– A. Ralston

Introduction — numeric limits — mathematical functions — ***valarray*** — vector operations — slices — ***slice_array*** — elimination of temporaries — ***gslice_array*** — ***mask_array*** — ***indirect_array*** — ***complex*** — generalized algorithms — random numbers — advice — exercises.

22.1 Introduction [num.intro]

It is rare to write any real code without doing some calculation. However, most code requires little mathematics beyond simple arithmetic. This chapter presents the facilities the standard library offers to people who go beyond that.

Neither C nor C++ were designed primarily with numeric computation in mind. However, numeric computation typically occurs in the context of other work – such as database access, networking, instrument control, graphics, simulation, financial analysis, etc. – so C++ becomes an attractive vehicle for computations that are part of a larger system. Furthermore, numeric methods have come a long way from being simple loops over vectors of floating-point numbers. Where more complex data structures are needed as part of a computation, C++'s strengths become relevant. The net effect is that C++ is increasingly used for scientific and engineering computation involving sophisticated numerics. Consequently, facilities and techniques supporting such computation have emerged. This chapter describes the parts of the standard library that support numerics and presents a few techniques for dealing with issues that arise when people express numeric

computations in C++. I make no attempt to teach numeric methods. Numeric computation is a fascinating topic in its own right. To understand it, you need a good course in numerical methods or at least a good textbook – not just a language manual and tutorial.

22.2 Numeric Limits [num.limits]

To do anything interesting with numbers, we typically need to know something about general properties of built-in numeric types that are implementation-defined rather than fixed by the rules of the language itself (§4.6). For example, what is the largest ***int***? What is the smallest ***float***? Is a ***double*** rounded or truncated when assigned to a ***float***? How many bits are there in a ***char***?

Answers to such questions are provided by the specializations of the ***numeric_limits*** template presented in ***<limits>***. For example:

```
void f(double d, int i)
{
    if (numeric_limits<char>::digits != 8) {
        // unusual bytes (number of bits not 8)
    }

    if (i<numeric_limits<short>::min() || numeric_limits<short>::max()<i) {
        // i cannot be stored in a short without loss of precision
    }

    if (0<d && d<numeric_limits<double>::epsilon()) d = 0;

    if (numeric_limits<Quad>::is_specialized) {
        // limits information available for type Quad
    }
}
```

Each specialization provides the relevant information for its argument type. Thus, the general ***numeric_limits*** template is simply a notational handle for a set of constants and inline functions:

```
template<class T> class numeric_limits {
public:
    static const bool is_specialized = false; // is information available for numeric_limits<T>?

    // uninteresting defaults
};
```

The real information is in the specializations. Each implementation of the standard library provides a specialization of ***numeric_limits*** for each fundamental type (the character types, the integer and floating-point types, and ***bool***) but not for any other plausible candidates such as ***void***, enumerations, or library types (such as ***complex<double>***).

For an integral type such as ***char***, only a few pieces of information are of interest. Here is ***numeric_limits<char>*** for an implementation in which a ***char*** has 8 bits and is signed:

```
class numeric_limits<char> {
public:
    static const bool is_specialized = true;      // yes, we have information

    static const int digits = 8;                  // number of bits (''binary digits'')

    static const bool is_signed = true;           // this implementation has char signed
    static const bool is_integer = true;          // char is an integral type

    inline static char min() throw() { return -128; }    // smallest value
    inline static char max() throw() { return 127; }     // largest value

    // lots of declarations not relevant to a char
};
```

Most members of ***numeric_limits*** are intended to describe floating-point numbers. For example, this describes one possible implementation of ***float***:

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2;          // base of exponent (in this case, binary)
    static const int digits = 24;        // number radix digits in mantissa
    static const int digits10 = 6;       // number of base 10 digits in mantissa

    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;

    inline static float min() throw() { return 1.17549435E-38F; }
    inline static float max() throw() { return 3.40282347E+38F; }

    inline static float epsilon() throw() { return 1.19209290E-07F; }
    inline static float round_error() throw() { return 0.5F; }

    inline static float infinity() throw() { return /* some value */; }
    inline static float quiet_NaN() throw() { return /* some value */; }
    inline static float signaling_NaN() throw() { return /* some value */; }
    inline static float denorm_min() throw() { return min(); }

    static const int min_exponent = -125;
    static const int min_exponent10 = -37;
    static const int max_exponent = +128;
    static const int max_exponent10 = +38;

    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm = denorm_absent;    // enum from <limits>
    static const bool has_denorm_loss = false;

    static const bool is_iec559 = true;    // conforms to IEC-559
    static const bool is_bounded = true;
    static const bool is_modulo = false;
```

```
        static const bool traps = true;
        static const bool tinyness_before = true;

        static const float_round_style round_style = round_to_nearest;    // enum from <limits>
    };
```

Note that ***min*()** is the smallest *positive* normalized number and that ***epsilon*** is the smallest positive floating-point number such that ***1+epsilon-1*** is representable.

When defining a scalar type along the lines of the built-in ones, it is a good idea also to provide a suitable specialization of ***numeric_limits***. For example, if I wrote a quadruple-precision type ***Quad*** or if a vendor provided an extended-precision integer ***long long***, a user could reasonably expect ***numeric_limits<Quad>*** and ***numeric_limits<long long>*** to be supplied.

One can imagine specializations of ***numeric_limits*** describing properties of user-defined types that have little to do with floating-point numbers. In such cases, it is usually better to use the general technique for describing properties of a type than to specialize ***numeric_limits*** with properties not considered in the standard. Latin1...UL float_denom_style

Floating-point values are represented as inline functions. Integral values in ***numeric_limits***, however, must be represented in a form that allows them to be used in constant expressions. That implies that they must have in-class initializers (§10.4.6.2). If you use ***static const*** members rather than enumerators for that, remember to define the ***static***s.

22.2.1 Limit Macros [num.limit.c]

From C, C++ inherited macros that describe properties of integers. These are found in ***<climits>*** and ***<limits.h>*** and have names such as ***CHAR_BIT*** and ***INT_MAX***. Similarly, ***<cfloat>*** and ***<float.h>*** define macros describing properties of floating-point numbers. They have names such as ***DBL_MIN_EXP***, ***FLT_RADIX***, and ***LDBL_MAX***.

As ever, macros are best avoided.

22.3 Standard Mathematical Functions [num.math]

The headers ***<cmath>*** and ***<math.h>*** provide what is commonly referred to as ‘‘the usual mathematical functions:’’

```
double abs(double);                 // absolute value; not in C, same as fabs()
double fabs(double);                // absolute value

double ceil(double d);              // smallest integer not less than d
double floor(double d);             // largest integer not greater than d

double sqrt(double d);              // square root of d, d must be non-negative

double pow(double d, double e);     // d to the power of e,
                                    // error if d==0 and e<=0 or if d<0 and e isn't an integer.
double pow(double d, int i);        // d to the power of i; not in C
```

```
double cos(double);                    // cosine
double sin(double);                    // sine
double tan(double);                    // tangent

double acos(double);                   // arc cosine
double asin(double);                   // arc sine
double atan(double);                   // arc tangent
double atan2(double x, double y);      // atan(x/y)

double sinh(double);                   // hyperbolic sine
double cosh(double);                   // hyperbolic cosine
double tanh(double);                   // hyperbolic tangent

double exp(double);                    // exponential, base e
double log(double d);                  // natural (base e) logarithm, d must be > 0
double log10(double d);                // base 10 logarithm, d must be > 0

double modf(double d, double* p);      // return fractional part of d, place integral part in *p
double frexp(double d, int* p);        // find x in [.5,1) and y so that d = x*pow(2,y),
                                       // return x and store y in *p
double fmod(double d, double m);       // floating-point remainder, same sign as d
double ldexp(double d, int i);         // d*pow(2,i)
```

In addition, **<cmath>** and **<math.h>** supply these functions for ***float*** and ***long double*** arguments.

Where several values are possible results – as with ***asin()*** – the one nearest to ***0*** is returned. The result of ***acos()*** is non-negative.

Errors are reported by setting ***errno*** from **<cerrno>** to ***EDOM*** for a domain error and to ***ERANGE*** for a range error. For example:

```
void f()
{
    errno = 0; // clear old error state
    sqrt(-1);
    if (errno==EDOM) cerr << "sqrt() not defined for negative argument";
    pow(numeric_limits<double>::max(),2);
    if (errno == ERANGE) cerr << "result of pow() too large to represent as a double";
}
```

For historical reasons, a few mathematical functions are found in the **<cstdlib>** header rather than in **<cmath>**:

```
int abs(int);          // absolute value
long abs(long);        // absolute value (not in C)
long labs(long);       // absolute value

struct div_t { implementation_defined quot, rem; };
struct ldiv_t { implementation_defined quot, rem; };

div_t div(int n, int d);              // divide n by d, return (quotient,remainder)
ldiv_t div(long int n, long int d);   // divide n by d, return (quotient,remainder) (not in C)
ldiv_t ldiv(long int n, long int d);  // divide n by d, return (quotient,remainder)
```

22.4 Vector Arithmetic [num.valarray]

Much numeric work relies on relatively simple single-dimensional vectors of floating-point values. In particular, such vectors are well supported by high-performance machine architectures, libraries relying on such vectors are in wide use, and very aggressive optimization of code using such vectors is considered essential in many fields. Consequently, the standard library provides a vector – called ***valarray*** – designed specifically for speed of the usual numeric vector operations.

When looking at the ***valarray*** facilities, it is wise to remember that they are intended as a relatively low-level building block for high-performance computation. In particular, the primary design criterion wasn't ease of use, but rather effective use of high-performance computers when relying on aggressive optimization techniques. If your aim is flexibility and generality rather than efficiency, you are probably better off building on the standard containers from Chapter 16 and Chapter 17 than trying to fit into the simple, efficient, and deliberately traditional framework of ***valarray***.

One could argue that ***valarray*** should have been called ***vector*** because it is a traditional mathematical vector and that ***vector*** (§16.3) should have been called ***array***. However, this is not the way the terminology evolved. A ***valarray*** is a vector optimized for numeric computation, a ***vector*** is a flexible container designed for holding and manipulating objects of a wide variety of types, and an array is a low-level, built-in type.

The ***valarray*** type is supported by four auxiliary types for specifying subsets of a ***valarray***:

- ***slice_array*** and ***gslice_array*** represent the notion of slices (§22.4.6, §22.4.8),
- ***mask_array*** specifies a subset by marking each element in or out (§22.4.9), and
- ***indirect_array*** lists the indices of the elements to be considered (§22.4.10).

22.4.1 Valarray Construction [num.valarray.ctor]

The ***valarray*** type and its associated facilities are defined in namespace ***std*** and presented in <***valarray***>:

```
template<class T> class std::valarray {
    // representation
public:
    typedef T value_type;

    valarray();                                  // valarray with size()==0
    explicit valarray(size_t n);                 // n elements with value T()
    valarray(const T& val, size_t n);            // n elements with value val
    valarray(const T* p, size_t n);              // n elements with values p[0], p[1], ...
    valarray(const valarray& v);                 // copy of v

    valarray(const slice_array<T>&);             // see §22.4.6
    valarray(const gslice_array<T>&);            // see §22.4.8
    valarray(const mask_array<T>&);              // see §22.4.9
    valarray(const indirect_array<T>&);          // see §22.4.10

    ~valarray();

    // ...
};
```

This set of constructors allows us to initialize ***valarray***s from the auxiliary numeric array types and from single values. For example:

```
valarray<double> v0;                    // placeholder, we can assign to v0 later
valarray<float> v1(1000);               // 1000 elements with value float()==0.0F

valarray<int> v2(-1,2000);              // 2000 elements with value –1
valarray<double> v3(100,9.8064);        // bad mistake: floating-point valarray size

valarray<double> v4 = v3;               // v4 has v3.size() elements
```

In the two-argument constructors, the value comes before the number of elements. This differs from the convention for other standard containers (§16.3.4).

The number of elements of an argument ***valarray*** to a copy constructor determines the size of the resulting ***valarray***.

Most programs need data from tables or input; this is supported by a constructor that copies elements from a built-in array. For example:

```
const double vd[] = { 0, 1, 2, 3, 4 };
const int vi[] = { 0, 1, 2, 3, 4 };

valarray<double> v3(vd,4);      // 4 elements: 0,1,2,3
valarray<double> v4(vi,4);      // type error: vi is not pointer to double
valarray<double> v5(vd,8);      // undefined: too few elements in initializer
```

This form of initialization is important because numeric software that produces data in the form of large arrays is common.

The ***valarray*** and its auxiliary facilities were designed for high-speed computing. This is reflected in a few constraints on users and by a few liberties granted to implementers. Basically, an implementer of ***valarray*** is allowed to use just about every optimization technique you can think of. For example, operations may be inlined and the ***valarray*** operations are assumed to be free of side effects (except on their explicit arguments of course). Also, ***valarray***s are assumed to be alias free, and the introduction of auxiliary types and the elimination of temporaries is allowed as long as the basic semantics are maintained. Thus, the declarations in ***<valarray>*** may look somewhat different from what you find here (and in the standard), but they should provide the same operations with the same meaning for code that doesn't go out of the way to break the rules. In particular, the elements of a ***valarray*** should have the usual copy semantics (§17.1.4).

22.4.2 Valarray Subscripting and Assignment [num.valarray.sub]

For ***valarray***s, subscripting is used both to access individual elements and to obtain subarrays:

```
template<class T> class valarray {
public:
    // ...
    valarray& operator=(const valarray& v);    // copy v
    valarray& operator=(const T& val);         // assign val to every element

    T operator[](size_t) const;
    T& operator[](size_t);
```

```
        valarray operator[](slice) const;                    // see §22.4.6
        slice_array<T> operator[](slice);

        valarray operator[](const gslice&) const;            // see §22.4.8
        gslice_array<T> operator[](const gslice&);

        valarray operator[](const valarray<bool>&) const;    // see §22.4.9
        mask_array<T> operator[](const valarray<bool>&);

        valarray operator[](const valarray<size_t>&) const;  // see §22.4.10
        indirect_array<T> operator[](const valarray<size_t>&);

        valarray& operator=(const slice_array<T>&);          // see §22.4.6
        valarray& operator=(const gslice_array<T>&);         // see §22.4.8
        valarray& operator=(const mask_array<T>&);           // see §22.4.9
        valarray& operator=(const indirect_array<T>&);       // see §22.4.10

        // ...
    };
```

A ***valarray*** can be assigned to another of the same size. As one would expect, ***v1=v2*** copies every element of ***v2*** into its corresponding position in ***v1***. If ***valarray***s have different sizes, the result of assignment is undefined. Because ***valarray*** is designed to be optimized for speed, it would be unwise to assume that assigning with a ***valarray*** of the wrong size would cause an easily comprehensible error (such as an exception) or other ''reasonable'' behavior.

In addition to this conventional assignment, it is possible to assign a scalar to a ***valarray***. For example, ***v=7*** assigns ***7*** to every element of the ***valarray v***. This may be surprising, and is best understood as an occasionally useful degenerate case of the operator assignment operations (§22.4.3).

Subscripting with an integer behaves conventionally and does not perform range checking.

In addition to the selection of individual elements, ***valarray*** subscripting provides four ways of extracting subarrays (§22.4.6). Conversely, assignment (and constructors §22.4.1) accepts such subarrays as operands. The set of assignments on ***valarray*** ensures that it is not necessary to convert an auxiliary array type, such as ***slice_array***, to ***valarray*** before assigning it. An implementation may similarly replicate other vector operations, such as + and *, to assure efficiency. In addition, many powerful optimization techniques exist for vector operations involving ***slice***s and the other auxiliary vector types.

22.4.3 Member Operations [num.valarray.member]

The obvious, as well as a few less obvious, member functions are provided:

```
    template<class T> class valarray {
    public:
        // ...

        valarray& operator*=(const T& arg);      // v[i]*=arg for every element
        // similarly: /=, %=, +=, –=, ^=, &=, |=, <<=, and >>=

        T sum() const;                           // sum of elements
```

```
    valarray shift(int i) const;                    // logical shift (left for 0<i, right for i<0)
    valarray cshift(int i) const;                   // cyclic shift (left for 0<i, right for i<0)

    valarray apply(T f(T)) const;                   // result[i] = f(v[i]) for every element
    valarray apply(T f(const T&)) const;

    valarray operator-() const;                     // result[i] = -v[i] for every element
    valarray operator+() const;                     // result[i] = +v[i] for every element
    valarray operator~() const;                     // result[i] = ~v[i] for every element
    valarray operator!() const;                     // result[i] = !v[i] for every element

    T min() const; // smallest value using < for comparison; if size()==0 the value is undefined
    T max() const; // largest value using < for comparison; if size()==0 the value is undefined

    size_t size() const;                            // number of elements
    void resize(size_t n, const T& val = T());      // n elements with value val
};
```

For example, if *v* is a ***valarray***, it can be scaled like this: ***v*=.2***, and this: ***v/=1.3***. That is, applying a scalar to a vector means applying the scalar to each element of the vector. As usual, it is easier to optimize uses of `*=` than uses of a combination of `*` and `=` (§11.3.1).

Note that the non-assignment operations construct a new ***valarray***. For example:

```
double incr(double d) { return d+1; }

void f(valarray<double>& v)
{
    valarray<double> v2 = v.apply(incr);        // produce incremented valarray
}
```

This does not change the value of ***v***. Unfortunately, ***apply*()** does not accept a function object (§18.4) as an argument (§22.9[1]).

The logical and cyclic shift functions, ***shift*()** and ***cshift*()**, return a new ***valarray*** with the elements suitably shifted and leave the original one unchanged. For example, the cyclic shift ***v2=v.cshift(n)*** produces a ***valarray*** so that ***v2[i]==v[(i+n)%v.size()]***. The logical shift ***v3=v.shift(n)*** produces a ***valarray*** so that ***v3[i]*** is ***v[i+n]*** if ***i+n*** is a valid index for ***v***. Otherwise, the result is the default element value. This implies that both ***shift*()** and ***cshift*()** shift left when given a positive argument and right when given a negative argument. For example:

```
void f()
{
    int alpha[] = { 1, 2, 3, 4, 5 ,6, 7, 8 };
    valarray<int> v(alpha,8);           // 1, 2, 3, 4, 5, 6, 7, 8
    valarray<int> v2 = v.shift(2);      // 3, 4, 5, 6, 7, 8, 0, 0
    valarray<int> v3 = v<<2;            // 4, 8, 12, 16, 20, 24, 28, 32
    valarray<int> v4 = v.shift(-2);     // 0, 0, 1, 2, 3, 4, 5, 6
    valarray<int> v5 = v>>2;            // 0, 0, 0, 1, 1, 1, 1, 2
    valarray<int> v6 = v.cshift(2);     // 3, 4, 5, 6, 7, 8, 1, 2
    valarray<int> v7 = v.cshift(-2);    // 7, 8, 1, 2, 3, 4, 5, 6
}
```

For ***valarray***s, `>>` and `<<` are bit shift operators, rather than element shift operators or I/O

operators (§22.4.4). Consequently, <<= and >>= can be used to shift bits within elements of an integral type. For example:

```
void f(valarray<int> vi, valarray<double> vd)
{
    vi <<= 2;  // vi[i]<<=2 for all elements of vi
    vd <<= 2;  // error: shift is not defined for floating-point values
}
```

It is possible to change the size of a ***valarray***. However, ***resize*( )** is ***not*** an operation intended to make ***valarray*** into a data structure that can grow dynamically the way a ***vector*** and a ***string*** can. Instead, ***resize*( )** is a re-initialize operation that replaces the existing contents of a ***valarray*** by a set of default values. The old values are lost.

Often, a resized ***valarray*** is one that we created as an empty vector. Consider how we might initialize a ***valarray*** from input:

```
void f()
{
    int n = 0;
    cin >> n;                                   // read array size
    if (n<=0) error("bad array bound");

    valarray<double> v(n);                      // make an array of the right size
    int i = 0;
    while (i<n && cin>>v[i++]) ;                // fill array
    if (i!=n) error("too few elements on input");

    // ...
}
```

If we want to handle the input in a separate function, we might do it like this:

```
void initialize_from_input(valarray<double>& v)
{

    int n = 0;
    cin >> n;                                   // read array size
    if (n<=0) error("bad array bound");

    v.resize(n);                                // make v the right size
    int i = 0;
    while (i<n && cin>>v[i++]) ;                // fill array
    if (i!=n) error("too few elements on input");
}

void g()
{
    valarray<double> v;                 // make a default array
    initialize_from_input(v);           // give v the right size and elements
    // ...
}
```

This avoids copying large amounts of data.

If we want a ***valarray*** holding valuable data to grow dynamically, we must use a temporary:

```
void grow(valarray<int>& v, size_t n)
{
    if (n<=v.size()) return;

    valarray<int> tmp(n);                    // n default elements

    copy(&v[0],&v[v.size()],&tmp[0]);        // copy algorithm from §18.6.1
    v.resize(n);
    copy(&tmp[0],&tmp[v.size()],&v[0]);
}
```

This is *not* the intended way to use ***valarray***. A ***valarray*** is intended to have a fixed size after being given its initial value.

The elements of a ***valarray*** form a sequence; that is, `v[0]..v[n-1]` are contiguous in memory. This implies that ***T**** is a random-access iterator (§19.2.1) for ***valarray<T>*** so that standard algorithms, such as ***copy()***, can be used. However, it would be more in the spirit of ***valarray*** to express the copy in terms of assignment and subarrays:

```
void grow2(valarray<int>& v, size_t n)
{
    if (n<=v.size()) return;

    valarray<int> tmp(n);      // n default elements
    slice s(0,v.size(),1);     // subarray of v.size() elements (see §22.4.5)

    tmp[s] = v;
    v.resize(n);
    v[s] = tmp;
}
```

If for some reason input data is organized so that you have to count the elements before knowing the size of vector needed to hold them, it is usually best to read the input into a ***vector*** (§16.3.5) and then copy the elements into a ***valarray***.

22.4.4 Nonmember Operations [valarray.ops]

The usual binary operators and mathematical functions are provided:

```
template<class T> valarray<T> operator*(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator*(const valarray<T>&, const T&);
template<class T> valarray<T> operator*(const T&, const valarray<T>&);

// similarly: /, %, +, –, ^, &, |, <<, >>, &&, ||, ==, !=, <, >, <=, >=, atan2, and pow

template<class T> valarray<T> abs(const valarray<T>&);

// similarly: acos, asin, atan, cos, cosh, exp, log, log10, sin, sinh, sqrt, tan, and tanh
```

The binary operations are defined for ***valarray***s and for combinations of a ***valarray*** and its scalar type. For example:

```
void f(valarray<double>& v, valarray<double>& v2, double d)
{
    valarray<double> v3 = v*v2;     // v3[i] = v[i]*v2[i] for all i
    valarray<double> v4 = v*d;      // v4[i] = v[i]*d for all i
    valarray<double> v5 = d*v2;     // v5[i] = d*v2[i] for all i

    valarray<double> v6 = cos(v);   // v6[i] = cos(v[i]) for all i
}
```

These vector operations all apply their operations to each element of their operand(s) in the way indicated by the * and *cos*( ) examples. Naturally, an operation can be used only if the corresponding operation is defined for the template argument type. Otherwise, the compiler will issue an error when trying to specialize the template (§13.5).

Where the result is a ***valarray***, its length is the same as its ***valarray*** operand. If the lengths of the two arrays are not the same, the result of a binary operator on two ***valarray***s is undefined.

Curiously enough, no I/O operations are provided for ***valarray*** (§22.4.3); << and >> are shift operations. However, I/O versions of >> and << for ***valarray*** are easily defined (§22.9[5]).

Note that these ***valarray*** operations return new ***valarray***s rather than modifying their operands. This can be expensive, but it doesn't have to be when aggressive optimization techniques are applied (e.g., see §22.4.7).

All of the operators and mathematical functions on ***valarray***s can also be applied to ***slice_array***s (§22.4.6), ***gslice_array***s (§22.4.8), ***mask_array***s (§22.4.9), ***indirect_array***s (§22.4.10), and combinations of these types. However, an implementation is allowed to convert an operand that is not a ***valarray*** to a ***valarray*** before performing a required operation.

22.4.5 Slices [num.slice]

A ***slice*** is an abstraction that allows us to manipulate a vector efficiently as a matrix of arbitrary dimension. It is the key notion of Fortran vectors and of the BLAS (Basic Linear Algebra Subprograms) library, which is the basis for much numeric computation. Basically, a slice is every ***n***th element of some part of a ***valarray***:

```
class std::slice {
    // starting index, a length, and a stride
public:
    slice();
    slice(size_t start, size_t size, size_t stride);

    size_t start() const;       // index of first element
    size_t size() const;        // number of elements
    size_t stride() const;      // element n is at start()+n*stride()
};
```

A *stride* is the distance (in number of elements) between two elements of the ***slice***. Thus, a ***slice*** describes a sequence of integers. For example:

```
size_t slice_index(const slice& s, size_t i) // map i to its corresponding index
{
    return s.start()+i*s.stride();
}

void print_seq(const slice& s)   // print the elements of s
{
    for (int i = 0; i<s.size(); i++) cout << slice_index(s,i) << " ";
}

void f()
{
    print_seq(slice(0,3,4));  // row 0
    cout << ", ";
    print_seq(slice(1,3,4));  // row 1
    cout << ", ";
    print_seq(slice(0,4,1));  // column 0
    cout << ", ";
    print_seq(slice(4,4,1));  // column 1
}
```

prints *0 4 8 , 1 5 9 , 0 1 2 3 , 4 5 6 7*.

In other words, a ***slice*** describes a mapping of non-negative integers into indices. The number of elements (the ***size***( )) doesn't affect the mapping (addressing) but simply allows us to find the end of a sequence. This mapping can be used to simulate two-dimensional arrays within a one-dimensional array (such as ***valarray***) in an efficient, general, and reasonably convenient way. Consider a 3-by-4 matrix the way we often think of it (§C.7):

00	01	02
10	11	12
20	21	22
30	31	32

Following Fortran conventions, we can lay it out in memory like this:

0				4				8			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3								

This is *not* the way arrays are laid out in C++ (see §C.7). However, we should be able to present a concept with a clean and logical interface and then choose a representation to suit the constraints of the problem. Here, I have chosen to use Fortran layout to ease the interaction with numeric software that follows that convention. I have not, however, gone so far as to start indexing from ***1*** rather than ***0***; that is left as an exercise (§22.9[9]). Much numeric computation is done and will remain done in a mixture of languages and using a variety of libraries. Often the ability to manipulate data in a variety of formats determined by those libraries and language standards is essential.

Row *x* can be described by a *slice*(*x*,3,4). That is, the first element of row *x* is the *x*th element of the vector, the next element of the row is the (*x*+4)th, etc., and there are *3* elements in each row. In the figures, *slice*(*0*,*3*,*4*) describes the row *00*, *01*, and *02*.

Column *y* can be described by *slice*(*4*y*,*4*,*1*). That is, the first element of column *y* is the *4*y*th element of the vector, the next element of the column is the (*4*y*+*1*)th, etc., and there are *4* elements in each column. In the figures, *slice*(*0*,*4*,*1*) describes the column *00*, *10*, *20*, and *30*.

In addition to its use for simulating two-dimensional arrays, a *slice* can describe many other sequences. It is a fairly general way of specifying very simple sequences. This notion is explored further in §22.4.8.

One way of thinking of a slice is as an odd kind of iterator: a *slice* allows us to describe a sequence of indices for a *valarray*. We could build a real iterator based on that:

```
template<class T> class Slice_iter {
    valarray<T>* v;
    slice s;
    size_t curr;      // index of current element

    T& ref(size_t i) const { return (*v)[s.start()+i*s.stride()]; }
public:
    Slice_iter(valarray<T>* vv, slice ss) :v(vv), s(ss), curr(0) { }

    Slice_iter end()
    {
        Slice_iter t = *this;
        t.curr = s.size();          // index of last-plus-one element
        return t;
    }

    Slice_iter& operator++() { curr++; return *this; }
    Slice_iter operator++(int) { Slice_iter t = *this; curr++; return t; }

    T& operator[](size_t i) { return ref(curr=i); }      // C style subscript
    T& operator()(size_t i) { return ref(curr=i); }      // Fortran-style subscript
    T& operator*() { return ref(curr); }                 // current element

    // ...
};
```

Since a *slice* has a size, we could even provide range checking. Here, I have taken advantage of *slice*::*size*() to provide an *end*() operation to provide an iterator for the one-past-the-end element of the *valarray*.

Since a *slice* can describe either a row or a column, the *Slice_iter* allows us to traverse a *valarray* by row or by column.

For *Slice_iter* to be useful, ==, !=, and < must be defined:

```
template<class T> bool operator==(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr==q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}
```

```
template<class T> bool operator!=(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return !(p==q);
}

template<class T> bool operator<(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr<q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}
```

22.4.6 Slice_array [num.slicearray]

From a ***valarray*** and a ***slice***, we can build something that looks and feels like a ***valarray***, but which is really simply a way of referring to the subset of the array described by the slice. Such a ***slice_array*** is defined like this:

```
template <class T> class std::slice_array {
public:
    typedef T value_type;

    void operator=(const valarray<T>&);
    void operator=(const T& val);                    // assign val to each element

    void operator*=(const valarray<T>& val);         // v[i]*=val for each element
    // similarly: /=, %=, +=, –=, ^=, &=, |=, <<=, >>=

    ~slice_array();
private:
    slice_array();                                   // prevent construction
    slice_array(const slice_array&);                 // prevent copying
    slice_array& operator=(const slice_array&);      // prevent copying

    valarray<T>* p;                                  // implementation-defined representation
    slice s;
};
```

A user cannot directly create a ***slice_array***. Instead, the user subscripts a ***valarray*** to create a ***slice_array*** for a given slice. Once the ***slice_array*** is initialized, all references to it indirectly go to the ***valarray*** for which it is created. For example, we can create something that represents every second element of an array like this:

```
void f(valarray<double>& d)
{
    slice_array<double>& v_even = d[slice(0,d.size()/2,2)];
    slice_array<double>& v_odd = d[slice(1,d.size()/2,2)];

    v_odd *= 2;          // double every odd element of d
    v_even = 0;          // assign 0 to every even element of d
}
```

The ban on copying ***slice_array***s is necessary so as to allow optimizations that rely on absence of aliases. It can be quite constraining. For example:

```
slice_array<double> row(valarray<double>& d, int i)
{
    slice_array<double> v = d[slice(0,2,d.size()/2)];  // error: attempt to copy

    return d[slice(i%2,i,d.size()/2)];                 // error: attempt to copy
}
```

Often copying a ***slice*** is a reasonable alternative to copying a ***slice_array***.

Slices can be used to express a variety of subsets of an array. For example, we might use slices to manipulate contiguous subarrays like this:

```
inline slice sub_array(size_t first, size_t count) // [first:first+count[
{
    return slice(first,count,1);
}

void f(valarray<double>& v)
{
    size_t sz = v.size();
    if (sz<2) return;
    size_t n = sz/2;
    size_t n2 = sz-n;

    valarray<double> half1(n);
    valarray<double> half2(n2);

    half1 = v[sub_array(0,n)];          // copy of first half of v
    half2 = v[sub_array(n,n2)];         // copy of second half of v

    // ...
}
```

The standard library does not provide a matrix class. Instead, the intent is for ***valarray*** and ***slice*** to provide the tools for building matrices optimized for a variety of needs. Consider how we might implement a simple two-dimensional matrix using a ***valarray*** and ***slice_array***s:

```
class Matrix {
    valarray<double>* v;
    size_t d1, d2;
public:
    Matrix(size_t x, size_t y);         // note: no default constructor
    Matrix& Matrix(const Matrix&);
    Matrix& operator=(const Matrix&);
    ~Matrix();

    size_t size() const { return d1*d2; }
    size_t dim1() const { return d1; }
    size_t dim2() const { return d2; }

    Slice_iter<double> row(size_t i);
    Cslice_iter<double> row(size_t i) const;
```

```
        Slice_iter<double> column(size_t i);
        Cslice_iter<double> column(size_t i) const;

        double& operator()(size_t x, size_t y);                    // Fortran-style subscripts
        double operator()(size_t x, size_t y) const;

        Slice_iter<double> operator()(size_t i) { return row(i); }
        Cslice_iter<double> operator()(size_t i) const { return row(i); }

        Slice_iter<double> operator[](size_t i) { return row(i); }      // C-style subscript
        Cslice_iter<double> operator[](size_t i) const { return row(i); }

        Matrix& operator*=(double);

        valarray<double>& array() { return *v; }
};
```

The representation of a ***Matrix*** is a ***valarray***. We impose dimensionality on that array through slicing. When necessary, we can view that representation as having one, two, three, etc., dimensions in the same way that we provide the default two-dimensional view through ***row***( ) and ***column***( ). The ***Slice_iter***s are used to circumvent the ban on copying ***slice_arrays***. I couldn't return a ***slice_array***:

```
        slice_array<double> row(size_t i) { return (*v)(slice(i,d1,d2)); }
```

so I returned an iterator containing a pointer to the ***valarray*** and the ***slice*** itself instead of a ***slice_array***.

We need an additional class ''iterator for slice of constants,'' ***Cslice_iter*** to express the distinction between a slice of a ***const Matrix*** and a slice of a non-***const Matrix***:

```
inline Slice_iter<double> Matrix::row(size_t i)
{
        return Slice_iter<double>(v,slice(i,d1,d2));
}

inline Cslice_iter<double> Matrix::row(size_t i) const
{
        return Cslice_iter<double>(v,slice(i,d1,d2));
}

inline Slice_iter<double> Matrix::column(size_t i)
{
        return Slice_iter<double>(v,slice(i*d2,d2,1));
}

inline Cslice_iter<double> Matrix::column(size_t i) const
{
        return Cslice_iter<double>(v,slice(i*d2,d2,1));
}
```

The definition of ***Cslice_iter*** is identical to that of ***Slice_iter***, except that it returns ***const*** references to elements of its slice.

The rest of the member operations are fairly trivial:

```
Matrix::Matrix(size_t x, size_t y)
{
    // check that x and y are sensible
    d1 = x;
    d2 = y;
    v = new valarray<double>(x*y);
}

double& Matrix::operator()(size_t x, size_t y)
{
    return row(x)[y];
}

double mul(const valarray<double>& v1, const valarray<double>& v2)
{
    double res = 0;
    for (int i = 0; i<v1.size(); i++) res+= v1[i]*v2[i];
    return res;
}

valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
    valarray<double> res(m.dim1());
    for (int i = 0; i<m.dim1(); i++) res(i) = mul(m.row(i),v);
    return res;
}

Matrix& Matrix::operator*=(double d)
{
    (*v) *= d;
    return *this;
}
```

I provided `(i,j)` to express *Matrix* subscripting because `()` is a single operator and because that notation is the most familiar to many in the numeric community. The concept of a row provides the more familiar (in the C and C++ communities) `[i][j]` notation:

```
void f(Matrix& m)
{
    m(1,2) = 5;          // Fortran-style subscripts
    m.row(1)(2) = 6;
    m.row(1)[2] = 7;
    m[1](2) = 8;         // undesirable mixed style (but it works)
    m[1][2] = 9;         // C++-style subscripts
}
```

The use of *slice_arrays* to express subscripting assumes a good optimizer.

Generalizing this to an *n*-dimensional matrix of arbitrary elements and with a reasonable set of operations is left as an exercise (§22.9[7]).

Maybe your first idea for a two-dimensional vector was something like this:

```
class Matrix {
    valarray< valarray<double> > v;
public:
    // ...
};
```

This would also work (§22.9[10]). However, it is not easy to match the efficiency and compatibility required by high-performance computations without dropping to the lower and more conventional level represented by *valarray* plus *slice*s.

22.4.7 Temporaries, Copying, and Loops [num.matrix]

If you build a vector or a matrix class, you will soon find that three related problems have to be faced to satisfy performance-conscious users:

[1] The number of temporaries must be minimized.

[2] Copying of matrices must be minimized.

[3] Multiple loops over the same data in composite operations must be minimized.

These issues are not directly addressed by the standard library. However, I can outline a technique that can be used to produce highly optimized implementations.

Consider *U=M*V+W*, where *U*, *V*, and *W* are vectors and *M* is a matrix. A naive implementation introduces temporary vectors for *M*V* and *M*V+W* and copies the results of *M*V* and *M*V+W*. A smart implementation calls a function *mul_add_and_assign*(&*U*,&*M*,&*V*,&*W*) that introduces no temporaries, copies no vectors, and touches each element of the matrices the minimum number of times.

This degree of optimization is rarely necessary for more than a few kinds of expressions, so a simple solution to efficiency problems is to provide functions such as *mul_add_and_assign*( ) and let the user call those where it matters. However, it is possible to design a *Matrix* so that such optimizations are applied automatically for expressions of the right form. That is, we can treat *U=M*V+W* as a use of a single operator with four operands. The basic technique was demonstrated for *ostream* manipulators (§21.4.6.3). In general, it can be used to make a combination of *n* binary operators act like an (*n+1*)-ary operator. Handling *U=M*V+W* requires the introduction of two auxiliary classes. However, the technique can result in impressive speedups (say, 30 times) on some systems by enabling more-powerful optimization techniques.

First, we define the result of multiplying a *Matrix* by a *Vector*:

```
struct MVmul {
    const Matrix& m;
    const Vector& v;

    MVmul(const Matrix& mm, const Vector &vv) :m(mm), v(vv) { }

    operator Vector();   // evaluate and return result
};

inline MVmul operator*(const Matrix& mm, const Vector& vv)
{
    return MVmul(mm,vv);
}
```

This ''multiplication'' does nothing except store references to its operands; the evaluation of *M*V* is deferred. The object produced by * is closely related to what is called a ***closure*** in many technical communities. Similarly, we can deal with what happens if we add a *Vector*:

```
struct MVmulVadd {
    const Matrix& m;
    const Vector& v;
    const Vector& v2;

    MVmulVadd(const MVmul& mv, const Vector& vv) :m(mv.m), v(mv.v), v2(vv) { }

    operator Vector();   // evaluate and return result
};

inline MVmulVadd operator+(const MVmul& mv, const Vector& vv)
{
    return MVmulVadd(mv,vv);
}
```

This defers the evaluation of *M*V+W*. We now have to ensure that it all gets evaluated using a good algorithm when it is assigned to a *Vector*:

```
class Vector {
    // ...
public:
    Vector(const MVmulVadd& m)                    // initialize by result of m
    {
        // allocate elements, etc.
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
    }

    Vector& operator=(const MVmulVadd& m)         // assign the result of m to *this
    {
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
        return *this;
    }
    // ...
};
```

Now *U=M*V+W* is automatically expanded to

```
U.operator=(MVmulVadd(MVmul(M,V),W))
```

which because of inlining resolves to the desired simple call

```
mul_add_and_assign(&U,&M,&V,&W)
```

Clearly, this eliminates the copying and the temporaries. In addition, we might write *mul_add_and_assign*( ) in an optimized fashion. However, if we just wrote it in a fairly simple and unoptimized fashion, it would still be in a form that offered great opportunities to an optimizer.

I introduced a new *Vector* (rather than using a *valarray*) because I needed to define assignment (and assignment must be a member function; §11.2.2). However, *valarray* is a strong candidate for the representation of that *Vector*.

The importance of this technique is that most really time-critical vector and matrix computations are done using a few relatively simple syntactic forms. Typically, there is no real gain in optimizing expressions of half-a-dozen operators this way; more conventional techniques (§11.6) suffice.

This technique is based on the idea of using compile-time analysis and closure objects to transfer evaluation of subexpression into an object representing a composite operation. It can be applied to a variety of problems with the common attribute that several pieces of information need to be gathered into one function before evaluation can take place. I refer to the objects generated to defer evaluation as *composition closure objects*, or simply *compositors*.

22.4.8 Generalized Slices [num.gslice]

The ***Matrix*** example in §22.4.6 showed how two ***slice***s could be used to describe rows and columns of a two-dimensional array. In general, a ***slice*** can describe any row or column of an ***n***-dimensional array (§22.9[7]). However, sometimes we need to extract a subarray that is not a row or a column. For example, we might want to extract the 2-by-3 matrix from the top-left corner of a 3-by-4 matrix:

00	01	02
10	11	12
20	21	22
30	31	32

Unfortunately, these elements are not allocated in a way that can be described by a single slice:

0	1	2									
00	10	20	30	01	11	21	31	02	12	22	32
				4	5	6					

A ***gslice*** is a ''generalized slice'' that contains (almost) the information from ***n*** slices:

```
class std::gslice {
    // instead of 1 stride and one size like slice, gslice holds n strides and n sizes
public:
    gslice();
    gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t start() const;                    // index of first element
    valarray<size_t> size() const;           // number of elements in dimension
    valarray<size_t> stride() const;         // stride for index[0], index[1], ...
};
```

The extra values allow a ***gslice*** to specify a mapping between ***n*** integers and an index to be used to address elements of an array. For example, we can describe the layout of the 2-by-3 matrix by a pair of (length,stride) pairs. As shown in §22.4.5, a length of ***2*** and a stride of ***4*** describes two

elements of a row of the 3-by-4 matrix, when Fortran layout is used. Similarly, a length of ***3*** and a stride of ***1*** describes 3 elements of a column. Together, they describe every element of the 2-by-3 submatrix. To list the elements, we can write:

```
size_t gslice_index(const gslice& s, size_t i, size_t j)
{
    return s.start()+i*s.stride()[0]+j*s.stride()[1];
}

size_t len[] = { 2, 3 };          // (len[0],str[0]) describes a row
size_t str[] = { 4, 1 };          // (len[1],str[1]) describes a column

valarray<size_t> lengths(len,2);
valarray<size_t> strides(str,2);

void f()
{
    gslice s(0,lengths,strides);

    for (int i = 0 ; i<s.size()[0]; i++) cout << gslice_index(s,i,0) << " ";// row
    cout << ", ";
    for (int j = 0 ; j<s.size()[1]; j++) cout << gslice_index(s,0,j) << " ";// column
}
```

This prints ***0 4 , 0 1 2***.

In this way, a ***gslice*** with two (length,stride) pairs describes a subarray of a 2-dimensional array, a ***gslice*** with three (length,stride) pairs describes a subarray of a 3-dimensional array, etc. Using a ***gslice*** as the index of a ***valarray*** yields a ***gslice_array*** consisting of the elements described by the ***gslice***. For example:

```
void f(valarray<float>& v)
{
    gslice m(0,lengths,strides);
    v[m] = 0; // assign 0 to v[0],v[1],v[2],v[4],v[5],v[6]
}
```

The ***gslice_array*** offers the same set of members as ***slice_array***. In particular, a ***gslice_array*** cannot be constructed directly by the user and cannot be copied (§22.4.6). Instead, a ***gslice_array*** is the result of using a ***gslice*** as the subscript of a ***valarray*** (§22.4.2).

22.4.9 Masks [num.mask]

A ***mask_array*** provides yet another way of specifying a subset of a ***valarray*** and making the result look like a ***valarray***. In the context of ***valarray***s, a mask is simply a ***valarray<bool>***. When a mask is used as a subscript for a ***valarray***, a ***true*** bit indicates that the corresponding element of the ***valarray*** is considered part of the result. This allows us to operate on a subset of a ***valarray*** even if there is no simple pattern (such as a ***slice***) that describes that subset. For example:

```
void f(valarray<double>& v)
{
    bool b[] = { true , false, false, true, false, true };
    valarray<bool> mask(b,6);                  // elements 0, 3, and 5

    valarray<double> vv = cos(v[mask]);        // vv[0]==cos(v[0]), vv[1]==cos(v[3]),
                                               // vv[2]==cos(v[5])
}
```

The ***mask_array*** offers the same set of members as ***slice_array***. In particular, a ***mask_array*** cannot be constructed directly by the user and cannot be copied (§22.4.6). Instead, a ***mask_array*** is the result of using a ***valarray<bool>*** as the subscript of a ***valarray*** (§22.4.2). The number of elements of a ***valarray*** used as a mask must not be greater than the number of elements of the ***valarray*** for which it is used as a subscript.

22.4.10 Indirect Arrays [num.indirect]

An ***indirect_array*** provides a way of arbitrarily subsetting and reordering a ***valarray***. For example:

```
void f(valarray<double>& v)
{
    size_t i[] = { 3, 2, 1, 0 };               // first four elements in reverse order
    valarray<size_t> index(i,4);               // elements 3, 2, 1, 0 (in that order)

    valarray<double> vv = log(v[index]);       // vv[0]==log(v[3]), vv[1]==log(v[2]),
                                               // vv[2]==log(v[1]), vv[3]==log(v[0])
}
```

If an index is specified twice, we have referred to an element of a ***valarray*** twice in the same operation. That's exactly the kind of aliasing that ***valarray***s do not allow, so the behavior of an ***indirect_array*** is undefined if an index is repeated.

The ***indirect_array*** offers the same set of members as ***slice_array***. In particular, a an ***indirect_array*** cannot be constructed directly by the user and cannot be copied (§22.4.6). Instead, an ***indirect_array*** is the result of using a ***valarray<size_t>*** as the subscript of a ***valarray*** (§22.4.2). The number of elements of a ***valarray*** used as a subscript must not be greater than the number of elements of the ***valarray*** for which it is used as a subscript.

22.5 Complex Arithmetic [num.complex]

The standard library provides a ***complex*** template along the lines of the ***complex*** class described in §11.3. The library ***complex*** needs to be a template to serve the need for complex numbers based on different scalar types. In particular, specializations are provided for ***complex*** using ***float***, ***double***, and ***long double*** as its scalar type.

The ***complex*** template is defined in namespace ***std*** and presented in ***<complex>***:

```
template<class T> class std::complex {
    T re, im;
public:
    typedef T value_type;

    complex(const T& r = T(), const T& i = T()) : re(r), im(i) { }
    template<class X> complex(const complex<X>& a) : re(a.re), im(a.im) { }

    T real() const { return re; }
    T imag() const { return im; }

    complex<T>& operator=(const T& z); // assign complex(z,0)
    template<class X> complex<T>& operator=(const complex<X>&);
    // similarly: +=, –=, *=, /=
};
```

The representation and the inline functions are here for illustration. One could – barely – imagine a standard library ***complex*** that used a different representation. Note the use of member templates to ensure initialization and assignment of any ***complex*** type with any other (§13.6.2).

Throughout this book, I have used ***complex*** as a class rather than as a template. This is feasible because I assumed a bit of namespace magic to get the ***complex*** of ***double*** that I usually prefer:

```
typedef std::complex<double> complex;
```

The usual unary and binary operators are defined:

```
template<class T> complex<T> operator+(const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+(const complex<T>&, const T&);
template<class T> complex<T> operator+(const T&, const complex<T>&);

// similarly: –, *, /, ==, and !=

template<class T> complex<T> operator+(const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&);
```

The coordinate functions are provided:

```
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);

template<class T> complex<T> conj(const complex<T>&);

// construct from polar coordinates (abs(),arg()):
template<class T> complex<T> polar(const T& rho, const T& theta);


template<class T> T abs(const complex<T>&);        // sometimes called rho
template<class T> T arg(const complex<T>&);        // sometimes called theta

template<class T> T norm(const complex<T>&);       // square of abs()
```

The usual set of mathematical functions is provided:

```
template<class T> complex<T> sin(const complex<T>&);
//similarly: sinh, sqrt, tan, tanh, cos, cosh, exp, log, and log10

template<class T> complex<T> pow(const complex<T>&, int);
template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
template<class T> complex<T> pow(const T&, const complex<T>&);
```

Finally, stream I/O is provided:

```
template<class T, class Ch, class Tr>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, complex<T>&);
template<class T, class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const complex<T>&);
```

A complex is written out in the format `(x,y)` and can be read in the formats `x`, `(x)`, and `(x,y)` (§21.2.3, §21.3.5). The specializations ***complex<float>***, ***complex<double>***, and ***complex<long double>*** are provided to restrict conversions (§13.6.2) and to provide opportunities for optimized implementations. For example:

```
class complex<double> {
    double re, im;
public:
    typedef double value_type;

    complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
    complex(const complex<float>& a) : re(a.real()), im(a.imag()) { }
    explicit complex(const complex<long double>& a) : re(a.real()), im(a.imag()) { }

    // ...
};
```

Now a ***complex<float>*** can be quietly converted to a ***complex<double>***, while a ***complex< long double>*** can't. Similar specializations ensures that a ***complex<float>*** and a ***complex<double>*** can be quietly converted to a ***complex< long double>*** but that a ***complex< long double>*** cannot be implicitly converted to a ***complex<double>*** or to a ***complex<float>*** and a ***complex<double>*** cannot be implicitly converted to a ***complex<float>***. For example:

```
void f(complex<float> cf, complex<double> cd, complex<long double> cld)
{
    complex<double> c = cf;          // fine
    c = cd;                          // fine
    c = cld;                         // error: possible truncation
    c = complex<double>(cld);        // ok: you asked for truncation

    cf = cld;                        // error: possible truncation
    cf = cd;                         // error: possible truncation
    cf = complex<float>(cld);        // ok: you asked for truncation
    cf = complex<float>(cd);         // ok: you asked for truncation
}
```

22.6 Generalized Numeric Algorithms [num.general]

In *<numeric>*, the standard library provides a few generalized numeric algorithms in the style of the non-numeric algorithms from *<algorithm>* (Chapter 18) :

Generalized Numeric Algorithms <numeric>	
accumulate()	Accumulate results of operation on a sequence
inner_product()	Accumulate results of operation on two sequences
partial_sum()	Generate sequence by operation on a sequence
adjacent_difference()	Generate sequence by operation on a sequence

These algorithms generalize common operations such as computing a sum by letting them apply to all kinds of sequences and by making the operation applied to elements on those sequences a parameter. For each algorithm, the general version is supplemented by a version applying the most common operator for that algorithm.

22.6.1 Accumulate [num.accumulate]

The *accumulate*( ) algorithm can be understood as the generalization of a sum of the elements of a vector. The *accumulate*( ) algorithm is defined in namespace *std* and presented in *<numeric>*:

```
template <class In, class T> T accumulate(In first, In last, T init)
{
    while (first != last) init = init + *first++;   // plus
    return init;
}

template <class In, class T, class BinOp> T accumulate(In first, In last, T init, BinOp op)
{
    while (first != last) init = op(init, *first++);    // general operation
    return init;
}
```

The simple version of *accumulate*( ) adds elements of a sequence using their + operator. For example:

```
void f(vector<int>& price, list<float>& incr)
{
    int i = accumulate(price.begin(), price.end(), 0);    // accumulate in int
    double d = 0;
    d = accumulate(incr.begin(), incr.end(), d);          // accumulate in double
    // ...
}
```

Note how the type of the initial value passed determines the return type.

Not all items that we want to add are available as elements of a sequence. Where they are not, we can often supply an operation for *accumulate*( ) to call in order to produce the items to be added. The most obvious kind of operation to pass is one that extracts a value from a data structure. For example:

```
struct Record {
    // ...
    int unit_price;
    int number_of_units;
};

long price(long val, const Record& r)
{
    return val + r.unit_price * r.number_of_units;
}

void f(const vector<Record>& v)
{
    cout << "Total value: " << accumulate(v.begin(), v.end(), 0, price) << '\n';
}
```

Operations similar to ***accumulate*** are called ***reduce*** and ***reduction*** in some communities.

22.6.2 Inner_product [num.inner]

Accumulating from a sequence is very common, while accumulating from a pair of sequences is not uncommon. The ***inner_product*( )** algorithm is defined in namespace ***std*** and presented in *<numeric>*:

```
template <class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
{
    while (first != last) init = init + *first++ * *first2++;
    return init;
}

template <class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
    while (first != last) init = op(init, op2(*first++, *first2++));
    return init;
}
```

As usual, only the beginning of the second input sequence is passed as an argument. The second input sequence is assumed to be at least as long as the first.

The key operation in multiplying a ***Matrix*** by a ***valarray*** is an ***inner_product***:

```
valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
    valarray<double> res(m.dim1());

    for (int i=0; i<m.dim1(); i++) {
        Slice_iter<double>& ri = m.row(i);
        res(i) = inner_product(ri.begin(), ri.end(), &v[0], 0);
    }
    return res;
}
```

```
valarray<double> operator*(const valarray<double>& v, const Matrix& m)
{
    valarray<double> res(m.dim2());

    for (int j=0; j<m.dim2(); j++) {
        Slice_iter<double>& cj = m.column(j);
        res(j) = inner_product(&v[0], &v[v.size()], cj.begin(), 0);
    }
    return res;
}
```

Some forms of ***inner_product*** are often referred to as ‘‘dot product.’’

22.6.3 Incremental Change [num.incremental]

The ***partial_sum*()** and ***adjacent_difference*()** algorithms are inverses of each other and deal with the notion of incremental change. They are defined in namespace ***std*** and presented in *<numeric>*:

```
template <class In, class Out> Out adjacent_difference(In first, In last, Out res);

template <class In, class Out, class BinOp>
    Out adjacent_difference(In first, In last, Out res, BinOp op);
```

Given a sequence *a*, *b*, *c*, *d*, etc., ***adjacent_difference*()** produces *a*, *b-a*, *c-b*, *d-c*, etc.

Consider a vector of temperature readings. We could transform it into a vector of temperature changes like this:

```
vector<double> temps;

void f()
{
    adjacent_difference(temps.begin(), temps.end(), temps.begin());
}
```

For example, *17*, *19*, *20*, *20*, *17* turns into *17*, *2*, *1*, *0*, *-3*.

Conversely, ***partial_sum*()** allows us to compute the end result of a set of incremental changes:

```
template <class In, class Out, class BinOp>
Out partial_sum(In first, In last, Out res, BinOp op)
{
    if (first==last) return res;
    *res = *first;
    T val = *first;
    while (++first != last) {
        val = op(val, *first);
        *++res = val;
    }
    return ++res;
}
```

```
template <class In, class Out> Out partial_sum(In first, In last, Out res)
{
    return partial_sum(first, last, res, plus);    // §18.4.3
}
```

Given a sequence *a*, *b*, *c*, *d*, etc. , ***partial_sum*( )** produces *a*, *a+b*, *a+b+c*, *a+b+c+d*, etc. For example:

```
void f()
{
    partial_sum(temps.begin(), temps.end(), temps.begin());
}
```

Note the way ***partial_sum*( )** increments ***res*** before assigning a new value through it. This allows ***res*** to be the same sequence as its input; ***adjacent_difference*( )** behaves similarly. Thus,

```
partial_sum(v.begin(), v.end(), v.begin());
```

turns the sequence *a*, *b*, *c*, *d* into *a*, *a+b*, *a+b+c*, *a+b+c+d*, and

```
adjacent_difference(v.begin(), v.end(), v.begin());
```

turns it back into the original. In particular, ***partial_sum*( )** turns *17*, *2*, *1*, *0*, *-3* back into *17*, *19*, *20*, *20*, *17*.

For people who think of temperature differences as a boring detail of meteorology or science lab experiments, I point out that analyzing changes in stock prices involves exactly the same two operations.

22.7 Random Numbers [num.random]

Random numbers are essential to many simulations and games. In ***<cstdlib>*** and ***<stdlib.h>***, the standard library provides a simple basis for the generation of random numbers:

```
#define RAND_MAX implementation_defined /* large positive integer */

int rand();          // pseudo-random number between 0 and RAND_MAX
int srand(int i);    // seed random number generator by i
```

Producing a good random-number generator isn't easy, and unfortunately not all systems deliver a good ***rand*( )**. In particular, the low-order bits of a random number are often suspect, so ***rand*( )%*n*** is not a good portable way of generating a random number between *0* and *n-1*. Often, (*double*(*rand*( ))/*RAND_MAX*)**n* gives acceptable results.

A call of ***srand*( )** starts a new sequence of random numbers from the *seed* given as argument. For debugging, it is often important that a sequence of random numbers from a given seed be repeatable. However, we often want to start each real run with a new seed. In fact, to make games unpredictable, it is often useful to pick a seed from the environment of a program. For such programs, some bits from a real-time clock often make a good seed.

If you must write your own random-number generator, be sure to test it carefully (§22.9[14]).

A random-number generator is often more useful if represented as a class. In that way, random-number generators for different distributions are easily built:

```
class Randint {         // uniform distribution in the interval [0,max]
        unsigned long randx;
public:
        Randint(long s = 0) { randx=s; }
        void seed(long s) { randx=s; }

        // magic numbers chosen to use 31 bits of a 32-bit long:

        int abs(int x) { return x&0x7fffffff; }
        static double max() { return 2147483648.0; } // note: a double
        int draw() { return randx = randx*1103515245 + 12345; }

        double fdraw(){ return abs(draw())/max(); }

        int operator()() { return abs(draw()); }
};

class Urand : public Randint { // uniform distribution in the interval [0:n[
        int n;
public:
        Urand(int nn) { n = nn; }

        int operator()() { int r = n*fdraw(); return (r==n) ? n-1 : r; }
};

class Erand : public Randint { // exponential distribution random number generator
        int mean;
public:
        Erand(int m) { mean=m; }
        int operator()() { return -mean * log( (max()-draw())/max() + .5); }
};
```

Here is a simple test:

```
int main()
{
        Urand draw(10);
        map<int,int> bucket;
        for (int i = 0; i< 1000000; i++) bucket[draw()]++;
        for(int j = 0; j<10; j++) cout << bucket[j] << '\n';
}
```

Unless each bucket has approximately the value 10,000, there is a bug somewhere.

These random-number generators are slightly edited versions of what I shipped with the very first C++ library (actually, the first ''C with Classes'' library; §1.4).

22.8 Advice [num.advice]

[1] Numerical problems are often subtle. If you are not 100% certain about the mathematical aspects of a numerical problem, either take expert advice or experiment; §22.1.

[2] Use *numeric_limits* to determine properties of built-in types; §22.2.

[3] Specialize *numeric_limits* for user-defined scalar types; §22.2.

[4] Use ***valarray*** for numeric computation when run-time efficiency is more important than flexibility with respect to operations and element types; §22.4.
[5] Express operations on part of an array in terms of slices rather than loops; §22.4.6.
[6] Use compositors to gain efficiency through elimination of temporaries and better algorithms; §22.4.7.
[7] Use ***std::complex*** for complex arithmetic; §22.5.
[8] You can convert old code that uses a ***complex*** class to use the ***std::complex*** template by using a ***typedef***; §22.5.
[9] Consider ***accumulate***(), ***inner_product***(), ***partial_sum***(), and ***adjacent_difference***() before you write a loop to compute a value from a list; §22.6.
[10] Prefer a random-number class for a particular distribution over direct use of ***rand***(); §22.7.
[11] Be careful that your random numbers are sufficiently random; §22.7.

22.9 Exercises [num.exercises]

1. (∗1.5) Write a function that behaves like ***apply***() from §22.4.3, except that it is a nonmember function and accepts function objects.
2. (∗1.5) Write a function that behaves like ***apply***() from §22.4.3 , except that it is a nonmember function, accepts function objects, and modifies its ***valarray*** argument.
3. (∗2) Complete ***Slice_iter*** (§22.4.5). Take special care when defining the destructor.
4. (∗1.5) Rewrite the program from §17.4.1.3 using ***accumulate***().
5. (∗2) Implement I/O operators << and >> for ***valarray***. Implement a ***get_array***() function that creates a ***valarray*** of a size specified as part of the input itself.
6. (∗2.5) Define and implement a three-dimensional matrix with suitable operations.
7. (∗2.5) Define and implement an ***n***-dimensional matrix with suitable operations.
8. (∗2.5) Implement a ***valarray***-like class and implement + and * for it. Compare its performance to the performance of your C++ implementation's ***valarray***. Hint: Include ***x=0.5(x+y)-z*** among your test cases and try it with a variety of sizes for the vectors ***x***, ***y***, and ***z***.
9. (∗3) Implement a Fortran-style array ***Fort_array*** where indices start from ***1*** rather than ***0***.
10. (∗3) Implement ***Matrix*** using a ***valarray*** member as the representation of the elements (rather than a pointer or a reference to a ***valarray***).
11. (∗2.5) Use compositors (§22.4.7) to implement efficient multidimensional subscripting using the [] notation. For example, ***v1[x]***, ***v2[x][y]***, ***v2[x]***, ***v3[x][y][z]***, ***v3[x][y]***, and ***v3[x]*** should all yield the appropriate elements and subarrays using a simple calculation of an index.
12. (∗2) Generalize the idea from the program in §22.7 into a function that, given a generator as an argument, prints a simple graphical representation of its distribution that can be used as a crude visual check of the generator's correctness.
13. (∗1) If ***n*** is an ***int***, what is the distribution of (***double***(***rand***())/***RAND_MAX***)*n?
14. (∗2.5) Plot points in a square output area. The coordinate pairs for the points should be generated by ***Urand***(***N***), where ***N*** is the number of pixels on a side of the output area. What does the output tell you about the distribution of numbers generated by ***Urand***?
15. (∗2) Implement a Normal distribution generator, ***Nrand***.

Part IV

Design Using C++

This part presents C++ and the techniques it supports in the larger picture of software development. The focus is on design and the effective realization of design in terms of language constructs.

Chapters

23 Development and Design
24 Design and Programming
25 Roles of Classes

‘‘... I am just now beginning to discover the difficulty of expressing one’s ideas on paper. As long as it consists solely of description it is pretty easy; but where reasoning comes into play, to make a proper connection, a clearness & a moderate fluency, is to me, as I have said, a difficulty of which I had no idea ...’’

– Charles Darwin

23

Development and Design

There is no silver bullet.
– F. Brooks

Building software — aims and means — development process — development cycle — design aims — design steps — finding classes — specifying operations — specifying dependencies — specifying interfaces — reorganizing class hierarchies — models — experimentation and analysis — testing — software maintenance — efficiency — management — reuse — scale — the importance of individuals — hybrid design — bibliography — advice.

23.1 Overview [design.overview]

This chapter is the first of three that present the production of software in increasing detail, starting from a relatively high-level view of design and ending with C++ specific programming techniques and concepts directly supporting such design. After the introduction and a brief discussion of the aims and means of software development in §23.3, this chapter has two major parts:

§23.4 A view of the software development process

§23.5 Practical observations about the organization of software development

Chapter 24 discusses the relationship between design and programming language. Chapter 25 presents some roles that classes play in the organization of software from a design perspective. Taken as a whole, the three chapters of Part 4 aim to bridge the gap between would-be language-independent design and programming that is myopically focussed on details. Both ends of this spectrum have their place in a large project, but to avoid disaster and excessive cost, they must be part of a continuum of concerns and techniques.

23.2 Introduction [design.intro]

Constructing any nontrivial piece of software is a complex and often daunting task. Even for an individual programmer, the actual writing of program statements is only one part of the process. Typically, issues of problem analysis, overall program design, documentation, testing, and maintenance, as well as the management of all of this, dwarf the task of writing and debugging individual pieces of code. Naturally, one might simply label the totality of these activities ''programming'' and thereafter make a logically coherent claim that ''I don't design, I just program;'' but whatever one calls the activity, it is important sometimes to focus on its individual parts – just as it is important occasionally to consider the complete process. Neither the details nor the big picture must be permanently lost in the rush to get a system shipped – although often enough that is exactly what happens.

This chapter focusses on the parts of program development that do not involve writing and debugging individual pieces of code. The discussion is less precise and less detailed than the discussions of individual language features and specific programming techniques presented elsewhere in this book. This is necessary because there can be no cookbook method for creating good software. Detailed ''how to'' descriptions can exist for specific well-understood kinds of applications, but not for more general application areas. There is no substitute for intelligence, experience, and taste in programming. In consequence, this chapter offers only general advice, alternative approaches, and cautionary observations.

The discussion is hampered by the abstract nature of software and the fact that techniques that work for smaller projects (say, for one or two people writing 10,000 lines of code) do not necessarily scale to medium and large projects. For this reason, some discussions are formulated in terms of analogies from less abstract engineering disciplines rather than in terms of code examples. Please remember that ''proof by analogy'' is fraud, so analogy is used here for exposition only. Discussions of design issues phrased in C++ specific terms and with examples can be found in Chapter 24 and Chapter 25. The ideas expressed in this chapter are reflected in both the C++ language itself and in the presentation of the individual examples throughout this book.

Please also remember that because of the extraordinary diversity of application areas, people, and program-development environments, you cannot expect every observation made here to apply directly to your current problem. The observations are drawn from real-life projects and apply to a wide variety of situations, but they cannot be considered universal. Look at these observations with a healthy degree of skepticism.

C++ can be used simply as a better C. However, doing so leaves the most powerful techniques and language features unused so that only a small fraction of the potential benefits of using C++ will be gained. This chapter focusses on approaches to design that enable effective use of C++'s data abstraction and object-oriented programming facilities; such techniques are often called *object-oriented design*.

A few major themes run through this chapter:

- The most important single aspect of software development is to be clear about what you are trying to build.
- Successful software development is a long-term activity.
- The systems we construct tend to be at the limit of the complexity that we and our tools can handle.

- There are no ''cookbook'' methods that can replace intelligence, experience, and good taste in design and programming.
- Experimentation is essential for all nontrivial software development.
- Design and programming are iterative activities.
- The different phases of a software project, such as design, programming, and testing, cannot be strictly separated.
- Programming and design cannot be considered without also considering the management of these activities.

It is easy – and typically expensive – to underestimate any of these points. It is hard to transform the abstract ideas they embody into practice. The need for experience should be noted. Like boat building, bicycling, and programming, design is not a skill that can be mastered through theoretical study alone.

Too often, we forget the human aspects of system building and consider the software development process as simply ''a series of well-defined steps, each performing specific actions on inputs according to predefined rules to produce the desired outputs.'' The very language used conceals the human involvement! Design and programming are human activities; forget that and all is lost.

This chapter is concerned with the design of systems that are ambitious relative to the experience and resources of the people building the system. It seems to be the nature of individuals and organizations to attempt projects that are at the limits of their ability. Projects that don't offer such challenges don't need a discussion of design. Such projects already have established frameworks that need not be upset. Only when something ambitious is attempted is there a need to adopt new and better tools and procedures. There is also a tendency to assign projects that ''we know how to do'' to relative novices who don't.

There is no ''one right way'' to design and build all systems. I would consider belief in ''the one right way'' a childhood disease, if experienced programmers and designers didn't succumb to it so often. Please remember that just because a technique worked for you last year and for one project, it does not follow that it will work unmodified for someone else or for a different project. It is most important to keep an open mind.

Clearly, much of the discussion here relates to larger-scale software development. Readers who are not involved in such development can sit back and enjoy a look at the horrors they have escaped. Alternatively, they can look for the subset of the discussion that relates to individual work. There is no lower limit to the size of programs for which it is sensible to design before starting to code. There is, however, a lower limit for which any particular approach to design and documentation is appropriate. See §23.5.2 for a discussion of issues of scale.

The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: divide and conquer. A problem that can be separated into two sub-problems that can be handled separately is more than half solved by that separation. This simple principle can be applied in an amazing variety of ways. In particular, the use of a module or a class in the design of systems separates the program into two parts – the implementation and its users – connected only by an (ideally) well-defined interface. This is the fundamental approach to handling the inherent complexity of a program. Similarly, the process of designing a program can be broken into distinct activities with (ideally) well-defined interactions between the people involved. This is the basic approach to handling the inherent complexity of the development process and the people involved in it.

In both cases, the selection of the parts and the specification of the interfaces between the parts is where the most experience and taste is required. Such selection is not a simple mechanical process but typically requires insights that can be achieved only through a thorough understanding of a system at suitable levels of abstraction (see §23.4.2, §24.3.1, and §25.3). A myopic view of a program or of a software development process often leads to seriously flawed systems. Note also that for both people and programs, *separation* is easy. The hard part is to ensure effective *communication* between parties on different sides of a barrier without destroying the barrier or stifling the communication necessary to achieve cooperation.

This chapter presents an approach to design, not a complete design method. A complete formal design method is beyond the scope of this book. The approach presented here can be used with different degrees of formalization and as the basis for different formalizations. Similarly, this chapter is not a literature survey and does not attempt to touch every topic relevant to software development or to present every viewpoint. Again, that is beyond the scope of this book. A literature survey can be found in [Booch,1994]. Note that terms are used here in fairly general and conventional ways. Most ''interesting'' terms, such as *design*, *prototype*, and *programmer*, have several different and often conflicting definitions in the literature. Please be careful not to read something unintended into what is said here based on specialized or locally precise definitions of the terms.

23.3 Aims and Means [design.aims]

The purpose of professional programming is to deliver a product that satisfies its users. The primary means of doing so is to produce software with a clean internal structure and to grow a group of designers and programmers skilled enough and motivated enough to respond quickly and effectively to change and opportunities.

Why? The internal structure of the program and the process by which it was created are ideally of no concern to the end user. Stronger: if the end user has to worry about how the program was written, then there is something wrong with that program. Given that, what is the importance of the structure of a program and of the people who create the program?

A program needs a clean internal structure to ease:

- testing,
- porting,
- maintenance,
- extension,
- reorganization, and
- understanding.

The main point is that every successful major piece of software has an extended life in which it is worked on by a succession of programmers and designers, ported to new hardware, adapted to unanticipated uses, and repeatedly reorganized. Throughout the software's life, new versions of it must be produced with acceptable error rates and on time. Not planning for this is planning to fail.

Note that even though end users ideally don't have to know the internal structure of a system, they might actually want to. For example, a user might want to know the design of a system in detail to be able to assess its likely reliability and potential for revision and extension. If the software in question is not a complete system – rather, a set of libraries for building other software –

then the users will want to know more ‘‘details’’ to be able to better use the libraries and also to better benefit from them as sources of ideas.

A balance has to be struck between the lack of an overall design for a piece of software and overemphasis on structure. The former leads to endless cutting of corners (‘‘we’ll just ship this one and fix the problem in the next release’’). The latter leads to overelaborate designs in which essentials are lost in formalism and to situations where implementation gets delayed by program reorganizations (‘‘but this new structure is *much* better than the old one; people will want to wait for it’’). It also often results in systems so demanding of resources that they are unaffordable to most potential users. Such balancing acts are the most difficult aspects of design and the area in which talent and experience show themselves. The choices are hard for the individual designer or programmer and harder for the larger projects in which more people with differing skills are involved.

A program needs to be produced and maintained by an organization that can do this despite changes of personnel, direction, and management structure. A popular approach to coping with this problem has been to try to reduce system development into a few relatively low-level tasks slotted into a rigid framework. That is, the idea is to create a class of easy-to-train (cheap) and interchangeable low-level programmers (‘‘coders’’) and a class of somewhat less cheap but equally interchangeable (and therefore equally dispensable) designers. The coders are not supposed to make design decisions, while the designers are not supposed to concern themselves with the grubby details of coding. This approach often fails. Where it does work, it produces overly large systems with poor performance.

The problems with this approach are:

- insufficient communication between implementers and designers, which leads to missed opportunities, delays, inefficiencies, and repeated problems due to failure to learn from experience; and
- insufficient scope for initiative among implementers, which leads to lack of professional growth, lack of initiative, sloppiness, and high turnover.

Basically, such a system lacks feedback mechanisms to allow people to benefit from other people’s experience. It is wasteful of scarce human talent. Creating a framework within which people can utilize diverse talents, develop new skills, contribute ideas, and enjoy themselves is not just the only decent thing to do but also makes practical and economic sense.

On the other hand, a system cannot be built, documented, and maintained indefinitely without some form of formal structure. Simply finding the best people and letting them attack the problem as they think best is often a good start for a project requiring innovation. However, as the project progresses, more scheduling, specialization, and formalized communication between the people involved in the project become necessary. By ‘‘formal’’ I don’t mean a mathematical or mechanically verifiable notation (although that is nice, where available and applicable) but rather a set of guidelines for notation, naming, documentation, testing, etc. Again, a balance and a sense of appropriateness is necessary. A too-rigid system can prevent growth and stifle innovation. In this case, it is the manager’s talent and experience that is tested. For the individual, the equivalent dilemma is to choose where to try to be clever and where to simply ‘‘do it by the book.’’

The recommendation is to plan not just for the next release of the current project but also for the longer term. Looking only to the next release is planning to fail. We must develop organizations and software development strategies aimed at producing and maintaining many releases of many projects; that is, we must plan for a series of successes.

The purpose of ''design'' is to create a clean and relatively simple internal structure, sometimes also called an *architecture*, for a program. In other words, we want to create a framework into which the individual pieces of code can fit and thereby guide the writing of those individual pieces of code.

A design is the end product of the design process (as far as there is an *end* product of an iterative process). It is the focus of the communication between the designer and the programmer and between programmers. It is important to have a sense of proportion here. If I – as an individual programmer – design a small program that I'm going to implement tomorrow, the appropriate level of precision and detail may be some scribbles on the back of an envelope. At the other extreme, the development of a system involving hundreds of designers and programmers may require books of specifications carefully written using formal or semi-formal notations. Determining a suitable level of detail, precision, and formality for a design is in itself a challenging technical and managerial task.

In this and the following chapters, I assume that the design of a system is expressed as a set of class declarations (typically with their private declarations omitted as spurious details) and their relationships. This is a simplification. Many more issues enter into a specific design; for example, concurrency, management of namespaces, uses of nonmember function and data, parameterization of classes and functions, organization of code to minimize recompilation, persistence, and use of multiple computers. However, simplification is necessary for a discussion at this level of detail, and classes are the proper focus of design in the context of C++. Some of these other issues are mentioned in passing in this chapter, and some that directly affect the design of C++ programs are discussed in Chapter 24 and Chapter 25. For a more detailed discussion and examples of a specific object-oriented design method, see [Booch,1994].

I leave the distinction between analysis and design vague because a discussion of this issue is beyond the scope of this book and is sensitive to variations in specific design methods. It is essential to pick an analysis method to match the design method and to pick a design method to match the programming style and language used.

23.4 The Development Process [design.process]

Software development is an iterative and incremental process. Each stage of the process is revisited repeatedly during the development, and each visit refines the end products of that stage. In general, the process has no beginning and no end. When designing and implementing a system, you start from a base of other people's designs, libraries, and application software. When you finish, you leave a body of design and code for others to refine, revise, extend, and port. Naturally, a specific project can have a definite beginning and end, and it is important (though often surprisingly hard) to delimit the project cleanly and precisely in time and scope. However, pretending that you are starting from a clean slate can cause serious problems. Pretending that the world ends at the ''final delivery'' can cause equally serious problems for your successors (often yourself in a different role).

One implication of this is that the following sections could be read in any order because the aspects of design and implementation can be almost arbitrarily interleaved in a real project. That is, ''design'' is almost always redesign based on a previous design and some implementation

experience. Furthermore, the design is constrained by schedules, the skills of the people involved, compatibility issues, etc. A major challenge to a designer/manager/programmer is to create order in this process without stifling innovation and destroying the feedback loops that are necessary for successful development.

The development process has three stages:

- Analysis: defining the scope of the problem to be solved
- Design: creating an overall structure for a system
- Implementation: writing and testing the code

Please remember the iterative nature of this process – it is significant that these stages are not numbered. Note that some major aspects of program development don't appear as separate stages because they ought to permeate the process:

- Experimentation
- Testing
- Analysis of the design and the implementation
- Documentation
- Management

Software ''maintenance'' is simply more iterations through this development process (§23.4.6).

It is most important that analysis, design, and implementation don't become too detached from each other and that the people involved share a culture so that they can communicate effectively. In larger projects, this is all too often not the case. Ideally, individuals move from one stage to another during a project; the best way to transfer subtle information is in a person's head. Unfortunately, organizations often establish barriers against such transfers, for example, by giving designers higher status and/or higher pay than ''mere programmers.'' If it is not practical for people to move around to learn and teach, they should at least be encouraged to talk regularly with individuals involved in ''the other'' stages of the development.

For small-to-medium projects, there often is no distinction made between analysis and design; these two phases have been merged into one. Similarly, in small projects there often is no distinction made between design and programming. Naturally, this solves the communication problems. It is important to apply an appropriate degree of formality for a given project and to maintain an appropriate degree of separation between these phases (§23.5.2). There is no one right way to do this.

The model of software development described here differs radically from the traditional ''waterfall model.'' In a waterfall model, the development progresses in an orderly and linear fashion through the development stages from analysis to testing. The waterfall model suffers from the fundamental problem that information tends to flow only one way. When problems are found ''downstream,'' there is often strong methodological and organizational pressure to provide a local fix; that is, there is pressure to solve the problem without affecting the previous stages of the process. This lack of feedback leads to deficient designs, and the local fixes lead to contorted implementations. In the inevitable cases in which information does flow back toward the source and cause changes to the design, the result is a slow and cumbersome ripple effect through a system that is geared to prevent the need for such change and therefore unwilling and slow to respond. The argument for ''no change'' or for a ''local fix'' thus becomes an argument that one suborganization cannot impose large amounts of work on other suborganizations ''for its own convenience.'' In particular, by the time a major flaw is found there has often been so much paperwork generated

relating to the flawed decision that the effort involved in modifying the documentation dwarfs the effort needed to fix the code. In this way, paperwork can become the major problem of software development. Naturally, such problems can – and do – occur however one organizes the development of large systems. After all, *some* paperwork is essential. However, the pretense of a linear model of development (a waterfall) greatly increases the likelihood that this problem will get out of hand.

The problem with the waterfall model is insufficient feedback and the inability to respond to change. The danger of the iterative approach outlined here is a temptation to substitute a series of nonconverging changes for real thought and progress. Both problems are easier to diagnose than to solve, and however one organizes a task, it is easy and tempting to mistake activity for progress. Naturally, the emphasis on the different stages of the development process changes as a project progresses. Initially, the emphasis is on analysis and design, and programming issues receive less attention. As time passes, resources shift towards design and programming and then become more focussed on programming and testing. However, the key is never to focus on one part of the analysis/design/implementation spectrum to the exclusion of all other concerns.

Remember that no amount of attention to detail, no application of proper management technique, no amount of advanced technology can help you if you don't have a clear idea of what you are trying to achieve. More projects fail for lack of well-defined and realistic goals than for any other reason. Whatever you do and however you go about it, be clear about your aims, define tangible goals and milestones, and don't look for technological solutions to sociological problems. On the other hand, do use whatever *appropriate* technology is available – even if it involves an investment; people do work better with appropriate tools and in reasonable surroundings. Don't get fooled into believing that following this advice is easy.

23.4.1 The Development Cycle [design.cycle]

Developing a system should be an iterative activity. The main loop consists of repeated trips through this sequence:

[0] Examine the problem.
[1] Create an overall design.
[2] Find standard components.
  – Customize the components for this design.
[3] Create new standard components.
  – Customize the components for this design.
[4] Assemble the design.

As an analogy, consider a car factory. For a project to start, there needs to be an overall design for a new type of car. This first cut will be based on some kind of analysis and specifies the car in general terms related mostly to its intended use rather than to details of how to achieve desired properties. Deciding which properties are desirable – or even better, providing a relatively simple guide to deciding which properties are desirable – is often the hardest part of a project. When done well, this is typically the work of a single insightful individual and is often called a *vision*. It is quite common for projects to lack such clear goals – and for projects to falter or fail for that reason.

Say we want to build a medium-sized car with four doors and a fairly powerful engine. The first stage in the design is most definitely not to start designing the car (and all of its sub-

components) from scratch. A software designer or programmer in a similar circumstance might unwisely try exactly that.

The first stage is to consider which components are available from the factory's own inventory and from reliable suppliers. The components thus found need not be exactly right for the new car. There will be ways of customizing the components. It might even be possible to affect the specification of the ''next release'' of such components to make them more suitable for our project. For example, there may be an engine available with the right properties except for a slight deficiency in delivered power. Either we or the engine supplier might be able to add a turbocharger to compensate without affecting the basic design. Note that making such a change ''without affecting the basic design'' is unlikely unless the original design anticipated at least some form of customization. Such customization will typically require cooperation between you and your engine supplier. A software designer or programmer has similar options. In particular, polymorphic classes and templates can often be used effectively for customization. However, don't expect to be able to effect arbitrary extensions without foresight by or cooperation with the provider of such a class.

Having run out of suitable standard components, the car designer doesn't rush to design optimal new components for the new car. That would simply be too expensive. Assume that there were no suitable air conditioning unit available and that there was a suitable L-shaped space available in the engine compartment. One solution would be to design an L-shaped air conditioning unit. However, the probability that this oddity could be used in other car types – even after extensive customization – is low. This implies that our car designer will not be able to share the cost of producing such units with the designers of other car types and that the useful life of the unit will be short. It will thus be worthwhile to design a unit that has a wider appeal; that is, design a unit that has a cleaner design and is more suited for customization than our hypothetical L-shaped oddity. This will probably involve more work than the L-shaped unit and might even involve a modification of the overall design of our car to accommodate the more general-purpose unit. Because the new unit was designed to be more widely useful than our L-shaped wonder, it will presumably need a bit of customization to fit our revised needs perfectly. Again, the software designer or programmer has a similar option. That is, rather than writing project-specific code the designer can design a new component of a generality that makes it a good candidate to become a standard in some universe.

Finally, when we have run out of potential standard components we assemble the ''final'' design. We use as few specially designed widgets as possible because next year we will have to go through a variant of this exercise again for the next new model and the specially designed widgets will be the ones we most likely will have to redo or throw away. Sadly, the experience with traditionally designed software is that few parts of a system can even be recognized as discrete components, and few of those are of use outside their original project.

I'm not saying that all car designers are as rational as I have outlined in this analogy or that all software designers make the mistakes mentioned. On the contrary, this model can be made to work with software. In particular, this chapter and the next present techniques for making it work with C++. I do claim, however, that the intangible nature of software makes those mistakes harder to avoid (§24.3.1, §24.3.4), and in §23.5.3 I argue that corporate culture often discourages people from using the model outlined here.

Note that this model of development really works well only when you consider the longer term. If your horizon extends only to the next release, the creation and maintenance of standard components makes no sense. It will simply be seen as spurious overhead. This model is suggested for an

organization with a life that spans several projects and of a size that makes worthwhile the necessary extra investment in tools (for design, programming, and project management) and education (of designers, programmers, and managers). It is a sketch of a kind of software factory. Curiously enough, it differs only in scale from the practices of the best individual programmers, who over the years build up a stock of techniques, designs, tools, and libraries to enhance their personal effectiveness. It seems, in fact, that most organizations have failed to take advantage of the best personal practices due to both a lack of vision and an inability to manage such practices on more than a very small scale.

Note that it is unreasonable to expect ''standard components'' to be universally standard. There will exist a few international standard libraries. However, most components will be standard (only) within a country, an industry, a company, a product line, a department, an application area, etc. The world is simply too large for universal standards to be a realistic or indeed a to be desirable aim for all components and tools.

Aiming for universality in an initial design is a prescription for a project that will never be completed. One reason that the development cycle is a cycle is that it is essential to have a working system from which to gain experience (§23.4.3.6).

23.4.2 Design Aims [design.design]

What are the overall aims of a design? Simplicity is one, of course, but simplicity according to what criteria? We assume that a design will have to evolve. That is, the system will have to be extended, ported, tuned, and generally changed in a number of ways that cannot all be foreseen. Consequently, we must aim for a design and an implemented system that is simple under the constraint that it will be changed in many ways. In fact, it is realistic to assume that the requirements for the system will change several times between the time of the initial design and the first release of the system.

The implication is that the system must be designed to *remain* as simple as possible under a sequence of changes. We must design for change; that is, we must aim for

- flexibility,
- extensibility, and
- portability.

This is best done by trying to encapsulate the areas of a system that are likely to change and by providing non-intrusive ways for a later designer/programmer to modify the behavior of the code. This is done by identifying the key concepts of an application and giving each class the exclusive responsibility for the maintenance of all information relating to a single concept. In that case, a change can be effected by a modification of that class only. Ideally, a change to a single concept can be done by deriving a new class (§23.4.3.5) or by passing a different argument to a template. Naturally, this ideal is much easier to state than to follow.

Consider an example. In a simulation involving meteorological phenomena, we want to display a rain cloud. How do we do that? We cannot have a general routine to display the cloud because what a cloud looks like depends on the internal state of the cloud, and that state should be the sole responsibility of the cloud.

A first solution to this problem is to let the cloud display itself. This style of solution is acceptable in many limited contexts. However, it is not general because there are many ways to view a

cloud: for example, as a detailed picture, as a rough outline, or as an icon on a map. In other words, what a cloud looks like depends on both the cloud and its environment.

A second solution to the problem is to make the cloud aware of its environment and then let the cloud display itself. This solution is acceptable in even more contexts. However, it is still not a general solution. Having the cloud know about such details of its environment violates the dictum that a class is responsible for one thing only and that every ''thing'' is the responsibility of some class. It may not be possible to come up with a coherent notion of ''the cloud's environment'' because in general what a cloud looks like depends on both the cloud and the viewer. Even in real life, what the cloud looks like to me depends rather strongly on how I look at it; for example, with my naked eyes, through a polarizing filter, or with a weather radar. In addition to the viewer and the cloud, some ''general background'' such as the relative position of the sun might have to be taken into account. Adding other objects, such as other clouds and airplanes, further complicates the matter. To make life really hard for the designer, add the possibility of having several simultaneous viewers.

A third solution is to have the cloud – and other objects such as airplanes and the sun – describe themselves to a viewer. This solution has sufficient generality to serve most purposes†. It may, however, impose a significant cost in both complexity and run-time overhead. For example, how do we arrange for a viewer to understand the descriptions produced by clouds and other objects?

Rain clouds are not particularly common in programs (but for an example, see §15.2), but objects that need to be involved in a variety of I/O operations are. This makes the cloud example relevant to programs in general and to the design of libraries in particular. C++ code for a logically similar example can be found in the manipulators used for formatted output in the stream I/O system (§21.4.6, §21.4.6.3). Note that the third solution is not ''the right solution;'' it is simply the most general solution. A designer must balance the various needs of a system to choose the level of generality and abstraction that is appropriate for a given problem in a given system. As a rule of thumb, the right level of abstraction for a long-lived program is the most general you can comprehend and afford, *not* the absolutely most general. Generalization beyond the scope of a given project and beyond the experience of the people involved can be harmful; that is, it can cause delays, unacceptable inefficiencies, unmanageable designs, and plain failure.

To make such techniques manageable and economical, we must also design and manage for reuse (§23.5.1) and not completely forget about efficiency (§23.4.7).

23.4.3 Design Steps [design.steps]

Consider designing a single class. Typically, this is *not* a good idea. Concepts do *not* exist in isolation; rather, a concept is defined in the context of other concepts. Similarly, a class does not exist in isolation but is defined together with logically related classes. Typically, one works on a set of related classes. Such a set is often called a *class library* or a *component*. Sometimes all classes in a component constitute a single class hierarchy, sometimes they are members of a single namespace, and sometimes they are a more ad-hoc collection of declarations (§24.4).

† Even this model is unlikely to be sufficient for extreme cases like high-quality graphics based on ray tracing. I suspect that achieving such detail requires the designer to move to a different level of abstraction.

The set of classes in a component is united by some logical criteria, often by a common style and often by a reliance on common services. A component is thus the unit of design, documentation, ownership, and often reuse. This does not mean that if you use one class from a component, you must understand and use all the classes from the component or maybe get the code for every class in the component loaded into your program. On the contrary, we typically strive to ensure that a class can be used with only minimal overhead in machine resources and human effort. However, to use any part of a component we need to understand the logical criteria that define the component (hopefully made abundantly clear in the documentation), the conventions and style embodied in the design of the component and its documentation, and the common services (if any).

So consider how one might approach the design of a component. Because this is often a challenging task, it is worthwhile breaking it into steps to help focus on the various subtasks in a logical and complete way. As usual, there is no one right way of doing this. However, here is a series of steps that have worked for some people:

[1] Find the concepts/classes and their most fundamental relationships.

[2] Refine the classes by specifying the sets of operations on them.

- Classify these operations. In particular, consider the needs for construction, copying, and destruction.
- Consider minimalism, completeness, and convenience.

[3] Refine the classes by specifying their dependencies.

- Consider parameterization, inheritance, and use dependencies.

[4] Specify the interfaces.

- Separate functions into public and protected operations.
- Specify the exact type of the operations on the classes.

Note that these are steps in an iterative process. Typically, several loops through this sequence are needed to produce a design one can comfortably use for an initial implementation or a re-implementation. One advantage of well-done analysis and data abstraction as described here is that it becomes relatively easy to reshuffle class relationships even after code has been written. This is never a trivial task, though.

After that, we implement the classes and go back and review the design based on what was learned from implementing them. In the following subsections, I discuss these steps one by one.

23.4.3.1 Step 1: Find Classes [design.find]

Find the concepts/classes and their most fundamental relationships. The key to a good design is to model some aspect of ''reality'' directly – that is, capture the concepts of an application as classes, represent the relationships between classes in well-defined ways such as inheritance, and do this repeatedly at different levels of abstraction. But how do we go about finding those concepts? What is a practical approach to deciding which classes we need?

The best place to start looking is in the application itself, as opposed to looking in the computer scientist's bag of abstractions and concepts. Listen to someone who will become an expert user of the system once it has been built and to someone who is a somewhat dissatisfied user of the system being replaced. Note the vocabulary they use.

It is often said that the nouns will correspond to the classes and objects needed in the program; often that is indeed the case. However, that is by no means the end of the story. Verbs may denote

operations on objects, traditional (global) functions that produce new values based on the value of their arguments, or even classes. As examples of the latter, note the function objects (§18.4) and manipulators (§21.4.6). Verbs such as ''iterate'' or ''commit'' can be represented by an iterator object and an object representing a database commit operation, respectively. Even adjectives can often usefully be represented by classes. Consider the adjectives ''storable,'' ''concurrent,'' ''registered,'' and ''bounded.'' These may be classes intended to allow a designer or programmer to pick and choose among desirable attributes for later-designed classes by specifying virtual base classes (§15.2.4).

Not all classes correspond to application-level concepts. For example, some represent system resources and implementation-level abstractions (§24.3.1). It is also important to avoid modeling an old system too closely. For example, we don't want a system that is centered around a database to faithfully replicate aspects of a manual system that exist only to allow individuals to manage the physical shuffling of pieces of paper.

Inheritance is used to represent commonality among concepts. Most important, it is used to represent hierachical organization based on the behavior of classes representing individual concepts (§1.7, §12.2.6, §24.3.2). This is sometimes referred to as *classification* or even *taxonomy*. Commonality must be actively sought. Generalization and classification are high-level activities that require insight to give useful and lasting results. A common base should represent a more general concept rather than simply a similar concept that happens to require less data to represent.

Note that the classification should be of aspects of the concepts that we model in our system, rather than aspects that may be valid in other areas. For example, in mathematics a circle is a kind of an ellipse, but in most programs a circle should not be derived from an ellipse or an ellipse derived from a circle. The often-heard arguments ''because that's the way it is in mathematics'' and ''because the representation of a circle is a subset of that of an ellipse'' are not conclusive and most often wrong. This is because for most programs, the key property of a circle is that it has a center and a fixed distance to its perimeter. All behavior of a circle (all operations) must maintain this property (invariant; §24.3.7.1). On the other hand, an ellipse is characterized by two focal points that in many programs can be changed independently of each other. If those focal points coincide, the ellipse looks like a circle, but it is not a circle because its operations do not preserve the circle invariant. In most systems, this difference will be reflected by having a circle and an ellipse provide sets of operations that are not subsets of each other.

We don't just think up a set of classes and relationships between classes and use them for the final system. Instead, we create an initial set of classes and relationships. These are then refined repeatedly (§23.4.3.5) to reach a set of class relationships that are sufficiently general, flexible, and stable to be of real help in the further evolution of a system.

The best tool for finding initial key concepts/classes is a blackboard. The best method for their initial refinement is discussions with experts in the application domain and a couple of friends. Discussion is necessary to develop a viable initial vocabulary and conceptual framework. Few people can do that alone. One way to evolve a set of useful classes from an initial set of candidates is to simulate a system, with designers taking the roles of classes. This brings the inevitable absurdities of the initial ideas out into the open, stimulates discussion of alternatives, and creates a shared understanding of the evolving design. This activity can be supported by and documented by notes on index cards. Such cards are usually called CRC cards (''Class, Responsibility, and Collaborators''; [Wirfs-Brock,1990]) because of the information they record.

A *use case* is a description of a particular use of a system. Here is a simple example of a use case for a telephony system: take the phone off hook, dial a number, the phone at the other end rings, the phone at the other end is taken off hook. Developing a set of such use cases can be of immense value at all stages of development. Initially, finding use cases can help us understand what we are trying to build. During design, they can be used to trace a path through the system (for example, using CRC cards) to check that the relatively static description of the system in terms of classes and objects actually makes sense from a user's point of view. During programming and testing, the use cases become a source of test cases. In this way, use cases provide an orthogonal way of viewing the system and act as a reality check.

Use cases view the system as a (dynamic) working entity. They can therefore trap a designer into a functional view of a system and distract from the essential task of finding useful concepts that can be mapped into classes. Especially in the hands of someone with a background in structured analysis and weak experience with object-oriented programming/design, an emphasis on use cases can lead to a functional decomposition. A set of use cases is not a design. A focus on the use of the system must be matched by a complementary focus on the system's structure.

A team can become trapped into an inherently futile attempt to find and describe ***all*** of the use cases. This is a costly mistake. Much as when we look for candidate classes for a system, there comes a time when we must say, ''Enough is enough. The time has come to try out what we have and see what happens.'' Only by using a plausible set of classes and a plausible set of use cases in further development can we obtain the feedback that is essential to obtaining a good system. It is always hard to know when to stop a useful activity. It is especially hard to know when to stop when we know that we must return later to complete the task.

How many cases are enough? In general it is impossible to answer that question. However, in a given project, there comes a time when it is clear that most of the ordinary functioning of the system has been covered and a fair bit of the more unusual and error handling issues have been touched upon. Then it is time to get on with the next round of design and programming.

When you are trying to estimate the coverage of the system by a set of use cases, it can be useful to separate the cases into primary and secondary use cases. The primary ones describe the system's most common and ''normal'' actions, and the secondary describe the more unusual and error-handling scenarios. An example of a secondary use case would be a variant of the ''make a phone call'' case, in which the called phone is off hook, dialing its caller. It is often said that when 80% of the primary use cases and some of the secondary ones have been covered, it is time to proceed, but since we cannot know what constitutes ''all of the cases'' in advance, this is simply a rule of thumb. Experience and good sense matter here.

The concepts, operations, and relationships mentioned here are the ones that come naturally from our understanding of the application area or that arise from further work on the class structure. They represent our fundamental understanding of the application. Often, they are classifications of the fundamental concepts. For example, a hook-and-ladder is a fire engine, which is a truck, which is a vehicle. Sections §23.4.3.2 and §23.4.5 explain a few ways of looking at classes and class hierarchies with the view of making improvements.

Beware of viewgraph engineering! At some stage, you will be asked to present the design to someone and you will produce a set of diagrams explaining the structure of the system being built. This can be a very useful exercise because it helps focus your attention on what is important about the system and forces you to express your ideas in terms that others can understand. A presentation

is an invaluable design tool. Preparing a presentation with the aim of conveying real understanding to people with the interest and ability to produce constructive criticism is an exercise in conceptualization and clean expression of ideas.

However, a formal presentation of a design is also a very dangerous activity because there is a strong temptation to present an ideal system – a system you wished you could build, a system your high management wish they had – rather than what you have and what you might possibly produce in a reasonable time. When different approaches compete and executives don't really understand or care about ''the details,'' presentations can become lying competitions, in which the team that presents the most grandiose system gets to keep its job. In such cases, clear expression of ideas is often replaced by heavy jargon and acronyms. If you are a listener to such a presentation – and especially if you are a decision maker and you control development resources – it is desperately important that you distinguish wishful thinking from realistic planning. High-quality presentation materials are no guarantee of quality of the system described. In fact, I have often found that organizations that focus on the real problems get caught short when it comes to presenting their results compared to organizations that are less concerned with the production of real systems.

When looking for concepts to represent as classes, note that there are important properties of a system that cannot be represented as classes. For example, reliability, performance, and testability are important measurable properties of a system. However, even the most thoroughly object-oriented system will not have its reliability localized in a reliability object. Pervasive properties of a system can be specified, designed for, and eventually verified through measurement. Concern for such properties must be applied across all classes and may be reflected in rules for the design and implementation of individual classes and components (§23.4.3).

23.4.3.2 Step 2: Specify Operations [design.operations]

Refine the classes by specifying the sets of operations on them. Naturally, it is not possible to separate finding the classes from figuring out what operations are needed on them. However, there is a practical difference in that finding the classes focusses on the key concepts and deliberately de-emphasizes the computational aspects of the classes, whereas specifying the operations focusses on finding a complete and usable set of operations. It is most often too hard to consider both at the same time, especially since related classes should be designed together. When it is time to consider both together, CRC cards (§23.4.3.1) are often helpful.

In considering what functions are to be provided, several philosophies are possible. I suggest the following strategy:

[1] Consider how an object of the class is to be constructed, copied (if at all), and destroyed.

[2] Define the *minimal* set of operations required by the concept the class is representing. Typically, these operations become the member functions (§10.3).

[3] Consider which operations could be added for notational convenience. Include only a few really important ones. Often, these operations become the nonmember ''helper functions'' (§10.3.2).

[4] Consider which operations are to be virtual, that is, operations for which the class can act as an interface for an implementation supplied by a derived class.

[5] Consider what commonality of naming and functionality can be achieved across all the classes of the component.

This is clearly a statement of minimalism. It is far easier to add every function that could conceivably be useful and to make all operations virtual. However, the more functions, the more likely they are to remain unused and the more likely they are to constrain the implementation and the further evolution of the system. In particular, functions that directly read or write part of the state of an object of a class often constrain the class to a single implementation strategy and severely limit the potential for redesign. Such functions lower the level of abstraction from a concept to one implementation of it. Adding functions also causes more work for the implementer – and for the designer in the next redesign. It is *much* easier to add a function once the need for it has been clearly established than to remove it once it has become a liability.

The reason for requiring that the decision to make a function virtual be explicit rather than a default or an implementation detail is that making a function virtual critically affects the use of its class and the relationships between that class and other classes. Objects of a class with even a single virtual function have a nontrivial layout compared to objects in languages such as C and Fortran. A class with even a single virtual function potentially acts as the interface to yet-to-be-defined classes, and a virtual function implies a dependency on yet-to-be-defined classes (§24.3.2.1).

Note that minimalism requires more work from the designer, rather than less.

When choosing operations, it is important to focus on what is to be done rather than how it is to be done. That is, we should focus more on desired behavior than on implementation issues.

It is sometimes useful to classify operations on a class in terms of their use of the internal state of objects:

- Foundation operators: constructors, destructors and copy operators
- Inspectors: operations that do not modify the state of an object
- Modifiers: operations that do modify the state of an object
- Conversions: operations that produce an object of another type based on the value (state) of the object to which they are applied
- Iterators: operations that allow access to or use of a sequence of contained objects

These categories are not orthogonal. For example, an iterator can be designed to be either an inspector or a modifier. These categories are simply a classification that has helped people approach the design of class interfaces. Naturally, other classifications are possible. Such classifications are especially useful for maintaining consistency across a set of classes within a component.

C++ provides support for the distinction between inspectors and modifiers in the form of ***const*** and non-***const*** member functions. Similarly, the notions of constructors, destructors, copy operations, and conversion functions are directly supported.

23.4.3.3 Step 3: Specify Dependencies [design.dependencies]

Refine the classes by specifying their dependencies. The various dependencies are discussed in §24.3. The key ones to consider in the context of design are parameterization, *inheritance*, and *use* relationships. Each involves consideration of what it means for a class to be responsible for a single property of a system. To be responsible certainly doesn't mean that the class has to hold all the data itself or that its member functions have to perform all the necessary operations directly. On the contrary, each class having a single area of responsibility ensures that much of the work of a class is done by directing requests ''elsewhere'' for handling by some other class that has that particular subtask as its responsibility. However, be warned that overuse of this technique can lead to

inefficient and incomprehensible designs by proliferating classes and objects to the point where no work is done except by a cascade of forwarded requests for service. What *can* be done here and now, should be.

The need to consider inheritance and use relationships at the design stage (and not just during implementation) follows directly from the use of classes to represent concepts. It also implies that the component (§23.4.3, §24.4), and not the individual class, is the unit of design.

Parameterization – often leading to the use of templates – is a way of making implicit dependencies explicit so that several alternatives can be represented without adding new concepts. Often, there is a choice between leaving something as a dependency on a context, representing it as a branch of an inheritance tree, or using a parameter (§24.4.1).

23.4.3.4 Step 4: Specify Interfaces [design.interfaces]

Specify the interfaces. Private functions don't usually need to be considered at the design stage. What implementation issues must be considered in the design stage are best dealt with as part of the consideration of dependencies in Step 2. Stronger: I use as a rule of thumb that unless at least two significantly different implementations of a class are possible, then there is probably something wrong with the class. That is, it is simply an implementation in disguise and not a representation of a proper concept. In many cases, considering if some form of lazy evaluation is feasible for a class is a good way of approaching the question, ''Is the interface to this class sufficiently implementation-independent?''

Note that public bases and friends are part of the public interface of a class; see also §11.5 and §24.4.2. Providing separate interfaces for inheriting and general clients by defining separate protected and public interfaces can be a rewarding exercise.

This is the step where the exact types of arguments are considered and specified. The ideal is to have as many interfaces as possible statically typed with application-level types; see §24.2.3 and §24.4.2.

When specifying the interfaces, look out for classes where the operations seem to support more than one level of abstraction. For example, some member functions of a class ***File*** may take arguments of type ***File_descriptor*** and others string arguments that are meant to be file names. The ***File_descriptor*** operations operate on a different level of abstraction than do the file name operations, so one must wonder whether they belong in the same class. Maybe it would be better to have two file classes, one supporting the notion of a file descriptor and another supporting the notion of a file name. Typically, all operations on a class should support the same level of abstraction. When they don't, a reorganization of the class and related classes should be considered.

23.4.3.5 Reorganization of Class Hierarchies [design.hier]

In Step 1 and again in Step 3, we examine the classes and class hierarchies to see if they adequately serve our needs. Typically they don't, and we have to reorganize to improve that structure or a design and/or an implementation.

The most common reorganizations of a class hierarchy are factoring the common part of two classes into a new class and splitting a class into two new ones. In both cases, the result is three classes: a base class and two derived classes. When should such reorganizations be done? What are common indicators that such a reorganization might be useful?

Unfortunately, there are no simple, general answers to such questions. This is not really surprising because what we are talking about are not minor implementation details, but changes to the basic concepts of a system. The fundamental – and nontrivial – operation is to look for commonality between classes and factor out the common part. The exact criteria for commonality are undefined but should reflect commonality in the concepts of the system, not just implementation conveniences. Clues that two or more classes have commonality that might be factored out into a common base class are common patterns of use, similarity of sets of operations, similarity of implementations, and simply that these classes often turn up together in design discussions. Conversely, a class might be a good candidate for splitting into two if subsets of the operations of that class have distinct usage patterns, if such subsets access separate subsets of the representation, and if the class turns up in apparently unrelated design discussions. Sometimes, making a set of related classes into a template is a way of providing necessary alternatives in a systematic manner (§24.4.1).

Because of the close relationship between classes and concepts, problems with the organization of a class hierarchy often surface as problems with the naming of classes and the use of class names in design discussions. If design discussion using class names and the classification implied by the class hierarchies sounds awkward, then there is probably an opportunity to improve the hierarchies. Note that I'm implying that two people are much better at analyzing a class hierarchy than is one. Should you happen to be without someone with whom to discuss a design, then writing a tutorial description of the design using the class names can be a useful alternative.

One of the most important aims of a design is to provide interfaces that can remain stable in the face of changes (§23.4.2). Often, this is best achieved by making a class on which many classes and functions depend into an abstract class presenting very general operations. Details are best relegated to more specialized derived classes on which fewer classes and functions directly depend. Stronger: the more classes that depend on a class, the more general that class should be and the fewer details it should reveal.

There is a strong temptation to add operations (and data) to a class used by many. This is often seen as a way of making that class more useful and less likely to need (further) change. The effect of such thinking is a class with a fat interface (§24.4.3) and with data members supporting several weakly related functions. This again implies that the class must be modified whenever there is a significant change to one of the many classes it supports. This, in turn, implies changes to apparently unrelated user classes and derived classes. Instead of complicating a class that is central to a design, we should usually keep it general and abstract. When necessary, specialized facilities should be presented as derived classes. See [Martin,1995] for examples.

This line of thought leads to hierarchies of abstract classes, with the classes near the roots being the most general and having the most other classes and functions dependent on them. The leaf classes are the most specialized and have only very few pieces of code depending directly on them. As an example, consider the final version of the ***Ival_box*** hierarchy (§12.4.3, §12.4.4).

23.4.3.6 Use of Models [design.model]

When I write an article, I try to find a suitable model to follow. That is, rather than immediately starting to type I look for papers on a similar topic to see if I can find one that can be an initial pattern for my paper. If the model I choose is a paper I wrote myself on a related topic, I might even be able to leave parts of the text in place, modify other parts as needed, and add new information

only where the logic of the information I'm trying to convey requires it. For example, this book is written that way based on its first and second editions. An extreme form of this writing technique is the form letter. In that case, I simply fill in a name and maybe add a few lines to ''personalize'' the letter. In essence, I'm writing such letters by specifying the differences from a basic model.

Such use of existing systems as models for new designs is the norm rather than the exception in all forms of creative endeavors. Whenever possible, design and programming should be based on previous work. This limits the degrees of freedom that the designer has to deal with and allows attention to be focussed on a few issues at a time. Starting a major project ''completely from scratch'' can be exhilarating. However, often a more accurate description is ''intoxicating'' and the result is a drunkard's walk through the design alternatives. Having a model is not constraining and does not require that the model should be slavishly followed; it simply frees the designer to consider one aspect of a design at a time.

Note that the use of models is inevitable because any design will be synthesized from the experiences of its designers. Having an explicit model makes the choice of a model a conscious decision, makes assumptions explicit, defines a common vocabulary, provides an initial framework for the design, and increases the likelihood that the designers have a common approach.

Naturally, the choice of an initial model is in itself an important design decision and often can be made only after a search for potential models and careful evaluation of alternatives. Furthermore, in many cases a model is suitable only with the understanding that major modification is necessary to adapt the ideas to a particular new application. Software design is hard, and we need all the help we can get. We should not reject the use of models out of misplaced disdain for ''imitation.'' Imitation is the sincerest form of flattery, and the use of models and previous work as inspiration is – within the bounds of propriety and copyright law – acceptable technique for innovative work in all fields: what was good enough for Shakespeare is good enough for us. Some people refer to such use of models in design as ''design reuse.''

Documenting general elements that turn up in many designs together with some description of the design problem they solve and the conditions under which they can be used is an obvious idea – at least once you think of it. The word *pattern* is often used to describe such a general and useful design element, and a literature exists documenting patterns and their use (for example, [Gamma,1994] and [Coplien,1995]).

It is a good idea for a designer to be acquainted with popular patterns in a given application domain. As a programmer, I prefer patterns that have some code associated with them as concrete examples. Like most people, I understand a general idea (in this case, a pattern) best when I have a concrete example (in this case, a piece of code illustrating a use of the pattern) to help me. People who use patterns heavily have a specialized vocabulary to ease communication among themselves. Unfortunately, this can become a private language that effectively excludes outsiders from understanding. As always, it is essential to ensure proper communication among people involved in different parts of a project (§23.3) and also with the design and programming communities at large.

Every successful large system is a redesign of a somewhat smaller working system. I know of no exceptions to this rule. The closest I can think of are projects that failed, muddled on for years at great cost, and then eventually became successes years after their intended completion date. Such projects unintentionally – and often unacknowledged – simply first built a nonworking system, then transformed that into a working system, and finally redesigned that into a system that approximated the original aims. This implies that it is a folly to set out to build a large system from

scratch exactly right according to the latest principles. The larger and the more ambitious a system we aim for, the more important it is to have a model from which to work. For a large system, the only really acceptable model is a somewhat smaller, related *working* system.

23.4.4 Experimentation and Analysis [design.experiment]

At the start of an ambitious development project, we do not know the best way to structure the system. Often, we don't even know precisely what the system should do because particulars will become clear only through the effort of building, testing, and using the system. How – short of building the complete system – do we get the information necessary to understand what design decisions are significant and to estimate their ramifications?

We conduct experiments. Also, we analyze the design and implementation as soon as we have something to analyze. Most frequently and importantly, we discuss the design and implementation alternatives. In all but the rarest cases, design is a social activity in which designs are developed through presentations and discussions. Often, the most important design tool is a blackboard; without it, the embryonic concepts of a design cannot be developed and shared among designers and programmers.

The most popular form of experiment seems to be to build a prototype, that is, a scaled-down version of the system or a part of the system. A prototype doesn't have stringent performance criteria, machine and programming-environment resources are typically ample, and the designers and programmers tend to be uncommonly well educated, experienced, and motivated. The idea is to get a version running as fast as possible to enable exploration of design and implementation choices.

This approach can be very successful when done well. It can also be an excuse for sloppiness. The problem is that the emphasis of a prototype can easily shift from ''exploring design alternatives'' to ''getting some sort of system running as soon as possible.'' This easily leads to a disinterest in the internal structure of the prototype (''after all, it is only a prototype'') and a neglect of the design effort in favor of playing around with the prototype implementation. The snag is that such an implementation can degenerate into the worst kind of resource hog and maintenance nightmare while giving the illusion of an ''almost complete'' system. Almost by definition, a prototype does not have the internal structure, the efficiency, and the maintenance infrastructure that allows it to scale to real use. Consequently, a ''prototype'' that becomes an ''almost product'' soaks up time and energy that could have been better spent on the product. The temptation for both developers and managers is to make the prototype into a product and postpone ''performance engineering'' until the next release. Misused this way, prototyping is the negation of all that design stands for.

A related problem is that the prototype developers can fall in love with their tools. They can forget that the expense of their (necessary) convenience cannot always be afforded by a production system and that the freedom from constraints and formalities offered by their small research group cannot easily be maintained for a larger group working toward a set of interlocking deadlines.

On the other hand, prototypes can be invaluable. Consider designing a user interface. In this case, the internal structure of the part of the system that doesn't interact directly with the user often *is* irrelevant and there are no other feasible ways of getting experience with users' reactions to the look and feel of a system. Another example is a prototype designed strictly for studying the internal workings of a system. Here, the user interface can be rudimentary – possibly with simulated users instead of real ones.

Prototyping is a way of experimenting. The desired results from building a prototype are the insights that building it brings, not the prototype itself. Maybe the most important criterion for a prototype is that it has to be so incomplete that it is obviously an experimental vehicle and cannot be turned into a product without a major redesign and reimplementation. Having a prototype ''incomplete'' helps keep the focus on the experiment and minimizes the danger of having the prototype become a product. It also minimizes the temptation to try to base the design of the product too closely on the design of the prototype – thus forgetting or ignoring the inherent limitations of the prototype. After use, a prototype should be thrown away.

It should be remembered that in many cases, there are experimental techniques that can be used as alternatives to prototyping. Where those can be used, they are often preferable because of their greater rigor and lower demands on designer time and system resources. Examples are mathematical models and various forms of simulators. In fact, one can see a continuum from mathematical models, through more and more detailed simulations, through prototypes, through partial implementations, to a complete system.

This leads to the idea of growing a system from an initial design and implementation through repeated redesign and reimplementation. This is the ideal strategy, but it can be very demanding on design and implementation tools. Also, the approach suffers from the risk of getting burdened with so much code reflecting initial design decisions that a better design cannot be implemented. At least for now, this strategy seems limited to small-to-medium-scale projects, in which major changes to the overall design are unlikely, and for redesigns and reimplementations after the initial release of the system, where such a strategy is inevitable.

In addition to experiments designed to provide insights into design choices, analysis of a design and/or an implementation itself can be an important source of further insights. For example, studies of the various dependencies between classes (§24.3) can be most helpful, and traditional implementer's tools such as call graphs, performance measurements, etc., must not be ignored.

Note that specifications (the output of the analysis phase) and designs are as prone to errors as is the implementation. In fact, they may be more so because they are even less concrete, are often specified less precisely, are not executable, and typically are not supported by tools of a sophistication comparable to what is available for checking and analyzing the implementation. Increasing the formality of the language/notation used to express a design can go some way toward enabling the application of tools to help the designer. This must not be done at the cost of impoverishing the programming language used for implementation (§24.3.1). Also, a formal notation can itself be a source of complexity and problems. This happens when the formalism is ill suited to the practical problem to which it is applied, when the rigor of the formalism exceeds the mathematical background and maturity of the designers and programmers involved, and when the formal description of a system gets out of touch with the system it is supposedly describing.

Design is inherently error-prone and hard to support with effective tools. This makes experience and feedback essential. Consequently, it is fundamentally flawed to consider the software-development process a linear process starting with analysis and ending with testing. An emphasis on iterative design and implementation is needed to gain sufficient feedback from experience during the various stages of development.

23.4.5 Testing [design.test]

A program that has not been tested does not work. The ideal of designing and/or verifying a program so that it works the first time is unattainable for all but the most trivial programs. We should strive toward that ideal, but we should not be fooled into thinking that testing is easy.

‘‘How to test?’’ is a question that cannot be answered in general. ‘‘When to test?’’ however, does have a general answer: as early and as often as possible. Test strategies should be generated as part of the design and implementation efforts or at least should be developed in parallel with them. As soon as there is a running system, testing should begin. Postponing serious testing until ‘‘after the implementation is complete’’ is a prescription for slipped schedules and/or flawed releases.

Wherever possible, a system should be designed specifically so that it is relatively easy to test. In particular, mechanisms for testing can often be designed right into the system. Sometimes this is not done out of fear of causing expensive run-time testing or for fear that the redundancy necessary for consistency checks will unduly enlarge data structures. Such fear is usually misplaced because most actual testing code and redundancy can, if necessary, be stripped out of the code before the system is shipped. Assertions (§24.3.7.2) are sometimes useful here.

More important than specific tests is the idea that the structure of the system should be such that we have a reasonable chance of convincing ourselves and our users/customers that we can eliminate errors by a combination of static checking, static analysis, and testing. Where a strategy for fault tolerance is developed (§14.9), a testing strategy can usually be designed as a complementary and closely related aspect of the total design.

If testing issues are completely discounted in the design phase, then testing, delivery date, and maintenance problems will result. The class interfaces and the class dependencies (as described in §24.3 and §24.4.2) are usually a good place to start work on a testing strategy.

Determining how much testing is enough is usually hard. However, too little testing is a more common problem than too much. Exactly how many resources should be allocated to testing compared to design and implementation naturally depends on the nature of the system and the methods used to construct it. However, as a rule of thumb, I can suggest that more resources in time, effort, and talent should be spent testing a system than on constructing the initial implementation. Testing should focus on problems that would have disastrous consequences and on problems that would occur frequently.

23.4.6 Software Maintenance [design.maintain]

‘‘Software maintenance’’ is a misnomer. The word ‘‘maintenance’’ suggests a misleading analogy to hardware. Software doesn’t need oiling, doesn’t have moving parts that wear down, and doesn’t have crevices in which water can collect and cause rust. Software can be replicated *exactly* and transported over long distances at minute costs. Software is not hardware.

The activities that go under the name of software maintenance are really redesign and reimplementation and thus belong under the usual program development cycle. When flexibility, extensibility, and portability are emphasized in the design, the traditional sources of maintenance problems are addressed directly.

Like testing, maintenance must not be an afterthought or an activity segregated from the mainstream of development. In particular, it is important to have some continuity in the group of people

involved in a project. It is not easy to successfully transfer maintenance to a new (and typically less-experienced) group of people with no links to the original designers and implementers. When a major change of people is necessary, there must be an emphasis on transferring an understanding of the system's structure and of the system's aims to the new people. If a ''maintenance crew'' is left guessing about the architecture of the system or must deduce the purpose of system components from their implementation, the structure of a system can deteriorate rapidly under the impact of local patches. Documentation is typically much better at conveying details than in helping new people to understand key ideas and principles.

23.4.7 Efficiency [design.efficiency]

Donald Knuth observed that ''premature optimization is the root of all evil.'' Some people have learned that lesson all too well and consider all concern for efficiency evil. On the contrary, efficiency must be kept in mind throughout the design and implementation effort. However, that does not mean the designer should be concerned with micro-efficiencies, but that first-order efficiency issues must be considered.

The best strategy for efficiency is to produce a clean and simple design. Only such a design can remain relatively stable over the lifetime of the project and serve as a base for performance tuning. Avoiding the gargantuanism that plagues large projects is essential. Far too often people add features ''just in case'' (§23.4.3.2, §23.5.3) and end up doubling and quadrupling the size and runtime of systems to support frills. Worse, such overelaborate systems are often unnecessarily hard to analyze so that it becomes difficult to distinguish the avoidable overheads from the unavoidable. Thus, even basic analysis and optimization is discouraged. Optimization should be the result of analysis and performance measurement, not random fiddling with the code. Especially in larger systems, a designer's or programmer's ''intuition'' is an unreliable guide in matters of efficiency.

It is important to avoid inherently inefficient constructs and constructs that will take much time and cleverness to optimize to an acceptable performance level. Similarly, it is important to minimize the use of inherently nonportable constructs and tools because using such tools and constructs condemns the project to run on older (less powerful and/or more expensive) computers.

23.5 Management [design.management]

Provided it makes some minimum of sense, most people do what they are encouraged to do. In particular, if in the context of a software project you reward certain ways of operating and penalize others, only exceptional programmers and designers will risk their careers to do what they consider right in the face of management opposition, indifference, and red tape†. It follows that an organization should have a reward structure that matches its stated aims of design and programming. However, all too often this is not the case: a major change of programming style can be achieved only through a matching change of design style, and both typically require changes in management style to be effective. Mental and organizational inertia all too easily leads to a local change that is not

† An organization that treats its programmers as morons will soon have programmers that are willing and able to act like morons only.

supported by global changes required to ensure its success. A fairly typical example is a change to a language that supports object-oriented programming, such as C++, without a matching change in the design strategies to take advantage of its facilities (see also §24.2). Another is a change to ''object-oriented design'' without the introduction of a programming language to support it.

23.5.1 Reuse [design.reuse]

Increased reuse of code and design is often cited as a major reason for adopting a new programming language or design strategy. However, most organizations reward individuals and groups that choose to re-invent the wheel. For example, a programmer may have his productivity measured in lines of code; will he produce small programs relying on standard libraries at the cost of income and, possibly, status? A manager may be paid somewhat proportionally to the number of people in her group; is she going to use software produced in another group when she can hire another couple of programmers for her own group instead? A company can be awarded a government contract, where the profit is a fixed percentage of the development cost; is that company going to minimize its profits by using the most effective development tools? Rewarding reuse is hard, but unless management finds ways to encourage and reward it, reuse will not happen.

Reuse is primarily a social phenomenon. I can use someone else's software provided that:

[1] It works: to be reusable, software must first be usable.

[2] It is comprehensible: program structure, comments, documentation, and tutorial material are important.

[3] It can coexist with software not specifically written to coexist with it.

[4] It is supported (or I'm willing to support it myself; typically, I'm not).

[5] It is economical (can I share the development and maintenance costs with other users?).

[6] I can find it.

To this, we may add that a component is not reusable until someone has ''reused'' it. The task of fitting a component into an environment typically leads to refinements in its operation, generalizations of its behavior, and improvements in its ability to coexist with other software. Until this exercise has been done at least once, even components that have been designed and implemented with the greatest care tend to have unintended and unexpected rough corners.

My experience is that the conditions necessary for reuse will exist only if someone makes it their business to make such sharing work. In a small group, this typically means that an individual, by design or by accident, becomes the keeper of common libraries and documentation. In a larger organization, this means that a group or department is chartered to gather, build, document, popularize, and maintain software for use by many groups.

The importance of such a ''standard components'' group cannot be overestimated. Note that as a first approximation, a system reflects the organization that produced it. If an organization has no mechanism for promoting and rewarding cooperation and sharing, cooperation and sharing will be rare. A standard components group must actively promote its components. This implies that good traditional documentation is essential but insufficient. In addition, the components group must provide tutorials and other information that allow a potential user to find a component and understand why it might be of help. This implies that activities that traditionally are associated with marketing and education must be undertaken by the components group.

Whenever possible, the members of this group should work in close cooperation with applications builders. Only then can they be sufficiently aware of the needs of users and alert to the opportunities for sharing components among different applications. This argues for there to be a consultancy role for such an organization and for the use of internships to transfer information into and out of the components group.

The success of a ''components group'' must be measured in terms of the success of its clients. If its success is measured simply in terms of the amount of tools and services it can convince development organizations to accept, such a group can become corrupted into a mere peddler of commercial software and a proponent of ever-changing fads.

Not all code needs to be reusable, and reusability is not a universal property. Saying that a component is ''reusable'' means that its reuse within a certain framework requires little or no work. In most cases, moving to a different framework will require significant work. In this respect, reuse strongly resembles portability. It is important to note that reuse is the result of design aimed at reuse, refinement of components based on experience, and deliberate effort to search out existing components to (re)use. Reuse does not magically arise from mindless use of specific language features or coding techniques. C++ features such as classes, virtual functions, and templates allow designs to be expressed so that reuse is made easier (and thus more likely), but in themselves such features do not ensure reusability.

23.5.2 Scale [design.scale]

It is easy for an individual or an organization to get excited about ''doing things right.'' In an institutional setting, this often translates into ''developing and strictly following proper procedures.'' In both cases, common sense can be the first victim of a genuine and often ardent desire to improve the way things are done. Unfortunately, once common sense is missing there is no limit to the damage that can unwittingly be done.

Consider the stages of the development process listed in §23.4 and the stages of the design steps listed in §23.4.3. It is relatively easy to elaborate these stages into a proper design method where each stage is more precisely defined and has well-defined inputs and outputs and a semiformal notation for expressing these inputs and outputs. Checklists can be developed to ensure that the design method is adhered to, and tools can be developed to enforce a large number of the procedural and notational conventions. Further, looking at the classification of dependencies presented in §24.3 one could decree that certain dependencies were good and others bad and provide analysis tools to ensure that these value judgements were applied uniformly across a project. To complete this ''firming up'' of the software-production process, one would define standards for documentation (including rules for spelling and grammar and typesetting conventions) and for the general look of the code (including specifications of which language features can and cannot be used, specifications of what kinds of libraries can and cannot be used, conventions for indentation and the naming of functions, variables, and types, etc.).

Much of this can be helpful for the success of a project. At least, it would be a folly to set out to design a system that will eventually contain ten million lines of code that will be developed by hundreds of people and maintained and supported by thousands more over a decade or more without a fairly well-defined and somewhat rigid framework along the lines described previously.

Fortunately, most systems do not fall into this category. However, once the idea is accepted that such a design method or adherence to such a set of coding and documentation standards is ‘‘the right way,’’ pressure builds to apply it universally and in every detail. This can lead to ludicrous constraints and overheads on small projects. In particular, it can lead to paper shuffling and forms filling replacing productive work as the measure of progress and success. If that happens, real designers and programmers will leave the project and be replaced with bureaucrats.

Once such a ridiculous misapplication of a (hopefully perfectly reasonable) design method has occurred in a community, its failure becomes the excuse for avoiding almost all formality in the development process. This in turn naturally leads to the kind of messes and failures that the design method was designed to prevent in the first place.

The real problem is to find an appropriate degree of formality for the development of a particular project. Don’t expect to find an easy answer to this problem. Essentially every approach works for a small project. Worse, it seems that essentially every approach – however ill conceived and however cruel to the individuals involved – also works for a large project, provided you are willing to throw indecent amounts of time and money at the problem.

A key problem in every software project is how to maintain the integrity of the design. This problem increases more than linearly with scale. Only an individual or a small group of people can grasp and keep sight of the overall aims of a major project. Most people must spend so much of their time on subprojects, technical details, day-to-day administration, etc., that the overall design aims are easily forgotten or subordinated to more local and immediate goals. It also is a recipe for failure not to have an individual or group with the explicit task of maintaining the integrity of the design. It is a recipe for failure not to enable such an individual or group to have an effect on the project as a whole.

Lack of a consistent long-term aim is much more damaging to a project and an organization than the lack of any individual feature. It should be the job of some small number of individuals to formulate such an overall aim, to keep that aim in mind, to write the key overall design documents, to write the introductions to the key concepts, and generally to help others to keep the overall aim in mind.

23.5.3 Individuals [design.people]

Use of design as described here places a premium on skillful designers and programmers. Thus, it makes the choice of designers and programmers critical to the success of an organization.

Managers often forget that organizations consist of individuals. A popular notion is that programmers are equal and interchangeable. This is a fallacy that can destroy an organization by driving out many of the most effective individuals and condemning the remaining people to work at levels well below their potential. Individuals are interchangeable only if they are not allowed to take advantage of skills that raise them above the absolute minimum required for the task in question. Thus, the fiction of interchangeability is inhumane and inherently wasteful.

Most programming performance measures encourage wasteful practices and fail to take critical individual contributions into account. The most obvious example is the relatively widespread practice of measuring progress in terms of number of lines of code produced, number of pages of documentation produced, number of tests passed, etc. Such figures look good on management charts but bear only the most tenuous relation to reality. For example, if productivity is measured in terms

of number of lines of code produced, a successful application of reuse will appear to cause negative performance of programmers. A successful application of the best principles in the redesign of a major piece of software typically has the same effect.

Quality of work produced is far harder to measure than quantity of output, yet individuals and groups must be rewarded based on the quality of their output rather than by crude quantity measures. Unfortunately, the design of practical quality measures has – to the best of my knowledge – hardly begun. In addition, measures that incompletely describe the state of a project tend to warp development. People adapt to meet local deadlines and to optimize individual and group performance as defined by the measures. As a direct result, overall system integrity and performance suffer. For example, if a deadline is defined in terms of bugs removed or known bugs remaining, we may see that deadline met at the expense of run-time performance or hardware resources needed to run the system. Conversely, if only run-time performance is measured the error rate will surely rise when the developers struggle to optimize the system for benchmarks. The lack of good and comprehensive quality measures places great demands on the technical expertise of managers, but the alternative is a systematic tendency to reward random activity rather than progress. Don't forget that managers are also individuals. Managers need as least as much education on new techniques as do the people they manage.

As in other areas of software development, we must consider the longer term. It is essentially impossible to judge the performance of an individual on the basis of a single year's work. Most individuals do, however, have consistent long-term track records that can be reliable predictors of technical judgement and a useful help in evaluating immediate past performance. Disregard of such records – as is done when individuals are considered merely as interchangeable cogs in the wheels of an organization – leaves managers at the mercy of misleading quantity measurements.

One consequence of taking a long-term view and avoiding the ''interchangeable morons school of management'' is that individuals (both developers and managers) need longer to grow into the more demanding and interesting jobs. This discourages job hopping as well as job rotation for ''career development.'' A low turnover of both key technical people and key managers must be a goal. No manager can succeed without a rapport with key designers and programmers and some recent and relevant technical knowledge. Conversely, no group of designers and developers can succeed in the long run without support from competent managers and a minimum of understanding of the larger nontechnical context in which they work.

Where innovation is needed, senior technical people, analysts, designers, programmers, etc., have a critical and difficult role to play in the introduction of new techniques. These are the people who must learn new techniques and in many cases unlearn old habits. This is not easy. These individuals have typically made great personal investments in the old ways of doing things and rely on successes achieved using these ways of operating for their technical reputation. So do many technical managers.

Naturally, there is often a fear of change among such individuals. This can lead to an overestimation of the problems involved in a change and a reluctance to acknowledge problems with the old ways of doing things. Equally naturally, people arguing for change tend to overestimate the beneficial effects of new ways of doing things and to underestimate the problems involved in a change. These two groups of individuals *must* communicate, they *must* learn to talk the same language, they *must* help each other hammer out a model for transition. The alternative is organizational paralysis and the departure of the most capable individuals from both groups. Both groups

should remember that the most successful ''old timers'' are often the ''young turks'' of yesteryear. Given a chance to learn without humiliation, more experienced programmers and designers can become the most successful and insightful proponents of change. Their healthy skepticism, knowledge of users, and acquaintance with the organizational hurdles can be invaluable. Proponents of immediate and radical change must realize that a transition, often involving a gradual adoption of new techniques, is more often than not necessary. Conversely, individuals who have no desire to change should search out areas in which no change is needed rather than fight vicious rear-guard battles in areas in which new demands have already significantly altered the conditions for success.

23.5.4 Hybrid Design [design.hybrid]

Introducing new ways of doing things into an organization can be painful. The disruption to the organization and the individuals in the organization can be significant. In particular, an abrupt change that overnight turns productive and proficient members of ''the old school'' into ineffective novices in ''the new school'' is typically unacceptable. However, it is rare to achieve major gains without changes, and significant changes typically involve risks.

C++ was designed to minimize such risks by allowing a gradual adoption of techniques. Although it is clear that the largest benefits from using C++ are achieved through data abstraction, object-oriented programming, and object-oriented design, it is not clear that the fastest way to achieve these gains is a radical break with the past. Occasionally, such a clean break is feasible. More often, the desire for improvement is – or should be – tempered by concerns about how to manage the transition. Consider:

- Designers and programmers need time to acquire new skills.
- New code needs to cooperate with old code.
- Old code needs to be maintained (often indefinitely).
- Work on existing designs and programs needs to be completed (on time).
- Tools supporting the new techniques need to be introduced into the local environment.

These factors lead naturally to a hybrid style of design – even where that isn't the intention of some designers. It is easy to underestimate the first two points.

By supporting several programming paradigms, C++ supports the notion of a gradual introduction into an organization in several ways:

- Programmers can remain productive while learning C++.
- C++ can yield significant benefits in a tool-poor environment.
- C++ program fragments can cooperate well with code written in C and other traditional languages.
- C++ has a large C-compatible subset.

The idea is that programmers can make the move to C++ from a traditional language by first adopting C++ while retaining a traditional (procedural) style of programming. Then they use the data abstraction techniques. Finally – when the language and its associated tools have been mastered – they move on to object-oriented programming and generic programming. Note that a well-designed library is much easier to use than it was to design and implement, so a novice can benefit from the more advanced uses of abstraction even during the early stages of this progress.

The idea of learning object-oriented design, object-oriented programming, and C++ in stages is supported by facilities for mixing C++ code with code written in languages that do not support

C++'s notions of data abstraction and object-oriented programming (§24.2.1). Many interfaces can simply be left procedural because there will be no immediate benefits in doing anything more complicated. For many key libraries, this will already have been done by the library provider so that the C++ programmer can stay ignorant of the actual implementation language. Using libraries written in languages such as C is the first, and initially most important, form of reuse in C++.

The next stage – to be used only where a more elaborate technique is actually needed – is to present facilities written in languages such as C and Fortran as classes by encapsulating the data structures and functions in C++ interface classes. A simple example of lifting the semantics from the procedure plus data structure level to the data abstraction level is the string class from §11.12. There, encapsulation of the C character string representation and the standard C string functions is used to produce a string type that is much simpler to use.

A similar technique can be used to fit a built-in or stand-alone type into a class hierarchy (§23.5.1). This allows designs for C++ to evolve to use data abstraction and class hierarchies in the presence of code written in languages in which these concepts are missing and even under the constraint that the resulting code must be callable from procedural languages.

23.6 Annotated Bibliography [design.ref]

This chapter only scratches the surface of the issues of design and of the management of programming projects. For that reason, a short annotated bibliography is provided. An extensive annotated bibliography can be found in [Booch,1994].

[Anderson,1990] Bruce Anderson and Sanjiv Gossain: *An Iterative Design Model for Reusable Object-Oriented Software*. Proc. OOPSLA'90. Ottawa, Canada. A description of an iterative design and redesign model with a specific example and a discussion of experience.

[Booch,1994] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings. 1994. ISBN 0-8053-5340-2. Contains a detailed description of design, a specific design method with a graphical notation, and several large examples of designs expressed in C++. It is an excellent book to which this chapter owes much. It provides a more in-depth treatment of many of the issues in this chapter.

[Booch,1996] Grady Booch: *Object Solutions*. Benjamin/Cummings. 1996. ISBN 0-8053-0594-7. Describes the development of object-oriented systems from a management perspective. Contains extensive C++ code examples.

[Brooks,1982] Fred Brooks: *The Mythical Man Month*. Addison-Wesley. 1982. Everyone should read this book every couple of years. A warning against hubris. It is a bit dated on technical matters, but it is not at all dated in matters related to individuals, organizations, and scale. Republished with additions in 1997. ISBN 1-201-83595-9.

[Brooks,1987] Fred Brooks: *No Silver Bullet*. IEEE Computer, Vol. 20, No. 4. April 1987. A summary of approaches to large-scale software development, with a much-needed warning against belief in miracle cures (''silver bullets'').

[Coplien,1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. 1995. ISBN 1-201-60734-4.

[Gamma,1994] Eric Gamma, et. al.: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2. A practical catalog of techniques for creating flexible and reusable software, with a nontrivial, well-explained example. Contains extensive C++ code examples.

[DeMarco,1987] T. DeMarco and T. Lister: *Peopleware*. Dorset House Publishing Co. 1987. One of the few books that focusses on the role of people in the production of software. A must for every manager. Smooth enough for bedside reading. An antidote for much silliness.

[Jacobson,1992] Ivar Jacobson et. al.: *Object-Oriented Software Engineering*. Addison-Wesley. 1992. ISBN 0-201-54435-0. A thorough and practical description of software development in an industrial setting with an emphasis on use cases (§23.4.3.1). Miscasts C++ by describing it as it was ten years ago.

[Kerr,1987] Ron Kerr: *A Materialistic View of the Software ''Engineering'' Analogy*. In SIGPLAN Notices, March 1987. The use of analogy in this chapter and the next owes much to the observations in this paper and to the presentations by and discussions with Ron that preceded it.

[Liskov,1987] Barbara Liskov: *Data Abstraction and Hierarchy*. Proc. OOPSLA'87 (Addendum). Orlando, Florida. A discussion of how the use of inheritance can compromise data abstraction. Note, C++ has specific language support to help avoid most of the problems mentioned (§24.3.4).

[Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. 1995. ISBN 0-13-203837-4. Shows how to go from a problem to C++ code in a fairly systematic way. Presents alternative designs and principles for choosing between them. More practical and more concrete than most books on design. Contains extensive C++ code examples.

[Parkinson,1957] C. N. Parkinson: *Parkinson's Law and other Studies in Administration*. Houghton Mifflin. Boston. 1957. One of the funniest and most cutting descriptions of disasters caused by administrative processes.

[Meyer,1988] Bertrand Meyer: *Object Oriented Software Construction*. Prentice Hall. 1988. Pages 1-64 and 323-334 give a good introduction to one view of object-oriented programming and design with many sound pieces of practical advice. The rest of the book describes the Eiffel language. Tends to confuse Eiffel with universal principles.

[Shlaer,1988] S. Shlaer and S. J. Mellor: *Object-Oriented Systems Analysis* and *Object Lifecycles*. Yourdon Press. ISBN 0-13-629023-X and 0-13-629940-7. Presents a view of analysis, design, and programming that differs strongly from the one presented here and embodied in C++ and does so using a vocabulary that makes it sound rather similar.

[Snyder,1986] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Proc. OOPSLA'86. Portland, Oregon. Probably the

first good description of the interaction between encapsulation and inheritance. Also provides a nice discussion of some notions of multiple inheritance.

[Wirfs-Brock,1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener: *Designing Object-Oriented Software*. Prentice Hall. 1990. Describes an anthropomorphic design method based on role playing using CRC (Classes, Responsibilities, and Collaboration) cards. The text, if not the method itself, is biased toward Smalltalk.

23.7 Advice [design.advice]

[1] Know what you are trying to achieve; §23.3.
[2] Keep in mind that software development is a human activity; §23.2, §23.5.3.
[3] Proof by analogy is fraud; §23.2.
[4] Have specific and tangible aims; §23.4.
[5] Don't try technological fixes for sociological problems; §23.4.
[6] Consider the longer term in design and in the treatment of people; §23.4.1, §23.5.3.
[7] There is no lower limit to the size of programs for which it is sensible to design before starting to code; §23.2.
[8] Design processes to encourage feedback; §23.4.
[9] Don't confuse activity for progress; §23.3, §23.4.
[10] Don't generalize beyond what is needed, what you have direct experience with, and what can be tested; §23.4.1, §23.4.2.
[11] Represent concepts as classes; §23.4.2, §23.4.3.1.
[12] There are properties of a system that should not be represented as a class; §23.4.3.1.
[13] Represent hierarchical relationships between concepts as class hierarchies; §23.4.3.1.
[14] Actively search for commonality in the concepts of the application and implementation and represent the resulting more general concepts as base classes; §23.4.3.1, §23.4.3.5.
[15] Classifications in other domains are not necessarily useful classifications in an inheritance model for an application; §23.4.3.1.
[16] Design class hierarchies based on behavior and invariants; §23.4.3.1, §23.4.3.5, §24.3.7.1.
[17] Consider use cases; §23.4.3.1.
[18] Consider using CRC cards; §23.4.3.1.
[19] Use existing systems as models, as inspiration, and as starting points; §23.4.3.6.
[20] Beware of viewgraph engineering; §23.4.3.1.
[21] Throw a prototype away before it becomes a burden; §23.4.4
[22] Design for change, focusing on flexibility, extensibility, portability, and reuse; §23.4.2.
[23] Focus on component design; §23.4.3.
[24] Let each interface represent a concept at a single level of abstraction; §23.4.3.1.
[25] Design for stability in the face of change; §23.4.2.
[26] Make designs stable by making heavily-used interfaces minimal, general, and abstract; §23.4.3.2, §23.4.3.5.
[27] Keep it small. Don't add features ''just in case;'' §23.4.3.2.

[28] Always consider alternative representations for a class. If no alternative representation is plausible, the class is probably not representing a clean concept; §23.4.3.4.
[29] Repeatedly review and refine both the design and the implementation; §23.4, §23.4.3.
[30] Use the best tools available for testing and for analyzing the problem, the design, and the implementation; §23.3, §23.4.1, §23.4.4.
[31] Experiment, analyze, and test as early as possible and as often as possible; §23.4.4, §23.4.5.
[32] Don't forget about efficiency; §23.4.7.
[33] Keep the level of formality appropriate to the scale of the project; §23.5.2.
[34] Make sure that someone is in charge of the overall design; §23.5.2.
[35] Document, market, and support reusable components; §23.5.1.
[36] Document aims and principles as well as details; §23.4.6.
[37] Provide tutorials for new developers as part of the documentation; §23.4.6.
[38] Reward and encourage reuse of designs, libraries, and classes; §23.5.1.

24

Design and Programming

Keep it simple:
as simple as possible,
but no simpler.
– A. Einstein

Design and programming language — classes — inheritance — type checking — programming — what do classes represent? — class hierarchies — dependencies — containment — containment and inheritance — design tradeoffs — use relationships — programmed-in relationships — invariants — assertions — encapsulation — components — templates — interfaces and implementations — advice.

24.1 Overview [lang.overview]

This chapter considers the ways programming languages in general and C++ in particular can support design:

§24.2 The fundamental role of classes, class hierarchies, type checking, and programming itself

§24.3 Uses of classes and class hierarchies, focussing on dependencies between different parts of a program

§24.4 The notion of a *component*, which is the basic unit of design, and some practical observations about how to express interfaces

More general design issues are found in Chapter 23, and the various uses of classes are discussed in more detail in Chapter 25.

24.2 Design and Programming Language [lang.intro]

If I were to build a bridge, I would seriously consider what material to build it out of. Also, the design of the bridge would be heavily influenced by the choice of material and vice versa. Reasonable designs for stone bridges differ from reasonable designs for steel bridges, from reasonable designs for wooden bridges, etc. I would not expect to be able to select the proper material for a bridge without knowing a bit about the various materials and their uses. Naturally, you don't have to be an expert carpenter to design a wooden bridge, but you do have to know the fundamentals of wooden constructions to choose between wood and iron as the material for a bridge. Furthermore, even though you don't personally have to be an expert carpenter to design a wooden bridge, you do need quite a detailed knowledge of the properties of wood and the mores of carpenters.

The analogy is that to choose a language for some software, you need knowledge of several languages, and to design a piece of software successfully, you need a fairly detailed knowledge of the chosen implementation language – even if you never personally write a single line of that software. The good bridge designer respects the properties of materials and uses them to enhance the design. Similarly, the good software designer builds on the strengths of the implementation language and – as far as possible – avoids using it in ways that cause problems for implementers.

One might think that this sensitivity to language issues comes naturally when only a single designer/programmer is involved. However, even in such cases the programmer can be seduced into misusing the language due to inadequate experience or undue respect for styles of programming established for radically different languages. When the designer is different from the programmer – and especially if they do not share a common culture – the likelihood of introducing error, inelegance, and inefficiencies into the resulting system approaches certainty.

So what can a programming language do for a designer? It can provide features that allow the fundamental notions of the design to be represented directly in the programming language. This eases the implementation, makes it easier to maintain the correspondence between the design and the implementation, enables better communication between designers and implementers, and allows better tools to be built to support both designers and implementers.

For example, most design methods are concerned about dependencies between different parts of a program (usually to minimize them and to ensure that they are well defined and understood). A language that supports explicit interfaces between parts of a program can support such design notions. It can guarantee that only the expected dependencies actually exist. Because many dependencies are explicit in code written in such a language, tools that read a program to produce charts of dependencies can be provided. This eases the job of designers and others that need to understand the structure of a program. A programming language such as C++ can be used to decrease the gap between design and program and consequently reduce the scope for confusion and misunderstandings.

The key notion of C++ is that of a class. A C++ class is a type. Together with namespaces, classes are also a primary mechanism for information hiding. Programs can be specified in terms of user-defined types and hierarchies of such user-defined types. Both built-in and user-defined types obey statically checked type rules. Virtual functions provide a mechanism for run-time binding without breaking the static type rules. Templates support the design of parameterized types. Exceptions provide a way of making error handling more regular. These C++ features can be used without incurring overhead compared to C programs. These are the first-order properties of C++

that must be understood and considered by a designer. In addition, generally available major libraries – such as matrix libraries, database interfaces, graphical user interface libraries, and concurrency support libraries – can strongly affect design choices.

Fear of novelty sometimes leads to sub-optimal use of C++. So does misapplication of lessons from other languages, systems, and application areas. Poor design tools can also warp designs. Five ways designers fail to take advantage of language features and fail to respect limitations are worth mentioning:

[1] Ignore classes and express the design in a way that constrains implementers to use the C subset only.
[2] Ignore derived classes and virtual functions and use only the data abstraction subset.
[3] Ignore the static type checking and express the design in such a way that implementers are constrained to simulate dynamic type checking.
[4] Ignore programming and express systems in a way that aims to eliminate programmers.
[5] Ignore everything except class hierarchies.

These variants are typical for designers with

[1] a C, traditional CASE, or structured design background,
[2] an Ada83, Visual Basic, or data abstraction background,
[3] a Smalltalk or Lisp background,
[4] a nontechnical or very specialized background,
[5] a background with heavy emphasis on ‘‘pure’’ object-oriented programming,

respectively. In each case, one must wonder if the implementation language was well chosen, if the design method was well chosen, or if the designer had failed to adapt to the tool in hand.

There is nothing unusual or shameful in such a mismatch. It is simply a mismatch that delivers sub-optimal designs and imposes unnecessary burdens on programmers. It does the same to designers when the conceptual framework of the design method is noticeably poorer than C++’s conceptual framework. Therefore, we avoid such mismatches wherever possible.

The following discussion is phrased as answers to objections because that is the way it often occurs in real life.

24.2.1 Ignoring Classes [lang.ignore.class]

Consider design that ignores classes. The resulting C++ program will be roughly equivalent to the C program that would have resulted from the same design process – and this program would again be roughly equivalent to the COBOL program that would have resulted from the same design process. In essence, the design has been made ‘‘programming language independent’’ at the cost of forcing the programmer to code in the common subset of C and COBOL. This approach does have advantages. For example, the strict separation of data and code that results makes it easy to use traditional databases that are designed for such programs. Because a minimal programming language is used, it would appear that less skill – or at least different skills – would be required from programmers. For many applications – say, a traditional sequential database update program – this way of thinking is quite reasonable, and the traditional techniques developed over decades are adequate for the job.

However, suppose the application differs sufficiently from traditional sequential processing of records (or characters) or the complexity involved is higher – say, in an interactive CASE system.

The lack of language support for data abstraction implied by the decision to ignore classes will hurt. The inherent complexity will show up in the application somewhere, and if the system is implemented in an impoverished language, the code will not reflect the design directly. The program will have too many lines of source code, lack type checking, and will in general not be amenable to tools. This is the prescription for a maintenance nightmare.

A common band-aid for this problem is to build specific tools to support the notions of the design method. These tools then provide higher-level constructs and checking to compensate for deficiencies of the (deliberately impoverished) implementation language. Thus, the design method becomes a special-purpose and typically corporate-owned programming language. Such programming languages are in most contexts poor substitutes for a widely available, general-purpose programming language supported by suitable design tools.

The most common reason for ignoring classes in design is simple inertia. Traditional programming languages don't support the notion of a class, and traditional design techniques reflect this deficiency. The most common focus of design has been the decomposition of the problems into a set of procedures performing required actions. This notion, called procedural programming in Chapter 2, is in the context of design often called *functional decomposition*. A common question is, ''Can we use C++ together with a design method based on functional decomposition?'' You can, but you will most likely end up using C++ as simply a better C and will suffer the problems mentioned previously. This may be acceptable in a transition period, for already completed designs, and for subsystems in which classes do not appear to offer significant benefits (given the experience of the individuals involved at this time). For the longer term and in general, however, the policy against large-scale use of classes implied by functional decomposition is not compatible with effective use of C++ or any other language that has support for abstraction.

The procedure-oriented and object-oriented views of programming are fundamentally different and typically lead to radically different solutions to the same problem. This observation is as true for the design phase as it is for the implementation phase: you can focus the design on the actions taken or on the entities represented, but not simultaneously on both.

So why prefer ''object-oriented design'' over the traditional design methods based on functional decomposition? A first-order answer is that functional decomposition leads to insufficient data abstraction. From this, it follows that the resulting design is

- less resilient to change,
- less amenable to tools,
- less suited for parallel development, and
- less suited for concurrent execution.

The problem is that functional decomposition causes interesting data to become global because when a system is structured as a tree of functions, any data accessed by two functions must be global to both. This ensures that ''interesting'' data bubbles up toward the root of the tree as more and more functions require access to it (as ever in computing, trees grow from the root down). Exactly the same process can be seen in single-rooted class hierarchies, in which ''interesting'' data and functions tend to bubble up toward a root class (§24.4). Focussing on the specification of classes and the encapsulation of data addresses this problem by making the dependencies between different parts of a program explicit and tractable. More important, though, it reduces the number of dependencies in a system by improving locality of reference to data.

However, some problems are best solved by writing a set of procedures. The point of an ''object-oriented'' approach to design is not that there should never be any nonmember functions in a program or that no part of a system may be procedure-oriented. Rather, the key point is to decouple different parts of a program to better reflect the concepts of the application. Typically, that is best done when classes, not functions, are the primary focus on the design effort. The use of a procedural style should be a conscious decision and not simply a default. Both classes and procedures should be used appropriately relative to the application and not just as artifacts of an inflexible design method.

24.2.2 Avoiding Inheritance [lang.avoid.hier]

Consider design that avoids inheritance. The resulting programs simply fail to take advantage of a key C++ feature, while still reaping many benefits of C++ compared to C, Pascal, Fortran, COBOL, etc. Common reasons for doing this – apart from inertia – are claims that ''inheritance is an implementation detail,'' ''inheritance violates information hiding,'' and ''inheritance makes cooperation with other software harder.''

Considering inheritance merely an implementation detail ignores the way that class hierarchies can directly model key relationships between concepts in the application domain. Such relationships should be explicit in the design to allow designers to reason about them.

A strong case can be made for excluding inheritance from the parts of a C++ program that must interface directly with code written in other languages. This is, however, *not* a sufficient reason for avoiding the use of inheritance throughout a system; it is simply a reason for carefully specifying and encapsulating a program's interface to ''the outer world.'' Similarly, worries about compromising information hiding through the use of inheritance (§24.3.2.1) are a reason to be careful with the use of virtual functions and protected members (§15.3). They are not a reason for general avoidance.

In many cases, there is no real advantage to be gained from inheritance. However, in a large project a policy of ''no inheritance'' will result in a less comprehensible and less flexible system in which inheritance is ''faked'' using more traditional language and design constructs. Further, I suspect that despite such a policy, inheritance will eventually be used anyway because C++ programmers will find convincing arguments for inheritance-based designs in various parts of the system. Therefore, a ''no inheritance'' policy will ensure only that a coherent overall architecture will be missing and will restrict the use of class hierarchies to specific subsystems.

In other words, keep an open mind. Class hierarchies are not an essential part of every good program, but in many cases they can help in both the understanding of the application and the expression of a solution. The fact that inheritance can be misused and overused is a reason for caution; it is a not reason for prohibition.

24.2.3 Ignoring Static Type Checking [lang.type]

Consider design that ignores static type checking. Commonly stated reasons to ignore static type checking in the design phase are that ''types are an artifact of the programming language,'' that ''it is more natural to think about objects without bothering about types,'' and that ''static type checking forces us to think about implementation issues too early.'' This attitude is fine as far as it goes and harmless up to a point. It is reasonable to ignore details of type checking in the design stage,

and it is often safe to ignore type issues almost completely in the analysis stage and early design stages. However, classes and class hierarchies are very useful in the design. In particular, they allow us to be specific about concepts, allow us to be precise about their relationships, and help us reason about the concepts. As the design progresses, this precision takes the form of increasingly precise statements about classes and their interfaces.

It is important to realize that precisely-specified and strongly-typed interfaces are a fundamental design tool. C++ was designed with this in mind. A strongly-typed interface ensures (up to a point) that only compatible pieces of software can be compiled and linked together and thus allows these pieces of software to make relatively strong assumptions about each other. These assumptions are guaranteed by the type system. The effect of this is to minimize the use of run-time tests, thus promoting efficiency and causing significant reductions in the integration phase of multiperson projects. In fact, strong positive experience with integrating systems that provide strongly-typed interfaces is the reason integration isn't a major topic of this chapter.

Consider an analogy. In the physical world, we plug gadgets together all the time, and a seemingly infinite number of standards for plugs exists. The most obvious thing about these plugs is that they are specifically designed to make it impossible to plug two gadgets together unless the gadgets were designed to be plugged together, and then they can be connected only in the right way. You cannot plug an electric shaver into a high-power socket. Had you been able to, you would have ended up with a fried shaver or a fried shavee. Much ingenuity is expended on ensuring that incompatible pieces of hardware cannot be plugged together. The alternative to using many incompatible plugs is gadgets that protect themselves against undesirable behavior from gadgets plugged into their sockets. A surge protector is a good example of this. Because perfect compatibility cannot be guaranteed at the ''plug compatibility level,'' we occasionally need the more expensive protection of circuitry that dynamically adapts to and/or protects from a range of inputs.

The analogy is almost exact. Static type checking is equivalent to plug compatibility, and dynamic checking corresponds to protection/adaptation circuitry. If both checks fail – in either the physical world or the software world – serious damage can result. In large systems, both forms of checking are used. In the early stages of a design, it may be reasonable simply to say, ''These two gadgets should be plugged together.'' However, it soon becomes relevant exactly how they should be plugged together. What guarantees does the plug provide about behavior? What error conditions are possible? What are the first-order cost estimates?

The use of ''static typing'' is not limited to the physical world. The use of units (for example, meters, kilograms, and seconds) to prevent the mixing of incompatible entities is pervasive in physics and engineering.

In the description of the design steps in §23.4.3, type information enters the picture in Step 2 (presumably after being superficially considered in Step 1) and becomes a major issue in Step 4.

Statically-checked interfaces are the prime vehicle for ensuring cooperation between C++ software developed by different groups. The documentation of these interfaces (including the exact types involved) is the primary means of communication between separate groups of programmers. These interfaces are one of the most important outputs of the design process and a focus of communication between designers and programmers.

Ignoring type issues when considering interfaces leads to designs that obscure the structure of the program and postpone error detection until run time. For example, an interface can be specified in terms of self-identifying objects:

```
// Example assuming dynamic type checking instead of static checking:

Stack s; // Stack can hold pointers to objects of any type

void f()
{
    s.push(new Saab900);
    s.push(new Saab37B);

    s.pop()->takeoff();        // fine: a Saab 37B is a plane
    s.pop()->takeoff();        // run-time error: car cannot take off
}
```

This is a severe underspecification of the interface (of *Stack*::*push*( )) that forces dynamic checking rather than static checking. The stack *s* is meant to hold *Plane*s, but that was left implicit in the code, so it becomes the user's obligation to make sure the requirement is upheld.

A more precise specification – a template plus virtual functions rather than unconstrained dynamic type checking – moves error detection from run time to compile time:

```
Stack<Plane*> s; // Stack can hold pointers to Planes

void f()
{
    s.push(new Saab900);    // error: a Saab900 is not a Plane
    s.push(new Saab37B);

    s.pop()->takeoff();        // fine: a Saab 37B is a plane
    s.pop()->takeoff();
}
```

A similar point is made in §16.2.2. The difference in run time between dynamic checking and static checking can be significant. The overhead of dynamic checking is usually a factor in the range of 3 to 10.

One should not go to the other extreme, though. It is not possible to catch all errors by static checking. For example, even the most thoroughly statically checked program is vulnerable to hardware failures. See also §25.4.1 for an example where complete static checking would be infeasible. However, the ideal is to have the vast majority of interfaces be statically typed with application-level types; see §24.4.2.

Another problem is that a design can be perfectly reasonable in the abstract but can cause serious trouble because it fails to take into account limitations of a basic tool, in this case C++. For example, a function *f*( ) that needs to perform an operation *turn_right*( ) on an argument can do so only provided all of its arguments are of a common type:

```
class Plane {
    // ...
    void turn_right();
};
```

```
class Car {
        // ...
        void turn_right();
};

void f(X* p)    // what type should X be?
{
        p->turn_right();
        // ...
}
```

Some languages (such as Smalltalk and CLOS) allow two types to be used interchangeably if they have the same operations by relating every type through a common base and postponing name resolution until run time. However, C++ (intentionally) supports this notion through templates and compile-time resolution only. A non-template function can accept arguments of two types only if the two types can be implicitly converted to a common type. Thus, in the previous example ***X*** must be a common base of ***Plane*** and ***Car*** (e.g., a ***Vehicle*** class).

Typically, examples inspired by notions alien to C++ *can* be mapped into C++ by expressing the assumptions explicitly. For example, given ***Plane*** and ***Car*** (without a common base), we can still create a class hierarchy that allows us to pass an object containing a ***Car*** or a ***Plane*** to ***f(X*)*** (§25.4.1). However, doing this often requires an undesirable amount of mechanism and cleverness. Templates are often a useful tool for such concept mappings. A mismatch between design notions and C++ typically leads to ''unnatural-looking'' and inefficient code. Maintenance programmers tend to dislike the non-idiomatic code that arises from such mismatches.

A mismatch between the design technique and the implementation language can be compared to word-for-word translation between natural languages. For example, English with German grammar is as awkward as German with English grammar, and both can be close to incomprehensible to someone fluent in only one of those languages.

Classes in a program are the concrete representation of the concepts of the design. Consequently, obscuring the relationships between the classes obscures the fundamental concepts of the design.

24.2.4 Avoiding Programming [lang.prog]

Programming is costly and unpredictable compared to many other activities, and the resulting code is often less than 100% reliable. Programming is labor-intensive and – for a variety of reasons – most serious project delays manifest themselves by code not being ready to ship. So, why not eliminate programming as an activity altogether?

To many managers, getting rid of the arrogant, undisciplined, over-paid, technology-obsessed, improperly-dressed, etc. programmers† would appear to be a significant added benefit. To a programmer, this suggestion may sound absurd. However, important problem areas with realistic alternatives to traditional programming do exist. For specific areas, it is possible to generate code directly from a high-level specification. In other areas, code can be generated by manipulating shapes on a screen. For example, useful user interfaces can be constructed by direct manipulation

† Yes, I'm a programmer.

in a tiny fraction of the time it would take to construct the same interface by writing traditional code. Similarly, database layouts and the code for accessing data according to such layouts can be generated from specifications that are far simpler than the code needed to express those operations directly in C++ or in any other general-purpose programming language. State machines that are smaller, faster, and more correct than most programmers could produce can be generated from specifications or by a direct manipulation interface.

These techniques work well in specific areas where there is either a sound theoretical foundation (e.g., math, state machines, and relational databases) or where a general framework exists into which small application fragments can be embedded (e.g., graphical user interfaces, network simulations, and database schema). The obvious usefulness of these techniques in limited – and typically crucial – areas can tempt people to think that the elimination of traditional programming by these techniques is ''just around the corner.'' It is not. The reason is that expanding specification techniques outside areas with sound theoretical frameworks implies that the complexity of a general-purpose programming language would be needed in the specification language. This defeats the purpose of a clean and well-founded specification language.

It is sometimes forgotten that the framework that allows elimination of traditional programming in an area is a system or library that has been designed, programmed, and tested in the traditional way. In fact, one popular use of C++ and the techniques described in this book is to design and build such systems.

A compromise that provides a small fraction of the expressiveness of a general-purpose language is the worst of both worlds when applied outside a restricted application domain. Designers who stick to a high-level modeling point of view are annoyed by the added complexity and produce specifications from which horrendous code is produced. Programmers who apply ordinary programming techniques are frustrated by the lack of language support and generate better code only by excessive effort and by abandoning high-level models.

I see no signs that programming as an activity can be successfully eliminated outside areas that either have well-founded theoretical bases or in which the basic programming is provided by a framework. In either case, there is a dramatic drop in the effectiveness of the techniques as one leaves the original framework and attempts more general-purpose work. Pretending otherwise is tempting and dangerous. Conversely, ignoring the high-level specification techniques and the direct-manipulation techniques in domains in which they are well-founded and reasonably mature would be a folly.

Designing tools, libraries, and frameworks is one of the highest forms of design and programming. Constructing a useful mathematically-based model of an application area is one of the highest forms of analysis. Thus, providing a tool, language, framework, etc., that makes the result of such work available to thousands is a way for programmers and designers to escape the trap of becoming craftsmen of one-of-a-kind artifacts.

It is most important that a specification system or a foundation library be able to interface effectively with a general-purpose programming language. Otherwise, the framework provided is inherently limiting. This implies that specification systems and direct-manipulation systems that generate code at a suitable high level into an accepted general-purpose programming language have a great advantage. A proprietary language is a long-term advantage to its provider only. If the code generated is so low-level that general code added must be written without the benefits of abstraction, then reliability, maintainability, and economy are lost. In essence, a generation system should

be designed to combine the strengths of higher-level specifications and higher-level programming languages. To exclude one or the other is to sacrifice the interests of system builders to the interests of tool providers. Successful large systems are multilevel and modular and evolve over time. Consequently, successful efforts to produce such systems involve a variety of languages, libraries, tools, and techniques.

24.2.5 Using Class Hierarchies Exclusively [lang.pure]

When we find that something new actually works, we often go a bit overboard and apply it indiscriminately. In other words, a great solution to some problems often appears to be *the* solution to almost all problems. Class hierarchies and operations that are polymorphic on their (one) object provide a great solution to many problems. However, not every concept is best represented as a part of a hierarchy and not every software component is best represented as a class hierarchy.

Why not? A class hierarchy expresses relationships between its classes and a class represents a concept. Now what is the common relationship between a smile, the driver for my CD-ROM reader, a recording of Richard Strauss' Don Juan, a line of text, a satellite, my medical records, and a real-time clock? Placing them all in a single hierarchy when their only shared property is that they are programming artifacts (they are all ''objects'') is of little fundamental value and can cause confusion (§15.4.5). Forcing everything into a single hierarchy can introduce artificial similarities and obscure real ones. A hierarchy should be used only if analysis reveals conceptual commonality or if design and programming discover useful commonality in the structures used to implement the concepts. In the latter case, we have to be very careful to distinguish genuine commonality (to be reflected as subtyping by public inheritance) and useful implementation simplifications (to be reflected as private inheritance; §24.3.2.1).

This line of thinking leads to a program that has several unrelated or weakly-related class hierarchies, each representing a set of closely related concepts. It also leads to the notion of a concrete class (§25.2) that is not part of a hierarchy because placing such a class in a hierarchy would compromise its performance and its independence of the rest of the system.

To be effective, most critical operations on a class that is part of a class hierarchy must be virtual functions. Furthermore, much of that class' data must be protected rather than private. This makes it vulnerable to modification from further derived classes and can seriously complicate testing. Where stricter encapsulation makes sense from a design point of view, non-virtual functions and private data should be used (§24.3.2.1).

Having one argument of an operation (the one designating ''the object'') special can lead to contorted designs. When several arguments are best treated equally, an operation is best represented as a nonmember function. This does not imply that such functions should be global. In fact, almost all such free-standing functions should be members of a namespace (§24.4).

24.3 Classes [lang.class]

The most fundamental notion of object-oriented design and programming is that the program is a model of some aspects of reality. The classes in the program represent the fundamental concepts of the application and, in particular, the fundamental concepts of the ''reality'' being modeled. Real-world objects and artifacts of the implementation are represented by objects of these classes.

The analysis of relationships between classes and within parts of a class is central to the design of a system:

§24.3.2 Inheritance relationships
§24.3.3 Containment relationships
§24.3.5 Use relationships
§24.2.4 Programmed-in relationships
§24.3.7 Relationships within a class

Because a C++ class is a type, classes and the relationships between classes receive significant support from compilers and are generally amenable to static analysis.

To be relevant in a design, a class doesn't just have to represent a useful concept; it must also provide a suitable interface. Basically, the ideal class has a minimal and well-defined dependence on the rest of the world and presents an interface that exposes the minimal amount of information necessary to the rest of the world (§24.4.2).

24.3.1 What Do Classes Represent? [lang.what]

There are essentially two kinds of classes in a system:

[1] Classes that directly reflect the concepts in the application domain; that is, concepts that are used by end-users to describe their problems and solutions
[2] Classes that are artifacts of the implementation; that is, concepts that are used by the designers and programmers to describe their implementation techniques.

Some of the classes that are artifacts of the implementation may also represent real-world entities. For example, the hardware and software resources of a system provide good candidates for classes in an application. This reflects the fact that a system can be viewed from several viewpoints. This implies that one person's implementation detail is another person's application. A well-designed system will contain classes supporting logically separate views of the system. For example:

[1] Classes representing user-level concepts (e.g., cars and trucks)
[2] Classes representing generalizations of the user-level concepts (e.g. vehicles)
[3] Classes representing hardware resources (e.g., a memory management class)
[4] Classes representing system resources (e.g., output streams)
[5] Classes used to implement other classes (e.g., lists, queues, locks)
[6] Built-in data types and control structures.

In larger systems, keeping logically separate types of classes separate and maintaining separation between several levels of abstraction becomes a challenge. A simple example can be considered to have three levels of abstraction:

[1+2] Provide an application level view of the system
[3+4] Represent the machine on which the model runs
[5+6] Represent a low-level (programming language) view of the implementation.

The larger the system, the more levels of abstraction are typically needed for the description of the system and the more difficult it becomes to define and maintain the levels. Note that such levels of abstraction have direct counterparts in nature and in other types of human constructions. For example, a house can be considered as consisting of

[1] atoms;
[2] molecules;

[3] lumber and bricks;
[4] floors, walls, and ceilings; and
[5] rooms.

As long as these levels of abstraction are kept separate, you can maintain a coherent view of the house. However, if you mix them, absurdities arise. For example, the statement, ''My house consists of several thousand pounds of carbon, some complex polymers, about 5,000 bricks, two bathrooms, and 13 ceilings,'' is silly. Given the abstract nature of software, the equivalent statement about a complex system is not always recognized for what it is.

The translation of a concept in the application area into a class in a design is not a simple mechanical operation. It often requires significant insights. Note that the concepts in an application area are themselves abstractions. For example, ''taxpayers,'' ''monks,'' and ''employees'' don't really exist in nature; such concepts are themselves labels put on individuals to classify them relative to some system. The real or even the imagined world (literature, especially science fiction) is sometimes simply a source of ideas for concepts that mutate radically in the transition into classes. For example, the screen of my PC doesn't really resemble my desktop despite its being designed to support the desktop metaphor†, and the windows on my screen bear only the slightest relation to the contraptions that let drafts into my office. The point about modeling reality is not to slavishly follow what we see but rather to use it as a starting point for design, a source of inspiration, and an anchor to hold on to when the intangible nature of software threatens to overcome our ability to understand our programs.

A word of caution: beginners often find it hard to ''find the classes,'' but that problem is usually soon overcome without long-term ill effects. Next, however, often follows a phase in which classes – and their inheritance relationships – seem to multiply uncontrollably. This can cause long-term problems with the complexity, comprehensibility, and efficiency of the resulting program. Not every minute detail needs to be represented by a distinct class, and not every relationship between classes needs to be represented as an inheritance relationship. Try to remember that the aim of a design is to model a system at an *appropriate* level of detail and at *appropriate* levels of abstraction. Finding a balance between simplicity and generality is not easy.

24.3.2 Class Hierarchies [lang.hier]

Consider simulating the traffic flow of a city to determine the likely times needed for emergency vehicles to reach their destinations. Clearly, we need to represent cars, trucks, ambulances, fire engines of various sorts, police cars, busses, etc. Inheritance comes into play because a real-world concept does not exist in isolation; it exists with numerous relationships to other concepts. Without understanding these relationships, we cannot understand the concepts. Consequently, a model that does not represent such relationships does not adequately represent our concepts. That is, in our programs we need classes to represent concepts, but that is not enough. We also need ways of representing relationships between classes. Inheritance is one powerful way of representing hierarchical relationships directly. In our example, we would probably consider emergency vehicles special and want also to distinguish between car-like and truck-like vehicles. This would yield a class hierarchy along these lines:

† I wouldn't be able to tolerate such a mess on my screen, anyway.

Here, ***Emergency*** represents the aspects of an emergency vehicle that are relevant to the simulation: it can violate some traffic rules, has priority in intersections when on an emergency call, it is under control of a dispatcher, etc.

Here is the C++ version:

```
class Vehicle { /* ... */ };
class Emergency { /* ... */ };
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car , protected Emergency { /* ... */ };
class Ambulance : public Car , protected Emergency { /* ... */ };
class Fire_engine : public Truck , protected Emergency { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
```

Inheritance is the highest level relationship that can be represented directly in C++ and the one that figures largest in the early stages of a design. Often there is a choice between using inheritance to represent a relationship and using membership. Consider an alternative notion of what it means to be an emergency vehicle: a vehicle is an emergency vehicle if it displays a flashing light. This would allow a simplification of the class hierarchy by replacing the ***Emergency*** class by a member in class ***Vehicle***:

Class ***Emergency*** is now simply used as a member in classes that might need to act as emergency vehicles:

```
class Emergency { /* ... */ };
class Vehicle { protected: Emergency* eptr; /* ... */ }; // better: provide proper interface to eptr
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car { /* ... */ };
class Ambulance : public Car { /* ... */ };
class Fire_engine : public Truck { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
```

Here, a vehicle is an emergency vehicle if ***Vehicle::eptr*** is nonzero. The ''plain'' cars and trucks are initialized with ***Vehicle::eptr*** zero; the others are initialized with ***Vehicle::eptr*** nonzero. For example:

```
Car::Car()      // Car constructor
{
    eptr = 0;
}

Police_car::Police_car()  // Police_car constructor
{
    eptr = new Emergency;
}
```

Defining things this way enables a simple conversion of an emergency vehicle to an ordinary vehicle and vice versa:

```
void f(Vehicle* p)
{
    delete p->eptr;
    p->eptr = 0;                    // no longer an emergency vehicle

    // ...

    p->eptr = new Emergency;        // an emergency vehicle again
}
```

So, which variant of the class hierarchy is best? The general answer is, ''The program that most directly models the aspects of the real world that we are interested in is the best.'' That is, in choosing between models we should aim for greater realism under the inevitable constraints of efficiency and simplicity. In this case, the easy conversion between ordinary vehicles and emergency vehicles seems unrealistic to me. Fire engines and ambulances are purpose-built vehicles manned by trained personnel and operated using dispatch procedures requiring specialized communication equipment. This view indicates that being an emergency vehicle should be a fundamental concept and represented directly in the program to improve type checking and other uses of tools. Had we been modeling a place where the roles of vehicles were less firmly defined – say, an area where private vehicles were routinely used to carry emergency personnel to accident sites and where communication was primarily based on portable radios – the other way of modeling the system might have been more appropriate.

For people who consider traffic simulations esoteric, it might be worth pointing out that such tradeoffs between inheritance and membership almost invariably occur in a design. The scrollbar example in §24.3.3 is an equivalent example.

24.3.2.1 Dependencies within a Class Hierarchy [lang.internal]

Naturally, a derived class depends on its base classes. It is less often appreciated that the opposite can also be true†. If a class has a virtual function, the class depends on derived classes to implement part of its functionality whenever a derived class overrides that function. If a member of a base class itself calls one of the class' virtual functions, then the base class depends on its derived classes for its own implementation. Similarly, if a class uses a protected member, then it is again dependent on its derived classes for its own implementation. Consider:

```
class B {
    // ...
protected:
    int a;
public:
    virtual int f();
    int g() { int x = f(); return x-a; }
};
```

What does *g*( ) do? The answer critically depends on the definition of *f*( ) in some derived class. Here is a version that will ensure that *g*( ) returns *1*:

```
class D1 : public B {
    int f() { return a+1; }
};
```

and a version that makes *g*( ) write ''*Hello, world*!'' and return *0*:

```
class D2 : public B {
    int f() { cout<<"Hello, world!\n"; return a; }
};
```

This example illustrates one of the most important points about virtual functions. Why is it silly? Why wouldn't a programmer ever write something like that? The answer is that a virtual function is part of an interface to a base class, and that class can supposedly be used without knowledge of the classes derived from it. Consequently, it must be possible to describe the expected behavior of an object of the base class in such a way that programs can be written without knowledge of the derived classes. Every class that overrides the virtual function must implement a variant of that behavior. For example, the virtual function *rotate*( ) of a *Shape* class rotates a shape. The *rotate*( ) functions for derived classes such as *Circle* and *Triangle* must rotate objects of their respective type; otherwise, a fundamental assumption about class *Shape* is violated. No such assumption about behavior is made for class *B* or its derived classes *D1* and *D2*; thus, the example is nonsensical. Even the names *B*, *D1*, *D2*, *f*, and *g* were chosen to obscure any possible meanings.

† This observation has been summarized as: ''Insanity is hereditary. You get it from your children.''

The specification of the expected behavior of virtual functions is a *major* focus of class design. Choosing good names for classes and functions is important – and not always easy.

Is a dependency on unknown (possibly yet unwritten) derived classes good or bad? Naturally, that depends on the intent of the programmer. If the intent is to isolate a class from all external influences so that it can be proven to behave in a specific way, then protected members and virtual functions are best avoided. If, however, the intent is to provide a framework into which a later programmer (such as the same programmer a few weeks later) can add code, then virtual functions are often an elegant mechanism for achieving this; and protected member functions have proven convenient for supporting such use. This technique is used in the stream I/O library (§21.6) and was illustrated by the final version of the ***Ival_box*** hierarchy (§12.4.2).

If a ***virtual*** function is meant to be used only indirectly by a derived class, it can be left ***private***. For example, consider a simple buffer template:

```
template<class T> class Buffer {
public:
        void put(T);      // call overflow(T) if buffer is full
        T get();          // call underflow() if buffer is empty
        // ...
private:
        virtual int overflow(T);
        virtual int underflow();
        // ...
};
```

The ***put***( ) and ***get***( ) functions call ***virtual*** functions ***overflow***( ) and ***underflow***( ), respectively. A user can now implement a variety of buffer types to suit a variety of needs by overriding ***overflow***( ) and ***underflow***( ):

```
template<class T> class Circular_buffer : public Buffer<T> {
        int overflow(T);        // wrap around if full
        int underflow();
        // ...
};

template<class T> class Expanding_buffer : public Buffer<T> {
        int overflow(T);        // increase buffer size if full
        int underflow();
        // ...
};
```

Only if a derived class needed to call ***overflow***( ) and ***underflow***( ) directly would these functions need to be ***protected*** rather than ***private***.

24.3.3 Containment Relationships [lang.contain]

Where containment is used, there are two major alternatives for representing an object of a class ***X***:

[1] Declare a member of type ***X***.

[2] Declare a member of type ***X***** or type ***X***&.

If the value of the pointer is never changed, these alternatives are equivalent, except for efficiency issues and the way you write constructors and destructors:

```
class X {
public:
    X(int);
    // ...
};

class C {
    X a;
    X* p;
    X& r;
public:
    C(int i, int j, int k) : a(i), p(new X(j)), r(*new X(k)) { }
    ~C() { delete p; delete &r; }
};
```

In such cases, membership of the object itself, as in the case of `C::a`, is usually preferable because it is the most efficient in time, space, and keystrokes. It is also less error-prone because the connection between the contained object and the containing object is covered by the rules of construction and destruction (§10.4.1, §12.2.2, §14.4.1). However, see also §24.4.2 and §25.7.

The pointer solution should be used when there is a need to change the pointer to the ''contained'' object during the life of the ''containing'' object. For example:

```
class C2 {
    X* p;
public:
    C2(int i) : p(new X(i)) { }
    ~C2() { delete p; }

    X* change(X* q)
    {
        X* t = p;
        p = q;
        return t;
    }
};
```

Another reason for using a pointer member is to allow the ''contained'' member to be supplied as an argument:

```
class C3 {
    X* p;
public:
    C3(X* q) : p(q) { }
    // ...
};
```

By having objects contain pointers to other objects, we create what are often called *object hierarchies*. This is an alternative and complementary technique to using class hierarchies. As shown in the emergency vehicle example in §24.3.2, it is often a tricky design issue to choose

between representing a property of a class as a base class or representing it as a member. A need to override is an indication that the former is the better choice. Conversely, a need to be able to allow the property to be represented by a variety of types is an indication that the latter is the better choice. For example:

```
class XX : public X { /* ... */ };

class XXX : public X { /* ... */ };

void f()
{
    C3* p1 = new C3(new X);      // C3 ''contains'' an X
    C3* p2 = new C3(new XX);     // C3 ''contains'' an XX
    C3* p3 = new C3(new XXX);    // C3 ''contains'' an XXX
    // ...
}
```

This could not be modeled by a derivation of ***C3*** from ***X*** or by ***C3*** having a member of type ***X***, because the exact type of a member needs to be used. This is important for classes with virtual functions, such as a shape class (§2.6.2) or an abstract set class (§25.3).

References can be used to simplify classes based on pointer membership when only one object is referred to during the life of the containing object. For example:

```
class C4 {
    X& r;
public:
    C4(X& q) : r(q) { }
    // ...
};
```

Pointer and reference members are also needed when an object needs to be shared:

```
X* p = new XX;
C4 obj1(*p);
C4 obj2(*p);   // obj1 and obj2 now share the new XX
```

Naturally, management of shared objects requires extra care – especially in concurrent systems.

24.3.4 Containment and Inheritance [lang.cont.hier]

Given the importance of inheritance relationships, it is not surprising that they are frequently overused and misunderstood. When a class ***D*** is publicly derived from another class ***B***, it is often said that a ***D*** *is* a ***B***:

```
class B { /* ... */ };
class D : public B { /* ... */ };      // D is a kind of B
```

Alternatively, this is expressed by saying that inheritance is an *is-a* relationship or – somewhat more precisely – that a ***D*** *is a kind of* ***B***. In contrast, a class ***D*** that has a member of another class ***B*** is often said to *have* a ***B*** or *contain* a ***B***. For example:

```
class D { // a D contains a B
public:
    B b;
    // ...
};
```

Alternatively, this is expressed by saying that membership is a *has-a* relationship.

For given classes ***B*** and ***D***, how do we choose between inheritance and membership? Consider an ***Airplane*** and an ***Engine***. Novices often wonder if it might be a good idea to derive class ***Airplane*** from ***Engine***. This is a bad idea, though, because an ***Airplane*** *is* not an ***Engine***; it *has* an ***Engine***. One way of seeing this is to consider if an ***Airplane*** might have two or more engines. Because that seems feasible (even if we are considering a program in which all of our ***Airplane***s will be single-engine ones), we should use membership rather than inheritance. The question ''can it have two?'' is useful in many cases when there is doubt. As usual, it is the intangible nature of software that makes this discussion relevant. Had all classes been as easy to visualize as ***Airplane*** and ***Engine***, trivial mistakes like deriving an ***Airplane*** from an ***Engine*** would be easily avoided. Such mistakes are, however, quite frequent – particularly among people who consider derivation as simply another mechanism for combining programming-language-level constructs. Despite the conveniences and shorthand notation that derivation provides, it should be used almost exclusively to express relationships that are well defined in a design. Consider:

```
class B {
public:
    virtual void f();
    void g();
};

class D1 {              // a D1 contains a B
public:
    B b;
    void f();           // does not override b.f()
};

void h1(D1* pd)
{
    B* pb = pd;         // error: no D1* to B* conversion
    pb = &pd->b;
    pb->g();            // calls B::g()
    pd->g();            // error: D1 doesn't have a member g()
    pd->b.g();
    pb->f();            // calls B::f (not overridden by D1::f())
    pd->f();            // calls D1::f()
}
```

Note that there is no implicit conversion from a class to one of its members and that a class containing a member of another class does not override the virtual functions of that member. This contrasts with the public derivation case:

```
class D2 : public B {        // a D2 is a B
public:
    void f();          // overrides B::f()
};

void h2(D2* pd)
{
    B* pb = pd;        // ok: implicit D2* to B* conversion
    pb->g();           // calls B::g()
    pd->g();           // calls B::g()
    pb->f();           // virtual call: invokes D2::f()
    pd->f();           // invokes D2::f()
}
```

The notational convenience provided by the ***D2*** example compared to the ***D1*** example is a factor that can lead to overuse. It should be remembered, though, that there is a cost of increased dependency between ***B*** and ***D2*** to be paid for that notational convenience (see §24.3.2.1). In particular, it is easy to forget the implicit conversion from ***D2*** to ***B***. Unless such conversions are an acceptable part of the semantics of your classes, *public* derivation is to be avoided. When a class is used to represent a concept and derivation is used to represent an *is-a* relationship, such conversions are most often exactly what is desired.

There are cases in which you would like inheritance but cannot afford to have the conversion happen. Consider writing a class ***Cfield*** (controlled field) that – in addition to whatever else it does – provides run-time access control for another class ***Field***. At first glance, defining ***Cfield*** by deriving it from ***Field*** seems just right:

```
class Cfield : public Field { /* ... */ };
```

This expresses the notion that a ***Cfield*** really is a kind of ***Field***, allows notational convenience when writing a ***Cfield*** function that uses a member of the ***Field*** part of the ***Cfield***, and – most importantly – allows a ***Cfield*** to override ***Field*** virtual functions. The snag is that the ***Cfield**** to ***Field**** conversion implied in the declaration of ***Cfield*** defeats all attempts to control access to the ***Field***:

```
void g(Cfield* p)
{
    *p = "asdf";       // access to Field controlled by Cfield's assignment operator:
                       // p->Cfield::operator=("asdf")

    Field* q = p;      // implicit Cfield* to Field* conversion
    *q = "asdf";       // OOPS! no control
}
```

A solution would be to define ***Cfield*** to have a ***Field*** as a member, but doing that precludes ***Cfield*** from overriding ***Field*** virtual functions. A better solution would be to use *private* derivation:

```
class Cfield : private Field { /* ... */ };
```

From a design perspective, private derivation is equivalent to containment, except for the (occasionally essential) issue of overriding. An important use of this is the technique of deriving a class

publicly from an abstract base class that defines an interface and using private or protected derivation from a concrete class to provide an implementation (§2.5.4, §12.3, §25.3). Because the inheritance implied in *private* and *protected* derivation is an implementation detail that is not reflected in the type of the derived class, it is sometimes called *implementation inheritance* and contrasted to *public* derivation, whereby the interface of the base class is inherited and the implicit conversion to the base type is allowed. The latter is sometimes referred to as *subtyping*, or *interface inheritance*.

Another way of stating this is to point out that an object of a derived class should be usable wherever an object of its public base class is. This is sometimes called ''the Liskov Substitution Principle'' (§23.6[Liskov,1987]). The public/protected/private distinction supports this directly for polymorphic types manipulated through pointers and references.

24.3.4.1 Member/Hierarchy Tradeoffs [lang.mem]

To further examine the design choices involving containment and inheritance, consider how to represent a scrollbar in an interactive graphics system and how to attach a scrollbar to a window. We need two kinds of scrollbars: horizontal and vertical. We can represent this either by two types – ***Horizontal_scrollbar*** and ***Vertical_scrollbar*** – or by a single ***Scrollbar*** type that takes an argument that says whether its layout is horizontal or vertical. The former choice implies the need for a third type, the plain ***Scrollbar***, as the base class of the two specific scollbar types. The latter choice implies the need for an extra argument to the scrollbar type and the need to choose values to represent the two kinds of scrollbars. For example:

```
enum Orientation { horizontal, vertical };
```

Once a choice is made, it determines the kind of change needed to extend the system. In the scrollbar example, we might want to introduce a third type of scrollbar. We may originally have thought that there could be only two kinds of scrollbars (''after all, a window has only two dimensions''). However, in this case – as in most – there are possible extensions that surface as redesign issues. For example, one might like to use a ''navigation button'' instead of two scrollbars. Such a button would cause scrolling in different directions depending on where a user pressed it. Pressing the middle of the top would cause ''scrolling up,'' pressing the middle left would cause ''scrolling left,'' while pressing the top-left corner would cause ''scrolling up and left.'' Such buttons are not uncommon. They can be seen as a refinement of the notion of a scrollbar that is particularly suited to applications in which the information scrolled over isn't plain text but rather more general sorts of pictures.

Adding a navigation button to a program with a three-scrollbar class hierarchy involves adding a new class, but it requires no changes to the old scrollbar code:

This is the nice aspect of the ''hierarchical'' solution.

Passing the orientation of the scrollbar as an argument implies the presence of type fields in the scrollbar objects and the use of switch statements in the code of the scrollbar member functions.

That is, we are facing a tradeoff between expressing this aspect of the structure of the system in terms of declarations or in terms of code. The former increases the degree of static checking and the amount of information on which tools have to work. The latter postpones decisions to run time and allows changes to be made by modifying individual functions without affecting the overall structure of the system as seen by the type checker and other tools. In most situations, I recommend using a class hierarchy to directly model hierarchical relationships of the concepts.

The single scrollbar type solution makes it easy to store and pass information specifying a kind of scrollbar:

```
void helper(Orientation oo)
{
    // ...
    p = new Scrollbar(oo);
    // ...
}

void me()
{
    helper(horizontal);
    // ...
}
```

This representation would also make it easy to re-orient a scrollbar at run time. This is unlikely to be of major importance in the case of scrollbars, but it can be important for equivalent examples. The point here is that there are always tradeoffs, and the tradeoffs are often nontrivial.

24.3.4.2 Containment/Hierarchy Tradeoffs [lang.tradeoff]

Now consider how to attach a scrollbar to a window. If we consider a ***Window_with_scrollbar*** as something that is both a ***Window*** and a ***Scrollbar***, we get something like:

```
class Window_with_scrollbar : public Window, public Scrollbar {
    // ...
};
```

This allows any ***Window_with_scrollbar*** to act like a ***Scrollbar*** and like a ***Window***, but it constrains us to using the single scrollbar-type solution.

On the other hand, if we consider a ***Window_with_scrollbar*** as a ***Window*** that has a ***Scrollbar***, we get something like:

```
class Window_with_scrollbar : public Window {
    // ...
    Scrollbar* sb;
public:
    Window_with_scrollbar(Scrollbar* p, /* ... */) : Window(/* ...*/), sb(p) { /* ... */ }
    // ...
};
```

This allows us to use the scrollbar-hierarchy solution. Passing the scrollbar as an argument allows the window to be oblivious to the exact type of its scrollbar. We could even pass a ***Scrollbar***

around the way we passed an ***Orientation*** (§24.3.4.1). If we need to have ***Window_with_scrollbar*** act as a scrollbar, we can add a conversion operator:

```
Window_with_scrollbar::operator Scrollbar&()
{
    return *sb;
}
```

My preference is to have a window contain a scrollbar. I find it easier to think of a window *having* a scrollbar than of a window *being* a scrollbar in addition to being a window. In fact, my favorite design strategy involves a scrollbar being a special kind of window, which is then contained in a window that needs scrollbar services. This strategy forces the decision in favor of the containment solution. An alternative argument for the containment solution comes from the ''can it have two?'' rule of thumb (§24.3.4). Because there is no logical reason why a window shouldn't have two scrollbars (in fact, many windows do have both a horizontal and a vertical scrollbar), ***Window_with_scrollbar*** ought not be derived from ***Scrollbar***.

Note that it is not possible to derive from an unknown class. The exact type of a base class must be known at compile time (§12.2). On the other hand, if an attribute of a class is passed as an argument to its constructor, then somewhere in the class there must be a member that represents it. However, if that member is a pointer or a reference we can pass an object of a class derived from the class specified for the member. For example, The ***Scrollbar**** member ***sb*** in the previous example can point to a ***Scrollbar*** of a type, such as ***Navigation_button***, that is unknown to users of the ***Scrollbar****.

24.3.5 Use Relationships [lang.use]

Knowledge of what other classes are used by a class and in which ways is often critical in order to express and understand a design. Such dependencies are supported only implicitly by C++. A class can use only names that have been declared (somewhere), but a list of names used is not provided in the C++ source. Tools (or in the absence of suitable tools, careful reading) are necessary for extracting such information. The ways a class ***X*** can use another class ***Y*** can be classified in several ways. Here is one way:

- ***X*** uses the name ***Y***.
- ***X*** uses ***Y***.
  - ***X*** calls a ***Y*** member function.
  - ***X*** reads a member of ***Y***.
  - ***X*** writes a member of ***Y***.
- ***X*** creates a ***Y***.
  - ***X*** allocates an ***auto*** or ***static*** variable of ***Y***.
  - ***X*** creates a ***Y*** using ***new***.
- ***X*** takes the size of a ***Y***.

Taking the size of an object is classified separately because doing so requires knowledge of the class declaration, but doesn't depend on the constructors. Naming ***Y*** is also classified as a separate dependency because just doing that – for example, in declaring a ***Y**** or mentioning ***Y*** in the declaration of an external function – doesn't require access to the declaration of ***Y*** at all (§5.7):

```
class Y;    // Y is the name of a class
Y* p;
extern Y f(const Y&);
```

It is often important to distinguish between the dependencies of a class' interface (the class declaration) and the dependencies of the class implementation (the class member definitions). In a well-designed system, the latter typically have many more dependencies, and those are far less interesting to a user than are the dependencies of the class declaration (§24.4.2). Typically, a design aims at minimizing the dependencies of an interface because they become dependencies of the class' users (§8.2.4.1, §9.3.2, §12.4.1.1, §24.4) .

C++ doesn't require the implementer of a class to specify in detail what other classes are used and how. One reason for this is that most significant classes depend on so many other classes, that an abbreviation of the list of those classes, such as an ***#include*** directive, would be necessary for readability. Another is that the classification and granularity of such dependencies doesn't appear to be a programming language issue. Rather, exactly how *uses* dependencies are viewed depends on the purpose of the designer, programmer, or tool. Finally, which dependencies are interesting may also depend on details of the language implementation.

24.3.6 Programmed-In Relationships [lang.prog]

A programming language cannot – and should not – directly support every concept from every design method. Similarly, a design language should not support every feature of every programming language. A design language should be richer and less concerned with details than a language suitable for systems programming must be. Conversely, a programming language must be able to support a variety of design philosophies, or it will fail for lack of adaptability.

When a programming language does not provide facilities for representing a concept from the design directly, a conventional mapping between the design construct and the programming language constructs should be used. For example, a design method may have a notion of delegation. That is, the design can specify that every operation not defined for a class ***A*** should be serviced by an object of a class ***B*** pointed to by a pointer ***p***. C++ cannot express this directly. However, the expression of that idea in C++ is so stylized that one could easily imagine a program generating the code. Consider:

```
class B {
    // ...
    void f();
    void g();
    void h();
};

class A {
    B* p;
    // ...
    void f();
    void ff();
};
```

A specification that ***A*** delegated to ***B*** through ***A::p*** would result in code like this:

```
class A {
    B* p;       // delegation through p
    // ...
    void f();
    void ff();
    void g() { p->g(); }      // delegate g()
    void h() { p->h(); }      // delegate h()
};
```

It is fairly obvious to a programmer what is going on here, but simulating a design concept in code is clearly inferior to a one-to-one correspondence. Such ''programmed-in'' relationships are not as well ''understood'' by the programming language and are therefore less amenable to manipulation by tools. For example, standard tools would not recognize the ''delegation'' from ***A*** to ***B*** through ***A*::*p*** as different from any other use of a ***B****.

A one-to-one mapping between the design concepts and the programming language concepts should be used wherever possible. A one-to-one mapping ensures simplicity and guarantees that the design really is reflected in the program so that programmers and tools can take advantage of it.

Conversion operators provide a language mechanism for expressing a class of programmed-in relationships. That is, a conversion operator ***X*::*operator Y*( )** specifies that wherever a ***Y*** is acceptable, an ***X*** can be used (§11.4.1). A constructor ***Y*::*Y*(*X*)** expresses the same relationship. Note that a conversion operator (and a constructor) produces a new object rather than changing the type of an existing object. Declaring a conversion function to ***Y*** is simply a way of requesting *implicit* application of a function that returns a ***Y***. Because the implicit application of conversions defined by constructors and conversion operators can be treacherous, it is sometimes useful to analyze them separately in a design.

It is important to ensure that the conversion graphs for a program do not contain cycles. If they do, the resulting ambiguity errors will render the types involved in the cycles unusable in combination. For example:

```
class Rational;

class Big_int {
public:
    friend Big_int operator+(Big_int, Big_int);
    operator Rational();
    // ...
};

class Rational {
public:
    friend Rational operator+(Rational, Rational);
    operator Big_int();
    // ...
};
```

The ***Rational*** and ***Big_int*** types will not interact as smoothly as one might have hoped:

```
void f(Rational r, Big_int i)
{
    g(r+i);                 // error, ambiguous: operator+(r,Rational(i)) or operator+(Big_int(r),i) ?
    g(r+Rational(i));       // one explicit resolution
    g(Big_int(r)+i);        // another explicit resolution
}
```

One can avoid such ''mutual'' conversions by making at least some of them explicit. For example, the ***Big_int*** to ***Rational*** conversion might have been defined as ***make_Rational***( ) instead of as a conversion operator, and the addition would have been resolved to ***g***(***Big_int***(***r***), ***i***). Where ''mutual'' conversion operators cannot be avoided, one must resolve the resulting clashes either by explicit conversions as shown or by defining many separate versions of binary operators, such as +.

24.3.7 Relationships within a Class [lang.within]

A class can conceal just about any implementation detail and just about any amount of dirt – and sometimes it has to. However, the objects of most classes do themselves have a regular structure and are manipulated in ways that are fairly easy to describe. An object of a class is a collection of other sub-objects (often called members), and many of these are pointers and references to other objects. Thus, an object can be seen as the root of a tree of objects and the objects involved can be seen as constituting an ''object hierarchy'' that is complementary to the class hierarchy, as described in §24.3.2.1. For example, consider a very simple ***String***:

```
class String {
    int sz;
    char* p;
public:
    String(const char* q);
    ~String();
    // ...
};
```

A ***String*** object can be represented graphically like this:

24.3.7.1 Invariants [lang.invariant]

The values of the members and the objects referred to by members are collectively called the *state* of the object (or simply, its *value*). A major concern of a class design is to get an object into a well-defined state (initialization/construction), to maintain a well-defined state as operations are performed, and finally to destroy the object gracefully. The property that makes the state of an object well-defined is called its *invariant*.

Thus, the purpose of initialization is to put an object into a state for which the invariant holds. Typically, this is done by a constructor. Each operation on a class can assume it will find the invariant true on entry and must leave the invariant true on exit. The destructor finally invalidates the invariant by destroying the object. For example, the constructor *String::String(const char*)* ensures that *p* points to an array of at least *sz+1* elements, where *sz* has a reasonable value and *p[sz]==0*. Every string operation must leave that assertion true.

Much of the skill in class design involves making a class simple enough to make it possible to implement it so that it has a useful invariant that can be expressed simply. It is easy enough to state that every class needs an invariant. The hard part is to come up with a useful invariant that is easy to comprehend and that doesn't impose unacceptable constraints on the implementer or on the efficiency of the operations. Note that ''invariant'' here is used to denote a piece of code that can potentially be run to check the state of an object. A stricter and more mathematical notion is clearly possible and, in some contexts, more appropriate. An invariant, as discussed here, is a practical – and therefore typically economical and logically incomplete – check on an object's state.

The notion of invariants has its origins in the work of Floyd, Naur, and Hoare on preconditions and postconditions and is present in essentially all work on abstract data types and program verification done over the last 30 years or so. It is also a staple of C debugging.

Typically, the invariant is not maintained during the execution of a member function. Functions that may be called while the invariant is invalid should not be part of the public interface. Private and protected functions can serve that purpose.

How can we express the notion of an invariant in a C++ program? A simple way is to define an invariant-checking function and insert calls to it in the public operations. For example:

```
class String {
        int sz;
        char* p;
public:
        class Range {};                         // exception classes
        class Invariant {};

        enum { TOO_LARGE = 16000 };             // length limit

        void check();                           // invariant check

        String(const char* q);
        String(const String&);
        ~String();

        char& operator[](int i);
        int size() { return sz; }

        // ...
};

void String::check()
{
        if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz-1]) throw Invariant();
}
```

```
char& String::operator[](int i)
{
    check();                                  // check on entry
    if (i<0 || sz<=i) throw Range();          // do work
    check();                                  // check on exit
    return p[i];
}
```

This will work nicely and is hardly any work for the programmer. However, for a simple class like ***String*** the invariant checking will dominate the run time and maybe even the code size. Therefore, programmers often execute the invariant checks only during debugging:

```
inline void String::check()
{
#ifndef NDEBUG
    if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant();
#endif
}
```

Here, the ***NDEBUG*** macro is used in a way similar to the way it is used in the standard C ***assert*( )** macro. ***NDEBUG*** is conventionally set to indicate that debugging is *not* being done.

The simple act of defining invariants and using them during debugging is an invaluable help in getting the code right and – more importantly – in getting the concepts represented by the classes well defined and regular. The point is that when you are designing invariants, a class will be considered from an alternative viewpoint and the code will contain redundancy. Both increase the likelihood of spotting mistakes, inconsistencies, and oversights.

24.3.7.2 Assertions [lang.assert]

An invariant is a special form of an assertion. An assertion is simply a statement that a given logical criterion must hold. The question is what to do when it doesn't.

The C standard library – and by implication the C++ standard library – provides the ***assert*( )** macro in ***<cassert>*** or ***<assert.h>***. An ***assert*( )** evaluates its argument and calls ***abort*( )** if the result is nonzero. For example:

```
void f(int* p)
{
    assert(p!=0);   // assert that p!=0; abort() if p is zero
    // ...
}
```

Before aborting, ***assert*( )** outputs the name of its source file and the number of the line on which it appears. This makes ***assert*( )** a useful debugging aid. ***NDEBUG*** is usually set by compiler options on a per-compilation-unit basis. This implies that ***assert*( )** shouldn't be used in inline functions and template functions that are included in several translation units unless great care is taken that ***NDEBUG*** is set consistently (§9.2.3). Like all macro magic, this use of ***NDEBUG*** is too low-level, messy, and error-prone. Also, it is typically a good idea to leave at least some checks active in even the best-checked program, and ***NDEBUG*** isn't well suited for that. Furthermore, calling ***abort*( )** is rarely acceptable in production code.

The alternative is to use an *Assert*( ) template that throws an exception rather than aborting so that assertions can be left in production code when that is desirable. Unfortunately, the standard library doesn't provide an *Assert*( ). However, it is trivially defined:

```
template<class X, class A> inline void Assert(A assertion)
{
    if (!assertion) throw X();
}
```

Assert( ) throws the exception *X*( ) if the *assertion* is false. For example:

```
class Bad_arg { };

void f(int* p)
{
    Assert<Bad_arg>(p!=0);  // assert p!=0; throw Bad_arg unless p!=0
    // ...
}
```

This style of assertion has the condition explicit, so if we want to check only while debugging we must say so. For example:

```
void f2(int* p)
{
    Assert<Bad_arg>(NDEBUG || p!=0);    // either I'm not debugging or p!=0
    // ...
}
```

The use of || rather than && in the assertion may appear surprising. However, *Assert<E>*(*a*||*b*) tests !(*a*||*b*) which is !*a*&&!*b*.

Using *NDEBUG* in this way requires that we define *NDEBUG* with a suitable value whether or not we are debugging. A C++ implementation does not do this for us by default, so it is better to use a value. For example:

```
#ifdef NDEBUG
const bool ARG_CHECK = false;       // we are not debugging: disable checks
#else
const bool ARG_CHECK = true;        // we are debugging
#endif

void f3(int* p)
{
    Assert<Bad_arg>(!ARG_CHECK || p!=0);    // either I'm not debugging or p!=0
    // ...
}
```

If the exception associated with an assertion is not caught, a failed *Assert*( ) *terminate*( )s the program much like an equivalent *assert*( ) would *abort*( ). However, an exception handler may be able to take some less drastic action.

In any realistically-sized program, I find myself turning assertions on and off in groups to suit the need for testing. Using *NDEBUG* is simply the crudest form of that technique. Early on in development, most assertions are enabled, whereas only key sanity checks are left enabled in

shipped code. This style of usage is most easily managed if the actual assertion is in two parts, with the first being an enabling condition (such as *ARG_CHECK*) and the second being the assertion proper.

If the enabling condition is a constant expression, the whole assertion will be compiled away when not enabled. However, the enabling condition can also be a variable so that it can be turned on and off at run time as debugging needs dictate. For example:

```
bool string_check = true;

inline void String::check()
{
    Assert<Invariant>(!string_check || (p && 0<=sz && sz<TOO_LARGE && p[sz]==0));
}

void f()
{
    String s = "wonder";
    // strings are checked here
    string_check = false;
    // no checking of strings here
}
```

Naturally, code will be generated in such cases, so we must keep an eye out for code bloat if we use such assertions extensively.

Saying

```
Assert<E>(a);
```

is simply another way of saying

```
if (!a) throw E();
```

Then why bother with *Assert*( ), rather than writing out the statement directly? Using *Assert*( ) makes the designer's intent explicit. It says that this is an assertion of something that is supposed to be always true. It is not an ordinary part of the program logic. This is valuable information to a reader of the program. A more practical advantage is that it is easy to search for *assert*( ) or *Assert*( ) whereas searching for conditional statements that throw exceptions is nontrival.

Assert( ) can be generalized to throw exceptions taking arguments and variable exceptions:

```
template<class A, class E> inline void Assert(A assertion, E except)
{
    if (!assertion) throw except;
}

struct Bad_g_arg {
    int* p;
    Bad_g_arg(int* pp) :p(pp) { }
};

bool g_check = true;
int g_max = 100;
```

```
void g(int* p, exception e)
{
    Assert(!g_check || p!=0, e);                          // pointer is valid
    Assert(!g_check || (0<*p&&*p<=g_max), Bad_g_arg(p));  // value is plausible
    // ...
}
```

In many programs, it is crucial that no code is generated for an *Assert*( ) where the assertion can be evaluated at compile time. Unfortunately, some compilers are unable to achieve this for the generalized *Assert*( ). Consequently, the two-argument *Assert*( ) should be used only when the exception is not of the form *E*( ) and it is also acceptable for some code to be generated independently of the value of the assertion.

In §23.4.3.5, it was mentioned that the two most common forms of class hierarchy reorganizations were to split a class into two and to factor out the common part of two classes into a base class. In both cases, well-designed invariants can give a clue to the potential for reorganization. Comparing the invariant with the code of operations will show most of the invariant checking to be redundant in a class that is ripe for splitting. In such cases, subsets of the operations will access only subsets of the object state. Conversely, classes that are ripe for merging will have similar invariants even if their detailed implementations differ.

24.3.7.3 Preconditions and Postconditions [lang.pre]

One popular use of assertions is to express preconditions and postconditions of a function. That is, checking that basic assumptions about input hold and verifying that the function leaves the world in the expected state upon exit. Unfortunately, the assertions we would like to make are often at a higher level than the programming language allows us to express conveniently and efficiently. For example:

```
template<class Ran> void sort(Ran first, Ran last)
{
    Assert<Bad_sequence>("[first, last) is a valid sequence");   // pseudo code

    // ... sorting algorithm ...

    Assert<Failed_sort>("[first, last) is in increasing order");   // pseudo code
}
```

This problem is fundamental. What we want to say *about* a program is best expressed in a mathematically-based higher language, rather than in the algorithmic programming language *in which* we write the program.

As for invariants, a certain amount of cleverness is needed to translate the ideal of what we would like to assert into something that is algorithmically feasible to check. For example:

```
template<class Ran> void sort(Ran first, Ran last)
{
    // [first,last) is a valid sequence: check plausibility:
    Assert<Bad_sequence>(NDEBUG || first<=last);

    // ... sorting algorithm ...
```

```
        // [first,last) is in increasing order: check a sample:
        Assert<Failed_sort>(NDEBUG ||
                (last-first<2 || (*first<=last[-1]
                        && *first<=first[(last-first)/2] && first[(last-first)/2]<=last[-1])));
}
```

I often find writing ordinary code-checking arguments and results simpler than composing assertions. However, it is important to try to express the real (ideal) preconditions and postconditions – and at least document them as comments – before reducing them to something less abstract that can be effectively expressed in a programming language.

Precondition checking can easily degenerate into simple checking of argument values. As an argument is often passed through several functions, this checking can be repetitive and expensive. However, simply asserting that every pointer argument is nonzero in every function is not particularly helpful and can give a false sense of security – especially if the tests are done during debugging only to prevent overhead. This is a major reason why I recommend a focus on invariants.

24.3.7.4 Encapsulation [lang.encapsulate]

Note that in C++, the class – not the individual object – is the unit of encapsulation. For example:

```
class List {
        List* next;
public:
        bool on(List*);
        // ...
};

bool List::on(List* p)
{
        if (p == 0) return false;
        for(List* q = this; q; q=q->next) if (p == q) return true;
        return false;
}
```

The chasing of the private ***List::next*** pointer is accepted because ***List::on()*** has access to every object of class ***List*** it can somehow reference. Where that is inconvenient, matters can be simplified by not taking advantage of the ability to access the representation of other objects from a member function. For example:

```
bool List::on(List* p)
{
        if (p == 0) return false;
        if (p == this) return true;
        if (next==0) return false;
        return next->on(p);
}
```

However, this turns iteration into recursion, and doing that can cause a major performance hit when a compiler isn't able to optimize the recursion back into an iteration.

24.4 Components [lang.component]

The unit of design is a collection of classes, functions, etc., rather than an individual class. Such a collection, often called a *library* or a *framework* (§25.8), is also the unit of reuse (§23.5.1), maintenance, etc. C++ provides three mechanisms for expressing the notion of a set of facilities united by a logical criteria:

[1] A class – containing a collection of data, function, template, and type members

[2] A class hierarchy – containing a collection of classes

[3] A namespace – containing a collection of data, function, template, and type members

A class provides many facilities to make it convenient to create objects of the type it defines. However, many significant components are not best described by a mechanism for creating objects of a single type. A class hierarchy expresses the notion of a set of related types. However, the individual members of a component are not always best expressed as classes and not all classes possess the basic similarity required to fit into a meaningful class hierarchy (§24.2.5). Therefore, a namespace is the most direct and the most general embodiment of the notion of a component in C++. A component is sometimes referred to as a ''class category.'' However, not every element of a component is or should be a class.

Ideally, a component is described by the set of interfaces it uses for its implementation plus the set of interfaces it provides for its users. Everything else is ''implementation detail'' and hidden from the rest of the system. This may indeed be the designer's description of a component. To make it real, the programmer needs to map it into declarations. Classes and class hierarchies provide the interfaces, and namespaces allow the programmer to group the interfaces and to separate interfaces used from interfaces provided. Consider:

Using the techniques described in §8.2.4.1, this becomes:

```
namespace A { // some facilities used by X's interface
        // ...
}

namespace X { // interface of component X

        using namespace A; // dependent on declarations from A
        // ...
        void f();
}
```

```
namespace X_impl { // facilities needed by X's implementation
    using namespace X;
    // ...
}

void X::f()
{
    using namespace X_impl; // dependent on declarations from X_impl
    // ...
}
```

The general interface ***X*** should not depend on the implementation interface ***X_impl***.

A component can have many classes that are not intended for general use. Such classes should be ''hidden'' within implementation classes or namespaces:

```
namespace X_impl { // component X implementation details

    class Widget {
        // ...
    };

    // ...
}
```

This ensures that ***Widget*** isn't used from other parts of the program. However, classes that represent coherent concepts are often candidates for reuse and should therefore be considered for inclusion into the interface of the component. Consider:

```
class Car {
    class Wheel {
        // ...
    };

    Wheel flw, frw, rlw, rrw;
    // ...
public:
    // ...
};
```

In most contexts, we need to have the actual wheels hidden to maintain the abstraction of a car (when you use a car you cannot operate the wheels independently). However, the ***Wheel*** class itself seems a good candidate for wider use, so moving it outside class ***Car*** might be better:

```
class Wheel {
    // ...
};

class Car {
    Wheel flw, frw, rlw, rrw;
    // ...
public:
    // ...
};
```

The decision to nest or not depends on the aims of the design and the generality of the concepts involved. Both nesting and ‘‘non-nesting’’ are widely applicable techniques for expressing a design. The default should be to make a class as local as possible until a need to make it more generally available is demonstrated.

There is a nasty tendency for ‘‘interesting’’ functions and data to ‘‘bubble up’’ to the global namespace, to widely-used namespaces, or to ultimate base classes in a hierarchy. This can easily lead to unintentional exposure of implementation details and to the problems associated with global data and global functions. This is most likely to happen in a single-rooted hierarchy, and in a program where only very few namespaces are used. Virtual base classes (§15.2.4) can be used to combat this phenomenon in the context of class hierarchies. Small ‘‘implementation’’ namespaces are the main tool for avoiding the problem in the context of namespaces.

Note that header files provide a powerful mechanism for supplying different views of a component to different users and for excluding classes that are considered part of the implementation from the user’s view (§9.3.2).

24.4.1 Templates [lang.temp]

From a design perspective, templates serve two, weakly-related needs:

- Generic programming
- Policy parameterization

Early in a design effort, operations are just operations. Later, when it is time to specify the type of operands templates become essential when using a statically-typed programming language, such as C++. Without templates, function definitions would have to be replicated or checking would have to be unnecessarily postponed to run time (§24.2.3). An operation that implements an algorithm for a variety of operand types is a candidate to be implemented as a template. If all operands fit into a single class hierarchy, and especially if there is a need to add new operand types at run time, the operand type is best represented as a class – often as an abstract class. If the operand types do not fit into a single hierarchy and especially if run-time performance is critical, the operation is best implemented as a template. The standard containers and their supporting algorithms are an example of when the need to take operands of a variety of unrelated types combined with a need for run-time performance lead to the use of templates (§16.2).

To make the template/hierarchy tradeoff more concrete, consider how to generalize a simple iteration:

```
void print_all(Iter_for_T x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

Here, the assumption is that ***Iter_for_T*** provides operations that yield ***T****s.

We can make the iterator ***Iter_for_T*** a template parameter:

```
template<class Iter_for_T> print_all(Iter_for_T x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

This allows us to use a variety of unrelated iterators as long as they all provide ***first***( ) and ***next***( ) with the right meanings and as long as we know the type of iterator for each call of ***print_all***( ) at compile time. The standard library containers and algorithms are based on this idea.

Alternatively, we can use the observation that ***first***( ) and ***next***( ) constitute an interface to iterators. We can then define a class to represent that interface:

```
class Iter {
public:
        virtual T* first( ) const = 0;
        virtual T* next( ) = 0;
};

void print_all2(Iter& x)
{
        for (T* p = x.first( ); p; p = x.next( )) cout << *p;
}
```

We can now use every iterator derived from ***Iter***. The actual code doesn't differ depending on whether we use templates or a class hierarchy to represent the parameterization – only the run-time, recompilation, etc., tradeoffs differ. In particular, class ***Iter*** is a candidate for use as an argument for the template:

```
void f(Iter& i)
{
        print_all(i);      // use the template
        print_all2(i);
}
```

Consequently, the two approaches can be seen as complementary.

Often, a template needs to use functions and classes as part of its implementation. Many of those must themselves be templates so as to maintain generality and efficiency. In that way, algorithms become generic over a range of types. This style of template use is called *generic programming* (§2.7). When we call ***std***::***sort***( ) on a ***vector***, the elements of the vector are the operands of the ***sort***( ); thus, ***sort***( ) is generic for the element types. In addition, the standard sort is generic for the container types because it is invoked on iterators for arbitrary, standard-conforming containers (§16.3.1).

The ***sort***( ) algorithm is also parameterized on the comparison criteria (§18.7.1). From a design perspective, this is different from taking an operation and making it generic on its operand type. Deciding to parameterize an algorithm on an object (or operation) in a way that controls the way the algorithm operates is a much higher-level design decision. It is a decision to give the designer/programmer control over some part of the policy governing the operation of the algorithm. From a programming language point of view, however, there is no difference.

24.4.2 Interfaces and Implementations [lang.interface]

The ideal interface

- presents a complete and coherent set of concepts to a user,
- is consistent over all parts of a component,

- does not reveal implementation details to a user,
- can be implemented in several ways,
- is statically typed,
- is expressed using application-level types, and
- depends in limited and well-defined ways on other interfaces.

Having noted the need for consistency across the classes that present the component's interface to the rest of the world (§24.4), we can simplify the discussion by looking at only a single class. Consider:

```
class Y { /* ... */ };        // needed by X

class Z { /* ... */ };        // needed by X

class X {  // example of poor interface style
        Y a;
        Z b;
public:
        void f(const char * ...);
        void g(int[], int);
        void set_a(Y&);
        Y& get_a();
};
```

This interface has several potential problems:

- The interface uses the types ***Y*** and ***Z*** in a way that requires the declarations of ***Y*** and ***Z*** to be known to compile it.
- The function ***X*::*f*()** takes an arbitrary number of arguments of unknown types (probably somehow controlled by a ''format string'' supplied as the first argument; §21.8).
- The function ***X*::*g*()** takes an ***int*[]** argument. This may be acceptable, but typically it is a sign that the level of abstraction is too low. An array of integers is not self-describing, so it is not obvious how many elements it is supposed to have.
- The ***set_a*()** and ***get_a*()** functions most likely expose the representation of objects of class ***X*** by allowing direct access to ***X*::*a***.

These member functions provide an interface at a very low level of abstraction. Basically, classes with interfaces at this level belong among the implementation details of a larger component – if they belong anywhere at all. Ideally, an argument of an interface function carries enough information to make it self-describing. A rule of thumb is that it should be possible to transmit the request for service over a thin wire for service at a remote server.

C++ allows the programmer to expose the representation of a class as part of the interface. This representation may be hidden (using ***private*** or ***protected***), but it is available to the compiler to allow allocation of automatic variables, to allow inline substitution of functions, etc. The negative effect of this is that use of class types in the representation of a class may introduce undesirable dependencies. Whether the use of members of types ***Y*** and ***Z*** is a problem depends on what kind of types ***Y*** and ***Z*** actually are. If they are simple types, such as ***list***, ***complex***, and ***string***, their use is most often quite appropriate. Such types can be considered stable, and the need to include their class declarations is an acceptable burden on the compiler. However, if ***Y*** and ***Z*** themselves had been interface classes of significant components, such as a graphics system or a bank account

management system, it might be wise not to depend too directly on them. In such cases, using a pointer or a reference member is often a better choice:

```
class Y;
class Z;

class X { // X accesses Y and Z through pointers and references only
    Y* a;
    Z& b;
    // ...
};
```

This decouples the definition of ***X*** from the definitions of ***Y*** and ***Z***; that is, the definition of ***X*** depends on the names ***Y*** and ***Z*** only. The implementation of ***X*** will, of course, still depend on the definitions of ***Y*** and ***Z***, but this will not adversely affect the users of ***X***.

This illustrates an important point: an interface that hides significant amounts of information – as a useful interface ought to – will have far fewer dependencies than the implementation it hides. For example, the definition of class ***X*** can be compiled without access to the definitions of ***Y*** and ***Z***. However, the definitions of ***X***'s member functions that manipulate the ***Y*** and ***Z*** objects will need access to the definitions of ***Y*** and ***Z***. When dependencies are analyzed, the dependencies of the interface and the implementation must be considered separately. In both cases, the ideal is for the dependency graphs of a system to be directed acyclic graphs to ease understanding and testing of the system. However, this ideal is far more critical and far more often achievable for interfaces than for implementations.

Note that a class can define three interfaces:

```
class X {
private:
    // accessible to members and friends only
protected:
    // accessible to members and friends and
    // to members and friends of derived classes only
public:
    // accessible to the general public
};
```

In addition, a ***friend*** is part of the public interface (§11.5).

A member should be part of the most restrictive interface possible. That is, a member should be ***private*** unless there is a reason for it to be more accessible. If it needs to be more accessible, it should be ***protected*** unless there is a reason for it to be ***public***. It is almost always a bad idea to make a data member ***public*** or ***protected***. The functions and classes that constitute the public interface should present a view of the class that fits with its role as representing a concept.

Note that abstract classes can be used to provide a further level of representation hiding (§2.5.4, §12.3, §25.3).

24.4.3 Fat Interfaces [lang.fat]

Ideally, an interface should offer only operations that make sense and that can be implemented well by every derived class implementing that interface. However, this is not always easy. Consider lists, arrays, associative arrays, trees, etc. As shown in §16.2.2, it is tempting and sometimes useful to provide a generalization of all of these types – usually called a *container* – that can be used as the interface to every one of these. This (apparently) relieves the user of having to deal with the details of all of these containers. However, defining the interface of a general container class is nontrivial. Assume that we want to define ***Container*** as an abstract type. What operations do we want ***Container*** to provide? We could provide only the operations that every container can support – the intersection of the sets of operations – but that is a ridiculously narrow interface. In fact, in many interesting cases that intersection is empty. Alternatively, we could provide the union of all the sets of operations and give a run-time error if a ''non-existent'' operation is applied to an object through this interface. An interface that is such a union of interfaces to a set of concepts is called a *fat interface*. Consider a ''general container'' of objects of type ***T***:

```
class Container {
public:
    struct Bad_oper {     // exception class
        const char* p;
        Bad_oper(const char* pp) : p(pp) { }
    };

    virtual void put(const T*) { throw Bad_oper("Container::put"); }
    virtual T* get() { throw Bad_oper("Container::get"); }

    virtual T*& operator[](int) { throw Bad_oper("Container::[](int)"); }
    virtual T*& operator[](const char*) { throw Bad_oper("Container::[](char*)"); }
    // ...
};
```

Containers could then be declared like this:

```
class List_container : public Container, private list {
public:
    void put(const T*);
    T* get();
    // ... no operator[] ...
};

class Vector_container : public Container, private vector {
public:
    T*& operator[](int);
    T*& operator[](const char*);
    // ... no put() or get() ...
};
```

As long as one is careful, all is well:

```
void f()
{
    List_container sc;
    Vector_container vc;
    // ...
    user(sc,vc);
}

void user(Container& c1, Container& c2)
{
    T* p1 = c1.get();
    T* p2 = c2[3];
    // don't use c2.get() or c1[3]
    // ...
}
```

However, few data structures support both the subscripting and the list-style operations well. Consequently, it is probably not a good idea to specify an interface that requires both. Doing so leads to the use of run-time type-inquiry (§15.4) or exception handling (Chapter 14) to avoid run-time errors. For example:

```
void user2(Container& c1, Container& c2) // detection is easy, but recovery can be hard
{
    try {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // ...
    }
    catch(Container::Bad_oper& bad) {
        // Oops!
        // Now what?
    }
}
```

or

```
void user3(Container& c1, Container& c2) // early detection is tedious; recovery can still be hard
{
    if (dynamic_cast<List_container*>(&c1) && dynamic_cast<Vector_container*>(&c2)) {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // ...
    }
    else {
        // Oops!
        // Now what?
    }
}
```

In both cases, run-time performance can suffer and the generated code can be surprisingly large. As a result, people are tempted to ignore the potential errors and hope that they don't actually occur

when the program is in the hands of users. The problem with this approach is that exhaustive testing is also hard and expensive.

Consequently, fat interfaces are best avoided where run-time performance is at a premium, where strong guarantees about the correctness of code are required, and in general wherever there is a good alternative. The use of fat interfaces weakens the correspondence between concepts and classes and thus opens the floodgates for the use of derivation as a mere implementation convenience.

24.5 Advice [lang.advice]

[1] Evolve use towards data abstraction and object-oriented programming; §24.2.
[2] Use C++ features and techniques as needed (only); §24.2.
[3] Match design and programming styles; §24.2.1.
[4] Use classes/concepts as a primary focus for design rather than functions/processing; §24.2.1.
[5] Use classes to represent concepts; §24.2.1, §24.3.
[6] Use inheritance to represent hierarchical relationships between concepts (only); §24.2.2, §24.2.5, §24.3.2.
[7] Express strong guarantees about interfaces in terms of application-level static types; §24.2.3.
[8] Use program generators and direct-manipulation tools to ease well-defined tasks; §24.2.4.
[9] Avoid program generators and direct-manipulation tools that do not interface cleanly with a general-purpose programming language; §24.2.4.
[10] Keep distinct levels of abstraction distinct; §24.3.1.
[11] Focus on component design; §24.4.
[12] Make sure that a virtual function has a well-defined meaning and that every overriding function implements a version of that desired behavior; §24.3.4, §24.3.2.1.
[13] Use public inheritance to represent *is-a* relationships; §24.3.4.
[14] Use membership to represent *has-a* relationships; §24.3.4.
[15] Prefer direct membership over a pointer to a separately-allocated object for expressing simple containment; §24.3.3, §24.3.4.
[16] Make sure that the *uses* dependencies are understood, non-cyclic wherever possible, and minimal; §24.3.5.
[17] Define invariants for all classes; §24.3.7.1.
[18] Explicitly express preconditions, postconditions, and other assertions as assertions (possibly using ***Assert***( )); §24.3.7.2.
[19] Define interfaces to reveal the minimal amount of information needed; §24.4.
[20] Minimize an interface's dependencies on other interfaces; §24.4.2.
[21] Keep interfaces strongly typed; §24.4.2.
[22] Express interfaces in terms of application-level types; §24.4.2.
[23] Express an interface so that a request could be transmitted to a remote server; §24.4.2.
[24] Avoid fat interfaces; §24.4.3.
[25] Use ***private*** data and member functions wherever possible; §24.4.2.
[26] Use the ***public/protected*** distinction to distinguish between the needs of designers of derived classes and general users; §24.4.2.

[27] Use templates for generic programming; §24.4.1.
[28] Use templates to parameterize an algorithm by a policy; §24.4.1.
[29] Use templates where compile-time type resolution is needed; §24.4.1.
[30] Use class hierarchies where run-time type resolution is needed; §24.4.1.

25

Roles of Classes

Some things better change ...
but fundamental themes
should revel in persistence.
– Stephen J. Gould

Kinds of classes — concrete types — abstract types — nodes — changing interfaces — object I/O — actions — interface classes — handles — use counts — application frameworks — advice — exercises.

25.1 Kinds of Classes [role.intro]

The C++ class is a programming language construct that serves a variety of design needs. In fact, I find that the solution to most knotty design problems involves the introduction of a new class to represent some notion that had been left implicit in the previous draft design (and maybe the elimination of other classes). The great variety of roles that a class can play leads to a variety of kinds of classes that are specialized to serve a particular need well. In this chapter, a few archetypical kinds of classes are described, together with their inherent strengths and weaknesses:

§25.2 Concrete types
§25.3 Abstract types
§25.4 Nodes
§25.5 Operations
§25.6 Interfaces
§25.7 Handles
§25.8 Application frameworks

These ‘‘kinds of classes’’ are design notions and not language constructs. The unattained, and probably unattainable, ideal is to have a minimal set of simple and orthogonal kinds of classes from which all well-behaved and useful classes could be constructed. It is important to note that each of

these kinds of classes has a place in design and none is inherently better than the others for all uses. Much confusion in discussions of design and programming comes from people trying to use only one or two kinds of classes exclusively. This is usually done in the name of simplicity, yet it leads to contorted and unnatural uses of the favored kinds of classes.

The description here emphasizes the pure forms of these kinds of classes. Naturally, hybrid forms can also be used. However, a hybrid ought to appear as the result of a design decision based on an evaluation of the engineering tradeoffs and not a result of some misguided attempt to avoid making decisions. ''Delaying decisions'' is too often a euphemism for ''avoiding thinking.'' Novice designers will usually do best by staying away from hybrids and also by following the style of an existing component with properties that resemble the desired properties for the new component. Only experienced programmers should attempt to write a general-purpose component or library, and every library designer should be ''condemned'' to use, document, and support his or her creation for some years. Also, please note §23.5.1.

25.2 Concrete Types [role.concrete]

Classes such as ***vector*** (§16.3), ***list*** (§17.2.2), ***Date*** (§10.3), and ***complex*** (§11.3, §22.5) are *concrete* in the sense that each is the representation of a relatively simple concept with all the operations essential for the support of that concept. Also, each has a one-to-one correspondence between its interface and an implementation and none are intended as a base for derivation. Typically, concrete types are not fitted into a hierarchy of related classes. Each concrete type can be understood in isolation with minimal reference to other classes. If a concrete type is implemented well, programs using it are comparable in size and speed to programs a user would write using a hand-crafted and specialized version of the concept. Similarly, if the implementation changes significantly the interface is usually modified to reflect the change. In all of this, a concrete type resembles a built-in type. Naturally, the built-in types are all concrete. User-defined concrete types, such as complex numbers, matrices, error messages, and symbolic references, often provide fundamental types for some application domain.

The exact nature of a class' interface determines what implementation changes are significant in this context; more abstract interfaces leave more scope for implementation changes but can compromise run-time efficiency. Furthermore, a good implementation does not depend on other classes more than absolutely necessary so that the class can be used without compile-time or run-time overheads caused by the accommodation of other ''similar'' classes in a program.

To sum up, a class providing a concrete type aims:

[1] to be a close match to a particular concept and implementation strategy;

[2] to provide run-time and space efficiency comparable to ''hand-crafted'' code through the use of inlining and of operations taking full advantage of the properties of the concept and its implementation;

[3] to have only minimal dependency on other classes; and

[4] to be comprehensible and usable in isolation.

The result is a tight binding between user code and implementation code. If the implementation changes in any way, user code will have to be recompiled because user code almost always contains calls of inline functions or local variables of a concrete type.

The name ''concrete type'' was chosen to contrast with the common term ''abstract type.'' The relationship between concrete and abstract types is discussed in §25.3.

Concrete types cannot directly express commonality. For example, ***list*** and ***vector*** provide similar sets of operations and can be used interchangeably by some template functions. However, there is no relationship between the types ***list<int>*** and ***vector<int>*** or between ***list<Shape*>*** and ***list<Circle*>*** (§13.6.3), even though ***we*** can discern their similarities.

For naively designed concrete types, this implies that code using them in similar ways will look dissimilar. For example, iterating through a ***List*** using a ***next( )*** operation differs dramatically from iterating through a ***Vector*** using subscripting:

```
void my(List& sl)
{
    for (T* p = sl.first(); p; p = sl.next()) { // ''natural'' list iteration
        // my stuff
    }
    // ...
}

void your(Vector& v)
{
    for (int i = 0; i<v.size(); i++) {    // ''natural'' vector iteration
        // your stuff
    }
    // ...
}
```

The difference in iteration style is natural in the sense that a get-next-element operation is essential to the notion of a list (but not that common for a vector) and subscripting is essential to the notion of a vector (but not for a list). The availability of operations that are ''natural'' relative to a chosen implementation strategy is often crucial for efficiency and important for ease of writing the code.

The obvious snag is that the code for fundamentally similar operations, such as the previous two loops, can look dissimilar, and code that uses different concrete types for similar operations cannot be used interchangeably. In realistic examples, it takes significant thought to find similarities and significant redesign to provide ways of exploiting such similarities once found. The standard containers and algorithms are an example of a thorough rethinking that makes it possible to exploit similarities between concrete types without losing their efficiency and elegance benefits (§16.2).

To take a concrete type as an argument, a function must specify that exact concrete type as an argument type. There will be no inheritance relationships that can be used to make the argument declaration less specific. Consequently, an attempt to exploit similarities between concrete types will involve templates and generic programming as described in §3.8. When the standard library is used, iteration becomes:

```
template<class C> void ours(const C& c)
{
    for (C::const_iterator p = c.begin(); p!=c.end(); ++p) { // standard library iteration
        // ...
    }
}
```

The fundamental similarity between containers is exploited, and this in turn opens the possibility for further exploitation as done by the standard algorithms (Chapter 18).

To use a concrete type well, the user must understand its particular details. There are (typically) no general properties that hold for all concrete types in a library that can be relied on to save the user the bother of knowing the individual classes. This is the price of run-time compactness and efficiency. Sometimes that is a price well worth paying; sometimes it is not. It can also be the case that an individual concrete class is easier to understand and use than is a more general (abstract) class. This is often the case for classes that represent well-known data types such as arrays and lists.

Note, however, that the ideal is still to hide as much of the implementation as is feasible without seriously hurting performance. Inline functions can be a great win in this context. Exposing member variables by making them public or by providing set and get functions that allow the user to manipulate them directly is almost never a good idea (§24.4.2). Concrete types should still be types and not just bags of bits with a few functions added for convenience.

25.2.1 Reuse of Concrete Types [role.reuse]

Concrete types are rarely useful as bases for further derivation. Each concrete type aims at providing a clean and efficient representation of a single concept. A class that does that well is rarely a good candidate for the creation of different but related classes through public derivation. Such classes are more often useful as members or private base classes. There, they can be used effectively without having their interfaces and implementations mixed up with and compromised by those of the new classes. Consider deriving a new class from ***Date***:

```
class My_date : public Date {
    // ...
};
```

Is it ever valid for ***My_date*** to be used as a plain ***Date***? Well, that depends on what ***My_date*** is, but in my experience it is rare to find a concrete type that makes a good base class without modification.

A concrete type is ‘‘reused’’ unmodified in the same way as built-in types such as ***int*** are (§10.3.4). For example:

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    // ...
};
```

This form of use (reuse?) is usually simple, effective, and efficient.

Maybe it was a mistake not to design ***Date*** to be easy to modify through derivation? It is sometimes asserted that *every* class should be open to modification by overriding and by access from derived class member functions. This view leads to a variant of ***Date*** along these lines:

```
class Date2 {
public:
        // public interface, consisting primarily of virtual functions
protected:
        // other implementation details (possibly including some representation)
private:
        // representation and other implementation details
};
```

To make writing overriding functions easy and efficient, the representation is declared ***protected***. This achieves the objective of making ***Date2*** arbitrarily malleable by derivation, yet keeping its user interface unchanged. However, there are costs:

[1] *Less efficient basic operations*. A C++ virtual function call is a fraction slower than an ordinary function call, virtual functions cannot be inlined as often as non-virtual functions, and a class with virtual functions typically incurs a one-word space overhead.

[2] *The need to use free store*. The aim of ***Date2*** is to allow objects of different classes derived from ***Date2*** to be used interchangeably. Because the sizes of these derived classes differ, the obvious thing to do is to allocate them on the free store and access them through pointers or references. Thus, the use of genuine local variables dramatically decreases.

[3] *Inconvenience to users*. To benefit from the polymorphism provided by the virtual functions, accesses to ***Date2***s must be through pointers or references.

[4] *Weaker encapsulation*. The virtual operations can be overridden and protected data can be manipulated from derived classes (§12.4.1.1).

Naturally, these costs are not always significant, and the behavior of a class defined in this way is often exactly what we want (§25.3, §25.4). However, for a simple concrete type, such as ***Date2***, the costs are unnecessary and can be significant.

Finally, a well-designed concrete type is often the ideal representation for a more malleable type. For example:

```
class Date3 {
public:
        // public interface, consisting primarily of virtual functions
private:
        Date d;
};
```

This is the way to fit concrete types (including built-in types) into a class hierarchy when that is needed. See also §25.10[1].

25.3 Abstract Types [role.abstract]

The simplest way of loosening the coupling between users of a class and its implementers and also between code that creates objects and code that uses such objects is to introduce an abstract class that represents the interface to a set of implementations of a common concept. Consider a naive ***Set***:

```
template<class T> class Set {
public:
    virtual void insert(T*) = 0;
    virtual void remove(T*) = 0;

    virtual int is_member(T*) = 0;

    virtual T* first() = 0;
    virtual T* next() = 0;

    virtual ~Set() { }
};
```

This defines an interface to a set with a built-in notion of iteration over its elements. The absence of a constructor and the presence of a virtual destructor is typical (§12.4.2). Several implementations are possible (§16.2.1). For example:

```
template<class T> class List_set : public Set<T>, private list<T> {
    // ...
};

template<class T> class Vector_set : public Set<T>, private vector<T> {
    // ...
};
```

The abstract class provides the common interface to the implementations. This means we can use a ***Set*** without knowing which kind of implementation is used. For example:

```
void f(Set<Plane*>& s)
{
    for (Plane** p = s.first(); p; p = s.next()) {
        // my stuff
    }
    // ...
}

List_set<Plane*> sl;
Vector_set<Plane*> v(100);

void g()
{
    f(sl);
    f(v);
}
```

For concrete types, we required a redesign of the implementation classes to express commonality and used a template to exploit it. Here, we must design a common interface (in this case ***Set***), but no commonality beyond the ability to implement the interface is required of the classes used for implementation.

Furthermore, users of ***Set*** need not know the declarations of ***List_set*** and ***Vector_set***, so users need not depend on these declarations and need not be recompiled or in any way changed if ***List_set*** or ***Vector_set*** changes or even if a new implementation of ***Set*** – say ***Tree_set*** – is

introduced. All dependencies are contained in functions that explicitly use a class derived from ***Set***. In particular, assuming the conventional use of header files the programmer writing ***f***(***Set***&) needs only include ***Set.h*** and not ***List_set.h*** or ***Vector_set.h***. An ''implementation header'' is needed only where a ***List_set*** or a ***Vector_set***, respectively, is created. An implementation can be further insulated from the actual classes by introducing an abstract class that handles requests to create objects (''a factory;'' §12.4.4).

This separation of the interface from the implementations implies the absence of access to operations that are ''natural'' to a particular implementation but not general enough to be part of the interface. For example, because a ***Set*** doesn't have a notion of ordering we cannot support a subscripting operator in the ***Set*** interface even if we happen to be implementing a particular ***Set*** using an array. This implies a run-time cost due to missed hand optimizations. Furthermore, inlining typically becomes infeasible (except in a local context, when the compiler knows the real type), and all interesting operations on the interface become virtual function calls. As with concrete types, sometimes the cost of an abstract type is worth it; sometimes it is not. To sum up, an abstract type aims to:

[1] define a single concept in a way that allows several implementations of it to coexist in a program;
[2] provide reasonable run-time and space efficiency through the use of virtual functions;
[3] let each implementation have only minimal dependency on other classes; and
[4] be comprehensible in isolation.

Abstract types are not better than concrete types, just different. There are difficult and important tradeoffs for the user to make. The library provider can dodge the issue by providing both, thus leaving the choice to the user. The important thing is to be clear about to which world a class belongs. Limiting the generality of an abstract type in an attempt to compete in speed with a concrete type usually fails. It compromises the ability to use interchangeable implementations without significant recompilation after changes. Similarly, attempting to provide ''generality'' in concrete types to compete with the abstract type notion also usually fails. It compromises the efficiency and appropriateness of a simple class. The two notions can coexist – indeed, they *must* coexist because concrete classes provide the implementations for the abstract types – but they must not be muddled together.

Abstract types are often not intended to be bases for further derivation beyond their immediate implementation. Derivation is most often used just to supply implementation. However, a new interface can be constructed from an abstract class by deriving a more extensive abstract class from it. This new abstract class must then in turn be implemented through further derivation by a non-abstract class (§15.2.5).

Why didn't we derive ***List*** and ***Vector*** classes from ***Set*** in the first place to save the introduction of ***List_set*** and ***Vector_set*** classes? In other words, why have concrete types when we can have abstract types?

[1] *Efficiency*. We want to have concrete types such as ***vector*** and ***list*** without the overheads implied by decoupling the implementations from the interfaces (as implied by the abstract type style).
[2] *Reuse*. We need a mechanism to fit types designed ''elsewhere'' (such as ***vector*** and ***list***) into a new library or application by giving them a new interface (rather than rewriting them).

[3] *Multiple interfaces*. Using a single common base for a variety of classes leads to fat interfaces (§24.4.3). Often, it is better to provide a new interface to a class used for new purposes (such as a ***Set*** interface for a ***vector***) rather than modify its interface to serve multiple purposes.

Naturally, these points are related. They are discussed in some detail for the ***Ival_box*** example (§12.4.2, §15.2.5) and in the context of container design (§16.2). Using the ***Set*** base class would have resulted in a based-container solution relying on node classes (§25.4).

Section §25.7 describes a more flexible iterator in that the binding of the iterator to the implementation yielding the objects can be specified at the point of initialization and changed at run time.

25.4 Node Classes [role.node]

A class hierarchy is built with a view of derivation different from the interface/implementer view used for abstract types. Here, a class is viewed as a foundation on which to build. Even if it is an abstract class, it usually has some representation and provides some services for its derived classes. Examples of node classes are ***Polygon*** (§12.3), the initial ***Ival_slider*** (§12.4.1), and ***Satellite*** (§15.2).

Typically, a class in a hierarchy represents a general concept of which its derived classes can be seen as specializations. The typical class designed as an integral part of a hierarchy, a *node class*, relies on services from base classes to provide its own services. That is, it calls base class member functions. A typical node class provides not just an implementation of the interface specified by its base class (the way an implementation class does for an abstract type). It also adds new functions itself, thus providing a wider interface. Consider ***Car*** from the traffic-simulation example in §24.3.2:

```
class Car : public Vehicle {
public:
    Car(int passengers, Size_category size, int weight, int fc)
        : Vehicle(passengers,size,weight), fuel_capacity(fc) { /* ... */ }

    // override relevant virtual functions from Vehicle:

    void turn(Direction);
    // ...

    // add Car-specific functions:

    virtual add_fuel(int amount);   // a car needs fuel to run
    // ...
};
```

The important functions are the constructor through which the programmer specifies the basic properties that are relevant to the simulation and the (virtual) functions that allow the simulation routines to manipulate a ***Car*** without knowing its exact type. A ***Car*** might be created and used like this:

```
void user()
{
    // ...
    Car* p = new Car(3,economy,1500,60);
    drive(p,bs_home,MH);    // enter into simulated traffic pattern
    // ...
}
```

A node class usually needs constructors and often a nontrivial constructor. In this, node classes differ from abstract types, which rarely have constructors.

The operations on *Car* will typically use operations from the base class *Vehicle* in their implementations. In addition, the user of a *Car* relies on services from its base classes. For example, *Vehicle* provides the basic functions dealing with weight and size so that *Car* doesn't have to:

```
bool Bridge::can_cross(const Vehicle& r)
{
    if (max_weight<r.weight()) return false;
    // ...
}
```

This allows programmers to create new classes such as *Car* and *Truck* from a node class *Vehicle* by specifying and implementing only what needs to differ from *Vehicle*. This is often referred to as ''programming by difference'' or ''programming by extension.''

Like many node classes, a *Car* is itself a good candidate for further derivation. For example, an *Ambulance* needs additional data and operations to deal with emergencies:

```
class Ambulance : public Car, public Emergency {
public:
    Ambulance();

    // override relevant Car virtual functions:

    void turn(Direction);
    // ...

    // override relevant Emergency virtual functions:

    virtual dispatch_to(const Location&);
    // ...

    // add Ambulance-specific functions:

    virtual int patient_capacity();  // number of stretchers
    // ...
};
```

To sum up, a node class

[1] relies on its base classes both for its implementation and for supplying services to its users;
[2] provides a wider interface (that is, an interface with more public member functions) to its users than do its base classes;
[3] relies primarily (but not necessarily exclusively) on virtual functions in its public interface;
[4] depends on all of its (direct and indirect) base classes;

[5] can be understood only in the context of its base classes;
[6] can be used as a base for further derivation; and
[7] can be used to create objects.

Not every node class will conform to all of points 1, 2, 6, and 7, but most do. A class that does not conform to point 6 resembles a concrete type and could be called a *concrete node class*. For example, a concrete node class can be used to implement an abstract class (§12.4.2) and variables of such a class can be allocated statically and on the stack. Such a class is sometimes called a *leaf class*. However, any code operating on a pointer or reference to a class with virtual functions must take into account the possibility of a further derived class (or assume without language support that further derivation hasn't happened). A class that does not conform to point 7 resembles an abstract type and could be called an *abstract node class*. Because of unfortunate traditions, many node classes have at least some ***protected*** members to provide a less restricted interface for derived classes (§12.4.1.1).

Point 4 implies that to compile a node class, a programmer must include the declarations of all of its direct and indirect base classes and all of the declarations that they, in turn, depend on. In this, a node class again provides a contrast to an abstract type. A user of an abstract type does not depend on the classes used to implement it and need not include them to compile.

25.4.1 Changing Interfaces [role.io]

By definition, a node class is part of a class hierarchy. Not every class in a hierarchy needs to offer the same interface. In particular, a derived class can provide more member functions, and a sibling class can provide a completely different set of functions. From a design perspective, ***dynamic_cast*** (§15.4) can be seen as a mechanism for asking an object if it provides a given interface.

As an example, consider a simple object I/O system. Users want to read objects from a stream, determine that they are of the expected types, and then use them. For example:

```
void user()
{
        // ... open file assumed to hold shapes, and attach ss as an istream for that file ...

        Io_obj* p = get_obj(ss); // read object from stream

        if (Shape* sp = dynamic_cast<Shape*>(p)) {
                sp->draw();     // use the Shape
                // ...
        }
        else {
                // oops: non-shape in Shape file
        }
}
```

The function ***user()*** deals with shapes exclusively through the abstract class ***Shape*** and can therefore use every kind of shape. The use of ***dynamic_cast*** is essential because the object I/O system can deal with many other kinds of objects and the user may accidentally have opened a file containing perfectly good objects of classes that the user has never heard of.

This object I/O system assumes that every object read or written is of a class derived from ***Io_obj***. Class ***Io_obj*** must be a polymorphic type to allow us to use ***dynamic_cast***. For example:

```
class Io_obj {
public:
    virtual Io_obj* clone() const =0;      // polymorphic
    virtual ~Io_obj() {}
};
```

The critical function in the object I/O system is ***get_obj()***, which reads data from an ***istream*** and creates class objects based on that data. Assume that the data representing an object on an input stream is prefixed by a string identifying the object's class. The job of ***get_obj()*** is to read that string prefix and call a function capable of creating and initializing an object of the right class. For example:

```
typedef Io_obj* (*PF)(istream&);          // pointer to function returning an Io_obj*

map<string,PF> io_map;                    // maps strings to creation functions

bool get_word(istream& is, string& s);    // read a word from is into s

Io_obj* get_obj(istream& s)
{
    string str;
    bool b = get_word(s,str);             // read initial word into str
    if (b == false) throw No_class();     // io format problem

    PF f = io_map[str];                   // lookup 'str' to get function
    if (f == 0) throw Unknown_class();    // no match for 'str'

    return f(s);                          // construct object from stream
}
```

The ***map*** called ***io_map*** holds pairs of name strings and functions that can construct objects of the class with that name.

We could define class ***Shape*** in the usual way, except for deriving it from ***Io_obj*** as required by ***user()***:

```
class Shape : public Io_obj {
    // ...
};
```

However, it would be more interesting (and in many cases more realistic) to use a defined ***Shape*** (§2.6.2) unchanged:

```
class Io_circle : public Circle, public Io_obj {
public:
    Io_circle* clone() const { return new Io_circle(*this); }  // using copy constructor
    Io_circle(istream&);  // initialize from input stream
    static Io_obj* new_circle(istream& s) { return new Io_circle(s); }
    // ...
};
```

This is an example of how a class can be fitted into a hierarchy using an abstract class with less foresight than would have been required to build it as a node class in the first place (§12.4.2, §25.3).

The *Io_circle*(*istream*&) constructor initializes an object with data from its *istream* argument. The *new_circle*() function is the one put into the *io_map* to make the class known to the object I/O system. For example:

```
io_map["Io_circle"]=&Io_circle::new_circle;
```

Other shapes are constructed in the same way:

```
class Io_triangle : public Triangle, public Io_obj {
    // ...
};
```

If the provision of the object I/O scaffolding becomes tedious, a template might help:

```
template<class T> class Io : public T, public Io_obj {
public:
    Io* clone() const { return new Io(*this); }     // override Io_obj::clone()

    Io(istream&);                                   // initialize from input stream

    static Io* new_io(istream& s) { return new Io(s); }
    // ...
};
```

Given this, we can define *Io_circle*:

```
typedef Io<Circle> Io_circle;
```

We still need to define *Io*<*Circle*>::*Io*(*istream*&) explicitly, though, because it needs to know about the details of *Circle*.

The *Io* template is an example of a way to fit concrete types into a class hierarchy by providing a handle that is a node in that hierarchy. It derives from its template parameter to allow casting from *Io_obj*. Unfortunately, this precludes using *Io* for a built-in type:

```
typedef Io<Date> Io_date;       // wrap concrete type
typedef Io<int> Io_int;         // error: cannot derive from built-in type
```

This problem can be handled by providing a separate template for built-in types or by using a class representing a built-in type (§25.10[1]).

This simple object I/O system may not do everything anyone ever wanted, but it almost fits on a single page and the key mechanisms have many uses. In general, these techniques can be used to invoke a function based on a string supplied by a user and to manipulate objects of unknown type through interfaces discovered through run-time type identification.

25.5 Actions [role.action]

The simplest and most obvious way to specify an action in C++ is to write a function. However, if an action has to be delayed, has to be transmitted ‘‘elsewhere’’ before being performed, requires its own data, has to be combined with other actions (§25.10[18,19]), etc., then it often becomes attractive to provide the action in the form of a class that can execute the desired action and provide other services as well. A function object used with the standard algorithms is an obvious example

(§18.4), and so are the manipulators used with ***iostream***s (§21.4.6). In the former case, the actual action is performed by the application operator, and in the latter case, by the << or >> operators. In the case of ***Form*** (§21.4.6.3) and ***Matrix*** (§22.4.7), compositor classes were used to delay execution until sufficient information had been gathered for efficient execution.

A common form of action class is a simple class containing just one virtual function (typically called something like ''do_it''):

```
class Action {
public:
    virtual int do_it(int) = 0;
    virtual ~Action() { }
};
```

Given this, we can write code – say a menu – that can store actions for later execution without using pointers to functions, without knowing anything about the objects invoked, and without even knowing the name of the operation it invokes. For example:

```
class Write_file : public Action {
    File& f;
public:
    int do_it(int) { return f.write().succeed(); }
};

class Error_response : public Action {
    string message;
public:
    int do_it(int);
};

int Error_response::do_it(int)
{
    Response_box db(message.c_str(), "continue","cancel","retry");

    switch (db.get_response()) {
    case 0:
        return 0;
    case 1:
        abort();
    case 2:
        current_operation.redo();
        return 1;
    }
}

Action* actions[] = {
    new Write_file(f),
    new Error_response("you blew it again"),
    // ...
};
```

A user of ***Action*** can be completely insulated from any knowledge of derived classes such as ***Write_file*** and ***Error_response***.

This is a powerful technique that should be treated with some care by people with a background in functional decomposition. If too many classes start looking like ***Action***, the overall design of the system may have deteriorated into something unduly functional.

Finally, a class can encode an operation for execution on a remote machine or for storage for future use (§25.10[18]).

25.6 Interface Classes [role.interface]

One of the most important kinds of classes is the humble and mostly overlooked interface class. An interface class doesn't do much – if it did, it wouldn't be an interface class. It simply adjusts the appearance of some service to local needs. Because it is impossible in principle to serve all needs equally well all the time, interface classes are essential to allow sharing without forcing all users into a common straitjacket.

The purest form of an interface doesn't even cause any code to be generated. Consider the ***Vector*** specialization from §13.5:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector() : Base() {}
    Vector(int i) : Base(i) {}

    T*& operator[](int i) { return static_cast<T*&>(Base::operator[](i)); }

    // ...
};
```

This (partial) specialization turns the unsafe ***Vector<void*>*** into a much more useful family of type-safe vector classes. Inline functions are often essential for making interface classes affordable. In cases such as this, when an inline forwarding function does only type adjustment, there is no added overhead in time or space.

Naturally, an abstract base class representing an abstract type implemented by concrete types (§25.2) is a form of interface class, as are the handles from §25.7. However, here we will focus on classes that have no more specific function than adjusting an interface.

Consider the problem of merging two hierarchies using multiple inheritance. What can be done if there is a name clash, that is, two classes have used the same name for virtual functions performing completely different operations? For example, consider a Wild-West videogame in which user interactions are handled by a general window class:

```
class Window {
    // ...
    virtual void draw(); // display image
};

class Cowboy {
    // ...
    virtual void draw(); // pull gun from holster
};
```

```
class Cowboy_window : public Cowboy, public Window {
    // ...
};
```

A *Cowboy_window* represents the animation of a cowboy in the game and handles the user/player's interactions with the cowboy character. We would prefer to use multiple inheritance, rather than declaring either the *Window* or the *Cowboy* as members, because there will be many service functions defined for both *Window*s and *Cowboy*s. We would like to pass a *Cowboy_window* to such functions without special actions required by the programmer. However, this leads to a problem defining *Cowboy_window* versions of *Cowboy*::*draw*( ) and *Window*::*draw*( ).

There can be only one function defined in *Cowboy_window* called *draw*( ). Yet because service functions manipulate *Window*s and *Cowboy*s without knowledge of *Cowboy_window*s, *Cowboy_window* must override both *Cowboy*'s *draw*( ) and *Window*'s *draw*( ). Overriding both functions by a single *draw*( ) function would be wrong because, despite the common name, the *draw*( ) functions are unrelated and cannot be redefined by a common function.

Finally, we would also like *Cowboy_window* to have distinct, unambiguous names for the inherited functions *Cowboy*::*draw*( ) and *Window*::*draw*( ).

To solve this problem, we need to introduce an extra class for *Cowboy* and an extra class for *Window*. These classes introduce the two new names for the *draw*( ) functions and ensure that a call of the *draw*( ) functions in *Cowboy* and *Window* calls the functions with the new names:

```
class CCowboy : public Cowboy {          // interface to Cowboy renaming draw()
public:
    virtual int cow_draw( ) = 0;
    void draw( ) { cow_draw( ); }        // override Cowboy::draw()
};

class WWindow : public Window {          // interface to Window renaming draw()
public:
    virtual int win_draw( ) = 0;
    void draw( ) { win_draw( ); }        // override Window::draw()
};
```

We can now compose a *Cowboy_window* from the interface classes *CCowboy* and *WWindow* and override *cow_draw*( ) and *win_draw*( ) with the desired effect:

```
class Cowboy_window : public CCowboy, public WWindow {
    // ...
    void cow_draw( );
    void win_draw( );
};
```

Note that this problem was serious only because the two *draw*( ) functions have the same argument type. If they have different argument types, the usual overloading resolution rules will ensure that no problem manifests itself despite the unrelated functions having the same name.

For each use of an interface class, one could imagine a special-purpose language extension that could perform the desired adjustment a little bit more efficiently or a little more elegantly. However, each use of an interface class is infrequent and supporting them all with specialized language constructs would impose a prohibitive burden of complexity. In particular, name clashes arising from the merging of class hierarchies are not common (compared with how often a programmer will write a class) and tend to arise from the merging of hierarchies generated from dissimilar cultures – such as games and window systems. Merging such dissimilar hierarchies is not easy, and resolving name clashes will more often than not be the least of the programmer's problems. Other problems include dissimilar error handling, dissimilar initialization, and dissimilar memory-management strategies. The resolution of name clashes is discussed here because the technique of introducing an interface class with a forwarding function has many other applications. It can be used not only to change names, but also to change argument and return types, to introduce run-time checking, etc.

Because the forwarding functions *CCowboy::draw()* and *WWindow::draw()* are virtual functions, they cannot be optimized away by simple inlining. It is, however, possible for a compiler to recognize them as simple forwarding functions and then optimize them out of the call chains that go through them.

25.6.1 Adjusting Interfaces [role.range]

A major use of interface functions is to adjust an interface to match users' expectations better, thus moving code that would have been scattered throughout a user's code into an interface. For example, the standard *vector* is zero-based. Users who want ranges other than *0* to *size-1* must adjust their usage. For example:

```
void f()
{
    vector v<int>(10);          // range [0:9]

    // pretend v is in the range [1:10]:

    for (int i = 1; i<=10; i++) {
        v[i-1] = 7;     // remember to adjust index
        // ...
    }
}
```

A better way is to provide a *vector* with arbitrary bounds:

```
class Vector : public vector<int> {
    int lb;
public:
    Vector(int low, int high) : vector<int>(high-low+1) { lb=low; }

    int& operator[](int i) { return vector<int>::operator[](i-lb); }

    int low() { return lb; }
    int high() { return lb+size()-1; }
};
```

A ***Vector*** can be used like this:

```
void g()
{
    Vector v(1,10);          // range [1:10]

    for (int i = 1; i<=10; i++) {
        v[i] = 7;
        // ...
    }
}
```

This imposes no overhead compared to the previous example. Clearly, the ***Vector*** version is easier to read and write and is less error-prone.

Interface classes are usually rather small and (by definition) do rather little. However, they crop up wherever software written according to different traditions needs to cooperate because then there is a need to mediate between different conventions. For example, interface classes are often used to provide C++ interfaces to non-C++ code and to insulate application code from the details of libraries (to leave open the possibility of replacing the library with another).

Another important use of interface classes is to provide checked or restricted interfaces. For example, it is not uncommon to have integer variables that are supposed to have values in a given range only. This can be enforced (at run time) by a simple template:

```
template<int low, int high> class Range {
    int val;
public:
    class Error { }; // exception class

    Range(int i) { Assert<Error>(low<=i&&i<high); val = i; }// see §24.3.7.2
    Range operator=(int i) { return *this=Range(i); }

    operator int() { return val; }
    // ...
};

void f(Range<2,17>);
void g(Range<-10,10>);

void h(int x)
{
    Range<0,2001> i = x;     // might throw Range::Error
    int i1 = i;

    f(3);
    f(17);                   // throws Range::Error
    g(-7);
    g(100);                  // throws Range::Error
}
```

The ***Range*** template is easily extended to handle ranges of arbitrary scalar types (§25.10[7]).

An interface class that controls access to another class or adjusts its interface is sometimes called a ***wrapper***.

25.7 Handle Classes [role.handle]

An abstract type provides an effective separation between an interface and its implementations. However, as used in §25.3 the connection between an interface provided by an abstract type and its implementation provided by a concrete type is permanent. For example, it is not possible to rebind an abstract iterator from one source – say, a set – to another – say, a stream – once the original source becomes exhausted.

Furthermore, unless one manipulates an object implementing an abstract class through pointers or references, the benefits of virtual functions are lost. User code may become dependent on details of the implementation classes because an abstract type cannot be allocated statically or on the stack (including being accepted as a by-value argument) without its size being known. Using pointers and references implies that the burden of memory management falls on the user.

Another limitation of the abstract class approach is that a class object is of fixed size. Classes, however, are used to represent concepts that require varying amounts of storage to implement them.

A popular technique for dealing with these issues is to separate what is used as a single object into two parts: a handle providing the user interface and a representation holding all or most of the object's state. The connection between the handle and the representation is typically a pointer in the handle. Often, handles have a bit more data than the simple representation pointer, but not much more. This implies that the layout of a handle is typically stable even when the representation changes and also that handles are small enough to move around relatively freely so that pointers and references need not be used by the user.

The ***String*** from §11.12 is a simple example of a handle. The handle provides an interface to, access control for, and memory management for the representation. In this case, both the handle and the representation are concrete types, but the representation class is often an abstract class.

Consider the abstract type ***Set*** from §25.3. How could one provide a handle for it, and what benefits and cost would that involve? Given a set class, one might simply define a handle by overloading the `->` operator:

```
template<class T> class Set_handle {
    Set<T>* rep;
public:
    Set<T>* operator->() { return rep; }

    Set_handle(Set<T>* pp) : rep(pp) { }
};
```

This doesn't significantly affect the way ***Set***s are used; one simply passes ***Set_handle***s around instead of ***Set***&s or ***Set****s. For example:

```
void f(Set_handle<int> s)
{
    for (int* p = s->first(); p; p = s->next())
    {
        // ...
    }
}

void user()
{
    Set_handle<int> sl(new List_set<int>);
    Set_handle<int> v(new Vector_set<int>(100));

    f(sl);
    f(v);
}
```

Often, we want a handle to do more than just provide access. For example, if the ***Set*** class and the ***Set_handle*** class are designed together it is easy to do reference counting by including a use count in each ***Set***. In general, we do not want to design a handle together with what it is a handle to, so we will have to store any information that needs to be shared by a handle in a separate object. In other words, we would like to have non-intrusive handles in addition to the intrusive ones. For example, here is a handle that removes an object when its last handle goes away:

```
template<class X> class Handle {
    X* rep;
    int* pcount;
public:
    X* operator->() { return rep; }

    Handle(X* pp) : rep(pp), pcount(new int(1)) { }
    Handle(const Handle& r) : rep(r.rep), pcount(r.pcount) { (*pcount)++; }

    Handle& operator=(const Handle& r)
    {
        if (rep == r.rep) return *this;
        if (--(*pcount) == 0) {
            delete rep;
            delete pcount;
        }
        rep = r.rep;
        pcount = r.pcount;
        (*pcount)++;
        return *this;
    }

    ~Handle() { if (--(*pcount) == 0) { delete rep; delete pcount; } }

    // ...
};
```

Such a handle can be passed around freely. For example:

```
void f1(Handle<Set>);

Handle<Set> f2()
{
    Handle<Set> h(new List_set<int>);
    // ...
    return h;
}

void g()
{
    Handle<Set> hh = f2();
    f1(hh);
    // ...
}
```

Here, the set created in *f2*( ) will be deleted upon exit from *g*( ) – unless *f1*( ) held on to a copy; the programmer does not need to know.

Naturally, this convenience comes at a cost, but for many applications the cost of storing and maintaining the use count is acceptable.

Sometimes, it is useful to extract the representation pointer from a handle and use it directly. For example, this would be needed to pass an object to a function that does not know about handles. This works nicely provided the called function does not destroy the object passed to it or store a pointer to it for use after returning to its caller. An operation for rebinding a handle to a new representation can also be useful:

```
template<class X> class Handle {
    // ...

    X* get_rep() { return rep; }

    void bind(X* pp)
    {
        if (pp != rep) {
            if (--*pcount == 0) {
                delete rep;
                *pcount = 1;                // recycle pcount
            }
            else
                pcount = new int(1);        // new pcount
            rep = pp;
        }
    }
};
```

Note that derivation of new classes from ***Handle*** isn't particularly useful. It is a concrete type without virtual functions. The idea is to have one handle class for a family of classes defined by a base class. Derivation from this base class can be a powerful technique. It applies to node classes as well as to abstract types.

As written, ***Handle*** doesn't deal with inheritance. To get a class that acts like a genuine use-counted pointer, ***Handle*** needs to be combined with ***Ptr*** from §13.6.3.1 (see §25.10[2]).

A handle that provides an interface that is close to identical to the class for which it is a handle is often called a *proxy*. This is particularly common for handles that refer to an object on a remote machine.

25.7.1 Operations in Handles [role.handle.op]

Overloading `->` enables a handle to gain control and do some work on each access to an object. For example, one could collect statistics about the number of uses of the object accessed through a handle:

```
template <class T> class Xhandle {
    T* rep;
    int no_of_accesses;
public:
    T* operator->() { no_of_accesses++; return rep; }

    // ...
};
```

Handles for which work needs to be done both before *and* after access require more elaborate programming. For example, one might want a set with locking while an insertion or a removal is being done. Essentially, the representation class' interface needs to be replicated in the handle class:

```
template<class T> class Set_controller {
    Set<T>* rep;
    Lock lock;
    // ...
public:
    void insert(T* p) { Lock_ptr x(lock); rep->insert(p); }  // see §14.4.1
    void remove(T* p) { Lock_ptr x(lock); rep->remove(p); }

    int is_member(T* p) { return rep->is_member(p); }

    T get_first() { T* p = rep->first(); return p ? *p : T(); }
    T get_next() { T* p = rep->next(); return p ? *p : T(); }

    T first() { Lock_ptr x(lock); T tmp = *rep->first(); return tmp; }
    T next() { Lock_ptr x(lock); T tmp = *rep->next(); return tmp; }

    // ...
};
```

Providing these forwarding functions is tedious (and therefore somewhat error-prone), although it is neither difficult nor costly in run time.

Note that only some of the ***set*** functions required locking. In my experience, it is typical that a class needing pre- and post-actions requires them for only some member functions. In the case of locking, locking on all operations – as is done for monitors in some systems – leads to excess locking and sometimes causes a noticeable decrease in concurrency.

An advantage of the elaborate definition of all operations on the handle over the overloading of `->` style of handles is that it is possible to derive from class ***Set_controller***. Unfortunately, some of the benefits of being a handle are compromised if data members are added in the derived class. In particular, the amount of code shared (in the handled class) decreases compared to the amount of code written in each handle.

25.8 Application Frameworks [role.framework]

Components built out of the kinds of classes described in §25.2–§25.7 support design and reuse of code by supplying building blocks and ways of combining them; the application builder designs a framework into which these common building blocks are fitted. An alternative, and sometimes more ambitious, approach to the support of design and reuse is to provide code that establishes a common framework into which the application builder fits application-specific code as building blocks. Such an approach is often called an *application framework*. The classes establishing such a framework often have such fat interfaces that they are hardly types in the traditional sense. They approximate the ideal of being complete applications, except that they don't do anything. The specific actions are supplied by the application programmer.

As an example, consider a filter, that is, a program that reads an input stream, (maybe) performs some actions based on that input, (maybe) produces an output stream, and (maybe) produces a final result. A naive framework for such programs would provide a set of operations that an application programmer might supply:

```
class Filter {
public:
    class Retry {
    public:
        virtual const char* message() { return 0; }
    };

    virtual void start() { }
    virtual int read() = 0;
    virtual void write() { }
    virtual void compute() { }
    virtual int result() = 0;

    virtual int retry(Retry& m) { cerr << m.message() << '\n'; return 2; }

    virtual ~Filter() { }
};
```

Functions that a derived class must supply are declared pure virtual; other functions are simply defined to do nothing.

The framework also provides a main loop and a rudimentary error-handling mechanism:

```
int main_loop(Filter* p)
{
    for(;;) {
        try {
            p->start();
            while (p->read()) {
                p->compute();
                p->write();
            }
            return p->result();
        }
        catch (Filter::Retry& m) {
            if (int i = p->retry(m)) return i;
        }
        catch (...) {
            cerr << "Fatal filter error\n";
            return 1;
        }
    }
}
```

Finally, I could write my program like this:

```
class My_filter : public Filter {
    istream& is;
    ostream& os;
    int nchar;
public:
    int read() { char c; is.get(c); return is.good(); }
    void compute() { nchar++; }
    int result() { os << nchar << " characters read\n"; return 0; }

    My_filter(istream& ii, ostream& oo) : is(ii), os(oo), nchar(0) { }
};
```

and activate it like this:

```
int main()
{
    My_filter f(cin,cout);
    return main_loop(&f);
}
```

Naturally, for a framework to be of significant use, it must provide more structure and many more services than this simple example does. In particular, a framework is typically a hierarchy of node classes. Having the application programmer supply leaf classes in a deeply nested hierarchy allows commonality between applications and reuse of services provided by such a hierarchy. A framework will also be supported by a library that provides classes that are useful for the application programmer when specifying the action classes.

25.9 Advice [role.advice]

[1] Make conscious decisions about how a class is to be used (both as a designer and as a user); §25.1.
[2] Be aware of the tradeoffs involved among the different kinds of classes; §25.1.
[3] Use concrete types to represent simple independent concepts; §25.2.
[4] Use concrete types to represent concepts where close-to-optimal efficiency is essential; §25.2.
[5] Don't derive from a concrete class; §25.2.
[6] Use abstract classes to represent interfaces where the representation of objects might change; §25.3.
[7] Use abstract classes to represent interfaces where different representations of objects need to coexist; §25.3.
[8] Use abstract classes to represent new interfaces to existing types; §25.3.
[9] Use node classes where similar concepts share significant implementation details; §25.4.
[10] Use node classes to incrementally augment an implementation; §25.4.
[11] Use Run-time Type Identification to obtain interfaces from an object; §25.4.1.
[12] Use classes to represent actions with associated state; §25.5.
[13] Use classes to represent actions that need to be stored, transmitted, or delayed; §25.5.
[14] Use interface classes to adapt a class for a new kind of use (without modifying the class); §25.6.
[15] Use interface classes to add checking; §25.6.1.
[16] Use handles to avoid direct use of pointers and references; §25.7.
[17] Use handles to manage shared representations; §25.7.
[18] Use an application framework where an application domain allows for the control structure to be predefined; §25.8.

25.10 Exercises [role.exercises]

1. (∗1) The ***Io*** template from §25.4.1 does not work for built-in types. Modify it so that it does.
2. (∗1.5) The ***Handle*** template from §25.7 does not reflect inheritance relationships of the classes for which it is a handle. Modify it so that it does. That is, you should be able to assign a ***Handle<Circle*>*** to a ***Handle<Shape*>*** but not the other way around.
3. (∗2.5) Given a ***String*** class, define another string class using it as the representation and providing its operations as virtual functions. Compare the performance of the two classes. Try to find a meaningful class that is best implemented by publicly deriving from the string with virtual functions.
4. (∗4) Study two widely used libraries. Classify the library classes in terms of concrete types, abstract types, node classes, handle classes, and interface classes. Are abstract node classes and concrete node classes used? Is there a more appropriate classification for the classes in these libraries? Are fat interfaces used? What facilities – if any – are provided for run-time type information? What is the memory-management strategy?
5. (∗2) Use the ***Filter*** framework (§25.8) to implement a program that removes adjacent duplicate words from an input stream but otherwise copies the input to output.
6. (∗2) Use the ***Filter*** framework to implement a program that counts the frequency of words on

an input stream and produces a list of (word,count) pairs in frequency order as output.

7. (∗1.5) Write a ***Range*** template that takes both the range and the element type as template parameters.
8. (∗1) Write a ***Range*** template that takes the range as constructor arguments.
9. (∗2) Write a simple string class that performs no error checking. Write another class that checks access to the first. Discuss the pros and cons of separating basic function and checking for errors.
10. (∗2.5) Implement the object I/O system from §25.4.1 for a few types, including at least integers, strings, and a class hierarchy of your choice.
11. (∗2.5) Define a class ***Storable*** as an abstract base class with virtual functions ***write_out***( ) and ***read_in***( ). For simplicity, assume that a character string is sufficient to specify a permanent storage location. Use class ***Storable*** to provide a facility for writing objects of classes derived from ***Storable*** to disk, and for reading such objects from disk. Test it with a couple of classes of your own choice.
12. (∗4) Define a base class ***Persistent*** with operations ***save***( ) and ***no_save***( ) that control whether an object is written to permanent storage by a destructor. In addition to ***save***( ) and ***no_save***( ), what operations could ***Persistent*** usefully provide? Test class ***Persistent*** with a couple of classes of your own choice. Is ***Persistent*** a node class, a concrete type, or an abstract type? Why?
13. (∗3) Write a class ***Stack*** for which it is possible to change implementation at run time. Hint: ‘‘Every problem is solved by yet another indirection.’’
14. (∗3.5) Define a class ***Oper*** that holds an identifier of type ***Id*** (maybe a ***string*** or a C-style string) and an operation (a pointer to function or some function object). Define a class ***Cat_object*** that holds a list of ***Oper***s and a ***void****. Provide ***Cat_object*** with operations ***add_oper***(***Oper***), which adds an ***Oper*** to the list; ***remove_oper***(***Id***), which removes an ***Oper*** identified by ***Id*** from the list; and an ***operator***( )(***Id***, ***arg***), which invokes the ***Oper*** identified by ***Id***. Implement a stack of ***Cat***s by a ***Cat_object***. Write a small program to exercise these classes.
15. (∗3) Define a template ***Object*** based on class ***Cat_object***. Use ***Object*** to implement a stack of ***String***s. Write a small program to exercise this template.
16. (∗2.5) Define a variant of class ***Object*** called ***Class*** that ensures that objects with identical operations share a list of operations. Write a small program to exercise this template.
17. (∗2) Define a ***Stack*** template that provides a conventional and type-safe interface to a stack implemented by the ***Object*** template. Compare this stack to the stack classes found in the previous exercises. Write a small program to exercise this template.
18. (∗3) Write a class for representing operations to be shipped to another computer to execute there. Test it either by actually sending commands to another machine or by writing commands to a file and then executing the commands read from the file.
19. (∗2) Write a class for composing operations represented as function objects. Given two function objects f and g, ***Compose***(f, g) should make an object that can be invoked with an argument x suitable for g and return $f(g(x))$, provided the return value of g( ) is an acceptable argument type for f( ).

Appendices and Index

These Appendices provide the C++ grammar, a discussion of compatibility issues that arise between C++ and C and between Standard C++ and prestandard versions of C++, and a variety of language-technical details. The index is extensive and considered an integral part of the book.

Chapters

A Grammar
B Compatibility
C Technicalities
I Index

Appendix A

Grammar

There is no worse danger for a teacher
than to teach words instead of things.
– Marc Block

Introduction — keywords — lexical conventions — programs — expressions — statements — declarations — declarators — classes — derived classes — special member functions — overloading — templates — exception handling — preprocessing directives.

A.1 Introduction

This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (§A.5, §A.7) must be applied to distinguish expressions from declarations. Moreover, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

The C and C++ standard grammars express very minor distinctions syntactically rather than through constraints. That gives precision, but it doesn't always improve readability.

A.2 Keywords

New context-dependent keywords are introduced into a program by ***typedef*** (§4.9.7), namespace (§8.2), class (Chapter 10), enumeration (§4.8), and ***template*** (Chapter 13) declarations.

typedef-name:
    identifier

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    template-id

enum-name:
    identifier

template-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name*.

Unless an identifier is explicitly declared to name a type, it is assumed to name something that is not a type (see §C.13.5).

The C++ keywords are:

C++ Keywords					
and	*and_eq*	*asm*	*auto*	*bitand*	*bitor*
bool	*break*	*case*	*catch*	*char*	*class*
compl	*const*	*const_cast*	*continue*	*default*	*delete*
do	*double*	*dynamic_cast*	*else*	*enum*	*explicit*
export	*extern*	*false*	*float*	*for*	*friend*
goto	*if*	*inline*	*int*	*long*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*register*	*reinterpret_cast*
return	*short*	*signed*	*sizeof*	*static*	*static_cast*
struct	*switch*	*template*	*this*	*throw*	*true*
try	*typedef*	*typeid*	*typename*	*union*	*unsigned*
using	*virtual*	*void*	*volatile*	*wchar_t*	*while*
xor	*xor_eq*				

A.3 Lexical Conventions

The standard C and C++ grammars present lexical conventions as grammar productions. This adds precision but also makes for large grammars and doesn't always increase readability:

hex-quad:
    hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
`\u` *hex-quad*
`\U` *hex-quad hex-quad*

preprocessing-token:
header-name
identifier
pp-number
character-literal
string-literal
preprocessing-op-or-punc
each non-white-space character that cannot be one of the above

token:
identifier
keyword
literal
operator
punctuator

header-name:
`<`*h-char-sequence*`>`
`"`*q-char-sequence*`"`

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any member of the source character set except new-line and `>`

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any member of the source character set except new-line and `"`

pp-number:
digit
`.` *digit*
pp-number digit
pp-number nondigit
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `.`

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of
universal-character-name
`_ a b c d e f g h i j k l m n o p q r s t u v w x y z`
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

digit: one of
`0 1 2 3 4 5 6 7 8 9`

preprocessing-op-or-punc: one of

```
{     }     [     ]     #     ##    (     )     <:    :>    <%    %>    %:%:
%:    ;     :     ?     ::    .     .*    +     -     *     /     %     ^
&     |     ~     !     =     <     >     +=    -=    *=    /=    %=    ^=
&=    |=    <<=   >>=   <<    >>    ==    !=    <=    >=    &&    ||    ++
--    ,     ->    ->*   ...   new   delete      and   and_eq       bitand
bitor       compl       not   or    not_eq      xor   or_eq       xor_eq
```

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal

integer-literal:
decimal-literal integer-suffix$_{opt}$
octal-literal integer-suffix$_{opt}$
hexadecimal-literal integer-suffix$_{opt}$

decimal-literal:
nonzero-digit
decimal-literal digit

octal-literal:
`0`
octal-literal octal-digit

hexadecimal-literal:
`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
  `u` `U`

long-suffix: one of
  `l` `L`

character-literal:
  `'` *c-char-sequence* `'`
  `L'` *c-char-sequence* `'`

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except the single-quote, backslash, or new-line character
  escape-sequence
  universal-character-name

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  `\'` `\"` `\?` `\\` `\a` `\b` `\f` `\n` `\r` `\t` `\v`

octal-escape-sequence:
  `\` *octal-digit*
  `\` *octal-digit octal-digit*
  `\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
  `\x` *hexadecimal-digit*
  hexadecimal-escape-sequence hexadecimal-digit

floating-literal:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$ `.` *digit-sequence*
  digit-sequence `.`

exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*

sign: one of
  `+` `-`

digit-sequence:
  digit
  digit-sequence digit

floating-suffix: one of
  `f l F L`

string-literal:
  `"`*s-char-sequence*$_{opt}$`"`
  `L"`*s-char-sequence*$_{opt}$`"`

s-char-sequence:
  s-char
  s-char-sequence s-char

s-char:
  any member of the source character set except double-quote, backslash , or new-line
  escape-sequence
  universal-character-name

boolean-literal:
  `false`
  `true`

A.4 Programs

A program is a collection of *translation-unit*s combined through linking (§9.4). A *translation-unit*, often called a *source file*, is a sequence of *declaration*s:

translation-unit:
  declaration-seq$_{opt}$

A.5 Expressions

See §6.2.

primary-expression:
  literal
  `this`
  `::` *identifier*
  `::` *operator-function-id*
  `::` *qualified-id*
  `(` *expression* `)`
  id-expression

id-expression:
  unqualified-id
  qualified-id

id-expression:
  unqualified-id
  qualified-id

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  `~` *class-name*
  template-id

qualified-id:
  nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
  class-or-namespace-name `::` *nested-name-specifier*$_{opt}$
  class-or-namespace-name `::` `template` *nested-name-specifier*

class-or-namespace-name:
  class-name
  namespace-name

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *expression-list*$_{opt}$ `)`
  simple-type-specifier `(` *expression-list*$_{opt}$ `)`
  `typename` `::`$_{opt}$ *nested-name-specifier identifier* `(` *expression-list*$_{opt}$ `)`
  `typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id* `(` *expression-list*$_{opt}$ `)`
  postfix-expression `.` `template`$_{opt}$ `::`$_{opt}$ *id-expression*
  postfix-expression `->` `template`$_{opt}$ `::`$_{opt}$ *id-expression*
  postfix-expression `.` *pseudo-destructor-name*
  postfix-expression `->` *pseudo-destructor-name*
  postfix-expression `++`
  postfix-expression `--`
  `dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
  `static_cast` `<` *type-id* `>` `(` *expression* `)`
  `reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
  `const_cast` `<` *type-id* `>` `(` *expression* `)`
  `typeid` `(` *expression* `)`
  `typeid` `(` *type-id* `)`

expression-list:
  assignment-expression
  expression-list `,` *assignment-expression*

pseudo-destructor-name:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `::` `~` *type-name*
  `::`$_{opt}$ *nested-name-specifier* `template` *template-id* `::` `~` *type-name*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `~` *type-name*

unary-expression:
- *postfix-expression*
- `++` *cast-expression*
- `--` *cast-expression*
- *unary-operator* *cast-expression*
- `sizeof` *unary-expression*
- `sizeof` `(` *type-id* `)`
- *new-expression*
- *delete-expression*

unary-operator: one of
- `*` `&` `+` `-` `!` `~`

new-expression:
- `::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
- `::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
- `(` *expression-list* `)`

new-type-id:
- *type-specifier-seq new-declarator*$_{opt}$

new-declarator:
- *ptr-operator new-declarator*$_{opt}$
- *direct-new-declarator*

direct-new-declarator:
- `[` *expression* `]`
- *direct-new-declarator* `[` *constant-expression* `]`

new-initializer:
- `(` *expression-list*$_{opt}$ `)`

delete-expression:
- `::`$_{opt}$ `delete` *cast-expression*
- `::`$_{opt}$ `delete [ ]` *cast-expression*

cast-expression:
- *unary-expression*
- `(` *type-id* `)` *cast-expression*

pm-expression:
- *cast-expression*
- *pm-expression* `.*` *cast-expression*
- *pm-expression* `->*` *cast-expression*

multiplicative-expression:
- *pm-expression*
- *multiplicative-expression* `*` *pm-expression*
- *multiplicative-expression* `/` *pm-expression*
- *multiplicative-expression* `%` *pm-expression*

additive-expression:
  multiplicative-expression
  additive-expression `+` *multiplicative-expression*
  additive-expression `-` *multiplicative-expression*

shift-expression:
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

relational-expression:
  shift-expression
  relational-expression `<` *shift-expression*
  relational-expression `>` *shift-expression*
  relational-expression `<=` *shift-expression*
  relational-expression `>=` *shift-expression*

equality-expression:
  relational-expression
  equality-expression `==` *relational-expression*
  equality-expression `!=` *relational-expression*

and-expression:
  equality-expression
  and-expression `&` *equality-expression*

exclusive-or-expression:
  and-expression
  exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression `|` *exclusive-or-expression*

logical-and-expression:
  inclusive-or-expression
  logical-and-expression `&&` *inclusive-or-expression*

logical-or-expression:
  logical-and-expression
  logical-or-expression `||` *logical-and-expression*

conditional-expression:
  logical-or-expression
  logical-or-expression `?` *expression* `:` *assignment-expression*

assignment-expression:
  conditional-expression
  logical-or-expression *assignment-operator* *assignment-expression*
  throw-expression

assignment-operator: one of
  `=` `*=` `/=` `%=` `+=` `-=` `>>=` `<<=` `&=` `^=` `|=`

expression:
    assignment-expression
    expression , *assignment-expression*

constant-expression:
    conditional-expression

Grammar ambiguities arise from the similarity between function style casts and declarations. For example:

```
int x;

void f()
{
    char(x);   // conversion of x to char or declaration of a char called x?
}
```

All such ambiguities are resolved to declarations. That is, ''if it could possibly be interpreted as a declaration, it is a declaration.'' For example:

```
T(a)->m;          // expression statement
T(a)++;           // expression statement

T(*e)(int(3));    // declaration
T(f)[4];          // declaration

T(a);             // declaration
T(a)=m;           // declaration
T(*b)();          // declaration
T(x),y,z=7;       // declaration
```

This disambiguation is purely syntactic. The only information used for a name is whether it is known to be a name of a type or a name of a template. If that cannot be determined, the name is assumed to name something that isn't a template or a type.

The construct ***template*** *unqualified-id* is used to state that the *unqualified-id* is the name of a template in a context in which that cannot be deduced (see §C.13.5).

A.6 Statements

See §6.3.

statement:
    labeled-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
    declaration-statement
    try-block

labeled-statement:
  identifier `:` *statement*
  `case` *constant-expression* `:` *statement*
  `default` `:` *statement*

expression-statement:
  expression$_{opt}$ `;`

compound-statement:
  `{` *statement-seq*$_{opt}$ `}`

statement-seq:
  statement
  statement-seq statement

selection-statement:
  `if` `(` *condition* `)` *statement*
  `if` `(` *condition* `)` *statement* `else` *statement*
  `switch` `(` *condition* `)` *statement*

condition:
  expression
  type-specifier-seq declarator `=` *assignment-expression*

iteration-statement:
  `while` `(` *condition* `)` *statement*
  `do` *statement* `while` `(` *expression* `)` `;`
  `for` `(` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

for-init-statement:
  expression-statement
  simple-declaration

jump-statement:
  `break` `;`
  `continue` `;`
  `return` *expression*$_{opt}$ `;`
  `goto` *identifier* `;`

declaration-statement:
  block-declaration

A.7 Declarations

The structure of declarations is described in Chapter 4, enumerations in §4.8, pointers and arrays in Chapter 5, functions in Chapter 7, namespaces in §8.2, linkage directives in §9.2.4, and storage classes in §10.4.

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
- *block-declaration*
- *function-definition*
- *template-declaration*
- *explicit-instantiation*
- *explicit-specialization*
- *linkage-specification*
- *namespace-definition*

block-declaration:
- *simple-declaration*
- *asm-definition*
- *namespace-alias-definition*
- *using-declaration*
- *using-directive*

simple-declaration:
- *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

decl-specifier:
- *storage-class-specifier*
- *type-specifier*
- *function-specifier*
- `friend`
- `typedef`

decl-specifier-seq:
- *decl-specifier-seq*$_{opt}$ *decl-specifier*

storage-class-specifier:
- `auto`
- `register`
- `static`
- `extern`
- `mutable`

function-specifier:
- `inline`
- `virtual`
- `explicit`

typedef-name:
- *identifier*

type-specifier:
- *simple-type-specifier*
- *class-specifier*
- *enum-specifier*
- *elaborated-type-specifier*
- *cv-qualifier*

simple-type-specifier:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
  `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*
  char
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void

type-name:
  class-name
  enum-name
  typedef-name

elaborated-type-specifier:
  class-key `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
  `enum` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
  `typename` `::`$_{opt}$ *nested-name-specifier identifier*
  `typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*

enum-name:
  identifier

enum-specifier:
  `enum` *identifier*$_{opt}$ `{` *enumerator-list*$_{opt}$ `}`

enumerator-list:
  enumerator-definition
  enumerator-list `,` *enumerator-definition*

enumerator-definition:
  enumerator
  enumerator `=` *constant-expression*

enumerator:
  identifier

namespace-name:
  original-namespace-name
  namespace-alias

original-namespace-name:
  identifier

namespace-definition:
  named-namespace-definition
  unnamed-namespace-definition

named-namespace-definition:
  original-namespace-definition
  extension-namespace-definition

original-namespace-definition:
  `namespace` *identifier* `{` *namespace-body* `}`

extension-namespace-definition:
  `namespace` *original-namespace-name* `{` *namespace-body* `}`

unnamed-namespace-definition:
  `namespace {` *namespace-body* `}`

namespace-body:
  declaration-seq$_{opt}$

namespace-alias:
  identifier

namespace-alias-definition:
  `namespace` *identifier* `=` *qualified-namespace-specifier* `;`

qualified-namespace-specifier:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name*

using-declaration:
  `using typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
  `using ::` *unqualified-id* `;`

using-directive:
  `using namespace ::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name* `;`

asm-definition:
  `asm (` *string-literal* `) ;`

linkage-specification:
  `extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
  `extern` *string-literal* *declaration*

The grammar allows for arbitrary nesting of declarations. However, some semantic restrictions apply. For example, nested functions (functions defined local to other functions) are not allowed.

The list of specifiers that starts a declaration cannot be empty (there is no ‘‘implicit ***int***;’’ §B.2) and consists of the longest possible sequence of specifiers. For example:

```
typedef int I;
void f(unsigned I) { /* ... */ }
```

Here, ***f*( )** takes an unnamed ***unsigned int***.

An ***asm*( )** is an assembly code insert. Its meaning is implementation-defined, but the intent is for the string to be a piece of assembly code that will be inserted into the generated code at the place where it is specified.

Declaring a valiable ***register*** is a hint to the compiler to optimize for frequent access; doing so is redundant with most modern compilers.

A.7.1 Declarators

See §4.9.1, Chapter 5 (pointers and arrays), §7.7 (pointers to functions), and §15.5 (pointers to members).

init-declarator-list:
  init-declarator
  init-declarator-list `,` *init-declarator*

init-declarator:
  declarator *initializer*$_{opt}$

declarator:
  direct-declarator
  ptr-operator declarator

direct-declarator:
  declarator-id
  direct-declarator `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
  direct-declarator `[` *constant-expression*$_{opt}$ `]`
  `(` *declarator* `)`

ptr-operator:
  `*` *cv-qualifier-seq*$_{opt}$
  `&`
  `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
  cv-qualifier *cv-qualifier-seq*$_{opt}$

cv-qualifier:
  `const`
  `volatile`

declarator-id:
  `::`$_{opt}$ *id-expression*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*

type-id:
  type-specifier-seq *abstract-declarator*$_{opt}$

type-specifier-seq:
  type-specifier type-specifier-seq$_{opt}$

abstract-declarator:
  ptr-operator abstract-declarator$_{opt}$
  direct-abstract-declarator

direct-abstract-declarator:
  direct-abstract-declarator$_{opt}$ `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*
  direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`
  `(` *abstract-declarator* `)`

parameter-declaration-clause:
  parameter-declaration-list$_{opt}$ . . .$_{opt}$
  parameter-declaration-list , . . .

parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , *parameter-declaration*

parameter-declaration:
  decl-specifier-seq declarator
  decl-specifier-seq declarator = *assignment-expression*
  decl-specifier-seq abstract-declarator$_{opt}$
  decl-specifier-seq abstract-declarator$_{opt}$ = *assignment-expression*

function-definition:
  decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*
  decl-specifier-seq$_{opt}$ *declarator function-try-block*

function-body:
  compound-statement

initializer:
  = *initializer-clause*
  (*expression-list*)

initializer-clause:
  assignment-expression
  { *initializer-list* ,$_{opt}$ }
  { }

initializer-list:
  initializer-clause
  initializer-list , *initializer-clause*

A ***volatile*** specifier is a hint to a compiler that an object may change its value in ways not specified by the language so that aggressive optimizations must be avoided. For example, a real time clock might be declared:

```
extern const volatile clock;
```

Two successive reads of ***clock*** might give different results.

A.8 Classes

See Chapter 10.

class-name:
  identifier
  template-id

class-specifier:
  class-head { *member-specification*$_{opt}$ }

class-head:
  class-key identifier$_{opt}$ *base-clause*$_{opt}$
  class-key nested-name-specifier identifier base-clause$_{opt}$
  class-key nested-name-specifier `template` *template-id base-clause*$_{opt}$

class-key:
  `class`
  `struct`
  `union`

member-specification:
  member-declaration member-specification$_{opt}$
  access-specifier `:` *member-specification*$_{opt}$

member-declaration:
  decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ `;`
  function-definition `;`$_{opt}$
  `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id* `;`
  using-declaration
  template-declaration

member-declarator-list:
  member-declarator
  member-declarator-list `,` *member-declarator*

member-declarator:
  declarator pure-specifier$_{opt}$
  declarator constant-initializer$_{opt}$
  identifier$_{opt}$ `:` *constant-expression*

pure-specifier:
  `= 0`

constant-initializer:
  `=` *constant-expression*

To preserve C compatibility, a class and a non-class of the same name can be declared in the same scope (§5.7). For example:

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

In this case, the plain name (***stat***) is the name of the non-class. The class must be referred to using a *class-key* prefix .

Constant expressions are defined in §C.5.

A.8.1 Derived Classes

See Chapter 12 and Chapter 15.

base-clause:
  `:` *base-specifier-list*

base-specifier-list:
  base-specifier
  base-specifier-list `,` *base-specifier*

base-specifier:
  $::_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
  virtual access-specifier$_{opt}$ $::_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
  access-specifier virtual$_{opt}$ $::_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

access-specifier:
  `private`
  `protected`
  `public`

A.8.2 Special Member Functions

See §11.4 (conversion operators), §10.4.6 (class member initialization), and §12.2.2 (base initialization).

conversion-function-id:
  `operator` *conversion-type-id*

conversion-type-id:
  type-specifier-seq conversion-declarator$_{opt}$

conversion-declarator:
  ptr-operator conversion-declarator$_{opt}$

ctor-initializer:
  `:` *mem-initializer-list*

mem-initializer-list:
  mem-initializer
  mem-initializer `,` *mem-initializer-list*

mem-initializer:
  mem-initializer-id `(` *expression-list*$_{opt}$ `)`

mem-initializer-id:
  $\texttt{::}_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
  identifier

A.8.3 Overloading

See Chapter 11.

operator-function-id:
  `operator` *operator*

operator: one of

```
new  delete   new[]    delete[]
+    -    *    /    %    ^    &    |    ~    !    =    <    >
+=   -=   *=   /=   %=   ^=   &=   |=   <<   >>   >>=  <<=  ==
!=   <=   >=   &&   ||   ++   --   ,    ->*  ->   ()   []
```

A.9 Templates

Templates are explained in Chapter 13 and §C.13.

template-declaration:
  `export`$_{opt}$ `template <` *template-parameter-list* `>` *declaration*

template-parameter-list:
  template-parameter
  template-parameter-list `,` *template-parameter*

template-parameter:
  type-parameter
  parameter-declaration

type-parameter:
  `class` *identifier*$_{opt}$
  `class` *identifier*$_{opt}$ `=` *type-id*
  `typename` *identifier*$_{opt}$
  `typename` *identifier*$_{opt}$ `=` *type-id*
  `template <` *template-parameter-list* `> class` *identifier*$_{opt}$
  `template <` *template-parameter-list* `> class` *identifier*$_{opt}$ `=` *template-name*

template-id:
  template-name `<` *template-argument-list*$_{opt}$ `>`

template-name:
  identifier

template-argument-list:
  template-argument
  template-argument-list `,` *template-argument*

template-argument:
  assignment-expression
  type-id
  template-name

explicit-instantiation:
  template declaration

explicit-specialization:
  `template < >` *declaration*

The explicit template argument specification opens up the possibility of an obscure syntactic ambiguity. Consider:

```
void h()
{
    f<1>(0);  // ambiguity: ((f)<1) > (0) or (f<1>)(0) ?
              // resolution: f<1> is called with argument 0
}
```

The resolution is simple and effective: if *f* is a template name, *f*< is the beginning of a qualified template name and the subsequent tokens must be interpreted based on that; otherwise, < means less-than. Similarly, the first non-nested > terminates a template argument list. If a greater-than is needed, parentheses must be used:

```
f< a>b >(0);        // syntax error
f< (a>b) >(0);      // ok
```

A similar lexical ambiguity can occur when terminating >s get too close. For example:

```
list<vector<int>> lv1;      // syntax error: unexpected >> (right shift)
list< vector<int> > lv2;    // correct: list of vectors
```

Note the space between the two >s; >> is the right-shift operator. That can be a real nuisance.

A.10 Exception Handling

See §8.3 and Chapter 14.

try-block:
    `try` *compound-statement handler-seq*

function-try-block:
    `try` *ctor-initializer*$_{opt}$ *function-body handler-seq*

handler-seq:
    handler handler-seq$_{opt}$

handler:
    `catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
    type-specifier-seq declarator
    type-specifier-seq abstract-declarator
    type-specifier-seq
    `...`

throw-expression:
    `throw` *assignment-expression*$_{opt}$

exception-specification:
    `throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
    type-id
    type-id-list `,` *type-id*

A.11 Preprocessing Directives

The preprocessor is a relatively unsophisticated macro processor that works primarily on lexical tokens rather than individual characters. In addition to the ability to define and use macros (§7.8), the preprocessor provides mechanisms for including text files and standard headers (§9.2.1) and conditional compilation based on macros (§9.3.3). For example:

```
#if OPT==4
#include "header4.h"
#elif 0<OPT
#include "someheader.h"
#else
#include<cstdlib>
#endif
```

All preprocessor directives start with a #, which must be the first non-whitespace character on its line.

preprocessing-file:
  group$_{opt}$

group:
  group-part
  group group-part

group-part:
  pp-tokens$_{opt}$ *new-line*
  if-section
  control-line

if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
  # `if` *constant-expression new-line group*$_{opt}$
  # `ifdef` *identifier new-line group*$_{opt}$
  # `ifndef` *identifier new-line group*$_{opt}$

elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  # `elif` *constant-expression new-line group*$_{opt}$

else-group:
  # `else` *new-line group*$_{opt}$

endif-line:
  # `endif` *new-line*

control-line:
`#` `include` *pp-tokens new-line*
`#` `define` *identifier replacement-list new-line*
`#` `define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*
`#` `undef` *identifier new-line*
`#` `line` *pp-tokens new-line*
`#` `error` *pp-tokens*$_{opt}$ *new-line*
`#` `pragma` *pp-tokens*$_{opt}$ *new-line*
`#` *new-line*

lparen:
the left-parenthesis character without preceding white-space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

Appendix B

Compatibility

You go ahead and follow your customs,
and I´ll follow mine.
– C. Napier

C/C++ compatibility — silent differences between C and C++ — C code that is not C++ — deprecated features — C++ code that is not C — coping with older C++ implementations — headers — the standard library — namespaces — allocation errors — templates — *for-statement* initializers — advice — exercises.

B.1 Introduction

This appendix discusses the incompatibilities between C and C++ and between Standard C++ (as defined by ISO/IEC 14882) and earlier versions of C++. The purpose is to document differences that can cause problems for the programmer and point to ways of dealing with such problems. Most compatibility problems surface when people try to upgrade a C program to a C++ program, try to port a C++ program from one pre-standard version of C++ to another, or try to compile C++ using modern features with an older compiler. The aim here is not to drown you in the details of every compatibility problem that ever surfaced in an implementation, but rather to list the most frequently occurring problems and present their standard solutions.

When you look at compatibility issues, a key question to consider is the range of implementations under which a program needs to work. For learning C++, it makes sense to use the most complete and helpful implementation. For delivering a product, a more conservative strategy might be in order to maximize the number of systems on which the product can run. In the past, this has been a reason (and sometimes just an excuse) to avoid C++ features deemed novel. However, implementations are converging, so the need for portability across platforms is less cause for extreme caution than it was a couple of years ago.

B.2 C/C++ Compatibility

With minor exceptions, C++ is a superset of C (meaning C89, defined by ISO/IEC 9899:1990). Most differences stem from C++'s greater emphasis on type checking. Well-written C programs tend to be C++ programs as well. A compiler can diagnose every difference between C++ and C.

B.2.1 ''Silent'' Differences

With a few exceptions, programs that are both C++ and C have the same meaning in both languages. Fortunately, these ''silent differences'' are rather obscure:

In C, the size of a character constant and of an enumeration equals *sizeof*(*int*). In C++, *sizeof*(´*a*´) equals *sizeof*(*char*), and a C++ implementation is allowed to choose whatever size is most appropriate for an enumeration (§4.8).

C++ provides the // comments; C does not (although many C implementations provide them as an extension). This difference can be used to construct programs that behave differently in the two languages. For example:

```
int f(int a, int b)
{
    return a //* pretty unlikely */ b
        ;    /* unrealistic: semicolon on separate line to avoid syntax error */
}
```

C99 (meaning C as defined by ISO/IEC 9899:1999(E)), also provides //.

A structure name declared in an inner scope can hide the name of an object, function, enumerator, or type in an outer scope. For example:

```
int x[99];
void f()
{
    struct x { int a; };
    sizeof(x); /* size of the array in C, size of the struct in C++ */
}
```

B.2.2 C Code That Is Not C++

The C/C++ incompatibilities that cause most real problems are not subtle. Most are easily caught by compilers. This section gives examples of C code that is not C++. Most are deemed poor style or even obsolete in modern C.

In C, most functions can be called without a previous declaration. For example:

```
main()    /* poor style C. Not C++ */
{
    double sq2 = sqrt(2);                          /* call undeclared function */
    printf("the square root of 2 is %g\n",sq2);    /* call undeclared function */
}
```

Complete and consistent use of function declarations (function prototypes) is generally recommended for C. Where that sensible advice is followed, and especially where C compilers provide

options to enforce it, C code conforms to the C++ rule. Where undeclared functions are called, you have to know the functions and the rules for C pretty well to know whether you have made a mistake or introduced a portability problem. For example, the previous *main*( ) contains at least two errors as a C program.

In C, a function declared without specifying any argument types can take any number of arguments of any type at all. Such use is deemed obsolescent in Standard C, but it is not uncommon:

```
void f(); /* argument types not mentioned */

void g()
{
    f(2);       /* poor style C. Not C++ */
}
```

In C, functions can be defined using a syntax that optionally specifies argument types after the list of arguments:

```
void f(a,p,c) char *p; char c; { /* ... */ }    /* C. Not C++ */
```

Such definitions must be rewritten:

```
void f(int a, char* p, char c) { /* ... */ }
```

In C and in pre-standard versions of C++, the type specifier defaults to ***int***. For example:

```
const a = 7;    /* In C, type int assumed. Not C++ */
```

C99 disallows ''implicit ***int***,'' just as in C++.

C allows the definition of ***struct***s in return type and argument type declarations. For example:

```
struct S { int x,y; } f();              /* C. Not C++ */
void g(struct S { int x,y; } y);        /* C. Not C++ */
```

The C++ rules for defining types make such declarations useless, and they are not allowed.

In C, integers can be assigned to variables of enumeration type:

```
enum Direction { up, down };
enum Direction d = 1;                   /* error: int assigned to Direction; ok in C */
```

C++ provides many more keywords than C does. If one of these appears as an identifier in a C program, that program must be modified to make it a C++ program:

C++ Keywords That Are Not C Keywords					
and	*and_eq*	*asm*	*bitand*	*bitor*	*bool*
catch	*class*	*compl*	*const_cast*	*delete*	*dynamic_cast*
explicit	*export*	*false*	*friend*	*inline*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*reinterpret_cast*	*static_cast*
template	*this*	*throw*	*true*	*try*	*typeid*
typename	*using*	*virtual*	*wchar_t*	*xor*	*xor_eq*

In C, some of the C++ keywords are macros defined in standard headers:

C++ Keywords That Are C Macros							
and	*and_eq*	*bitand*	*bitor*	*bool*	*compl*	*false*	
not	*not_eq*	*or*	*or_eq*	*true*	*wchar_t*	*xor*	*xor_eq*

This implies that in C they can be tested using ***#ifdef***, redefined, etc.

In C, a global data object may be declared several times in a single translation unit without using the ***extern*** specifier. As long as at most one such declaration provides an initializer, the object is considered defined only once. For example:

```
int i; int i;      /* defines or declares a single integer 'i'; not C++ */
```

In C++, an entity must be defined exactly once; §9.2.3.

In C++, a class may not have the same name as a ***typedef*** declared to refer to a different type in the same scope; §5.7.

In C, a ***void**** may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not (§5.6). For example:

```
void f(int n)
{
        int* p = malloc(n*sizeof(int)); /* not C++. In C++, allocate using 'new' */
}
```

C allows transfer of control to a *labeled-statement* (§A.6) to bypass an initialization; C++ does not.

In C, a global ***const*** by default has external linkage; in C++ it does not and must be initialized, unless explicitly declared ***extern*** (§5.4).

In C, names of nested structures are placed in the same scope as the structure in which they are nested. For example:

```
struct S {
        struct T { /* ... */ };
        // ...
};

struct T x;      /* ok in C meaning 'S::T x;'. Not C++ */
```

In C, an array can be initialized by an initializer that has more elements than the array requires. For example:

```
char v[5] = "Oscar";      /* ok in C, the terminating 0 is not used. Not C++ */
```

B.2.3 Deprecated Features

By deprecating a feature, the standards committee expresses the wish that the feature would go away. However, the committee does not have a mandate to remove a heavily used feature – however redundant or dangerous it may be. Thus, a deprecation is a strong hint to the users to avoid the feature.

The keyword ***static***, which usually means ''statically allocated,'' can be used to indicate that a function or an object is local to a translation unit. For example:

```
// file1:
    static int glob;

// file2:
    static int glob;
```

This program genuinely has two integers called ***glob***. Each ***glob*** is used exclusively by functions defined in its translation unit.

The use of ***static*** to indicate ''local to translation unit'' is deprecated in C++. Use unnamed namespaces instead (§8.2.5.1).

The implicit conversion of a string literal to a (non-***const***) ***char***** is deprecated. Use named arrays of ***char*** or avoid assignment of string literals to ***char*****s (§5.2.2).

C-style casts should have been deprecated when the new-style casts were introduced. Programmers should seriously consider banning C-style casts from their own programs. Where explicit type conversion is necessary, ***static_cast***, ***reinterpret_cast***, ***const_cast***, or a combination of these can do what a C-style cast can. The new-style casts should be preferred because they are more explicit and more visible (§6.2.7).

B.2.4 C++ Code That Is Not C

This section lists facilities offered by C++ but not by C. The features are sorted by purpose. However, many classifications are possible and most features serve multiple purposes, so this classification should not be taken too seriously.

- Features primarily for notational convenience:
  - [1] // comments (§2.3); added to C99
  - [2] Support for restricted character sets (§C.3.1); partially added to C99
  - [3] Support for extended character sets (§C.3.3); added to C99
  - [4] Non-constant initializers for objects in ***static*** storage (§9.4.1)
  - [5] ***const*** in constant expressions (§5.4, §C.5)
  - [6] Declarations as statements (§6.3.1); added to C99
  - [7] Declarations in *for-statement* initializers (§6.3.3); added to C99
  - [8] Declarations in conditions (§6.3.2.1)
  - [9] Structure names need not be prefixed by ***struct*** (§5.7)
- Features primarily for strengthening the type system:
  - [1] Function argument type checking (§7.1); later added to C (§B.2.2)
  - [2] Type-safe linkage (§9.2, §9.2.3)
  - [3] Free store management using ***new*** and ***delete*** (§6.2.6, §10.4.5, §15.6)
  - [4] ***const*** (§5.4, §5.4.1); later added to C
  - [5] The Boolean type ***bool*** (§4.2); partially added to C99
  - [6] New cast syntax (§6.2.7)
- Facilities for user-defined types:
  - [1] Classes (Chapter 10)
  - [2] Member functions (§10.2.1) and member classes (§11.12)
  - [3] Constructors and destructors (§10.2.3, §10.4.1)
  - [4] Derived classes (Chapter 12, Chapter 15)

[5] ***virtual*** functions and abstract classes (§12.2.6, §12.3)
[6] Public/protected/private access control (§10.2.2, §15.3, §C.11)
[7] ***friend***s (§11.5)
[8] Pointers to members (§15.5, §C.12)
[9] ***static*** members (§10.2.4)
[10] ***mutable*** members (§10.2.7.2)
[11] Operator overloading (Chapter 11)
[12] References (§5.5)

– Features primarily for program organization (in addition to classes):
[1] Templates (Chapter 13, §C.13)
[2] Inline functions (§7.1.1); added to C99
[3] Default arguments (§7.5)
[4] Function overloading (§7.4)
[5] Namespaces (§8.2)
[6] Explicit scope qualification (operator `::`; §4.9.4)
[7] Exception handling (§8.3, Chapter 14)
[8] Run-time Type Identification (§15.4)

The keywords added by C++ (§B.2.2) can be used to spot most C++-specific facilities. However, some facilities, such as function overloading and ***const***s in constant expressions, are not identified by a keyword. In addition to the features listed, the C++ library (§16.1.2) is mostly C++ specific.

The ***__cplusplus*** macro can be used to determine whether a program is being processed by a C or a C++ compiler (§9.2.4).

B.3 Coping with Older C++ Implementations

C++ has been in constant use since 1983 (§1.4). Since then, several versions have been defined and many separately developed implementations have emerged. The fundamental aim of the standards effort was to ensure that implementers and users would have a single definition of C++ to work from. Until that definition becomes pervasive in the C++ community, however, we have to deal with the fact that not every implementation provides every feature described in this book.

It is unfortunately not uncommon for people to take their first serious look at C++ using a five-year-old implementation. The typical reason is that such implementations are widely available and free. Given a choice, no self-respecting professional would touch such an antique. For a novice, older implementations come with serious hidden costs. The lack of language features and library support means that the novice must struggle with problems that have been eliminated in newer implementations. Using a feature-poor older implementation also warps the novice's programming style and gives a biased view of what C++ is. The best subset of C++ to initially learn is *not* the set of low-level facilities (and not the common C and C++ subset; §1.2). In particular, I recommend relying on the standard library and on templates to ease learning and to get a good initial impression of what C++ programming can be.

The first commercial release of C++ was in late 1985. The language was defined by the first edition of this book. At that point, C++ did not offer multiple inheritance, templates, run-time type information, exceptions, or namespaces. Today, I see no reason to use an implementation that

doesn't provide at least some of these features. I added multiple inheritance, templates, and exceptions to the definition of C++ in 1989. However, early support for templates and exceptions was uneven and often poor. If you find problems with templates or exceptions in an older implementation, consider an immediate upgrade.

In general, it is wise to use an implementation that conforms to the standard wherever possible and to minimize the reliance on implementation-defined and undefined aspects of the language. Design as if the full language were available and then use whatever workarounds are needed. This leads to better organized and more maintainable programs than designing for the lowest-common-denominator subset of C++. Also, be careful to use implementation-specific language extensions only when absolutely necessary.

B.3.1 Headers

Traditionally, every header file had a *.h* suffix. Thus, C++ implementations provided headers such as *<map.h>* and *<iostream.h>*. For compatibility, most still do.

When the standards committee needed headers for redefined versions of standard libraries and for newly added library facilities, naming those headers became a problem. Using the old *.h* names would have caused compatibility problems. The solution was to drop the *.h* suffix in standard header names. The suffix is redundant anyway because the < > notation indicates that a standard header is being named.

Thus, the standard library provides non-suffixed headers, such as *<iostream>* and *<map>*. The declarations in those files are placed in namespace *std*. Older headers place their declarations in the global namespace and use a *.h* suffix. Consider:

```
#include<iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

If this fails to compile on an implementation, try the more traditional version:

```
#include<iostream.h>

int main()
{
    cout << "Hello, world!\n";
}
```

Some of the most serious portability problems occur because of incompatible headers. The standard headers are only a minor contributor to this. Often, a program depends on a large number of headers that are not present on all systems, on a large number of declarations that don't appear in the same headers on all systems, and on declarations that appear to be standard (because they are found in headers with standard names) but are not part of any standard.

There are no fully-satisfactory approaches to dealing with portability in the face of inconsistent headers. A general idea is to avoid direct dependencies on inconsistent headers and localize the remaining dependencies. That is, we try to achieve portability through indirection and localization.

For example, if declarations that we need are provided in different headers in different systems, we may choose to ***#include*** an application specific header that in turn ***#include***s the appropriate header(s) for each system. Similarly, if some functionality is provided in slightly different forms on different systems, we may choose to access that functionality through application-specific interface classes and functions.

B.3.2 The Standard Library

Naturally, pre-standard-C++ implementations may lack parts of the standard library. Most will have iostreams, non-templated ***complex***, a different ***string*** class, and the C standard library. However, some may lack ***map***, ***list***, ***valarray***, etc. In such cases, use the – typically proprietary – libraries available in a way that will allow conversion when your implementation gets upgraded to the standard. It is usually better to use a non-standard ***string***, ***list***, and ***map*** than to revert to C-style programming in the absence of these standard library classes. Also, good implementations of the STL part of the standard library (Chapter 16, Chapter 17, Chapter 18, Chapter 19) are available free for downloading.

Early implementations of the standard library were incomplete. For example, some had containers that didn't support allocators and others required allocators to be explicitly specified for each class. Similar problems occurred for other ''policy arguments,'' such as comparison criteria. For example:

```
list<int> li;                          // ok, but some implementations require an allocator
list<int, allocator<int> > li2;        // ok, but some implementations don't implement allocators

map<string, Record> m1;                // ok, but some implementations require a less-operation
map<string, Record, less<string> > m2;
```

Use whichever version an implementation accepts. Eventually, the implementations will accept all.

Early C++ implementations provided ***istrstream*** and ***ostrstream*** defined in ***<strstream.h>*** instead of ***istringstream*** and ***ostringstream*** defined in ***<sstream>***. The ***strstream***s operated directly on a ***char[]*** (see §21.10[26]).

The streams in pre-standard-C++ implementations were not parameterized. In particular, the templates with the ***basic_*** prefix are new in the standard, and the ***basic_ios*** class used to be called ***ios***. Curiously enough, ***iostate*** used to be called ***io_state***.

B.3.3 Namespaces

If your implementation does not support namespaces, use source files to express the logical structure of the program (Chapter 9). Similarly, use header files to express interfaces that you provide for implementations or that are shared with C.

In the absence of namespaces, use ***static*** to compensate for the lack of unnamed namespaces. Also use an identifying prefix to global names to distinguish your names from those of other parts of the code. For example:

```
// for use on pre-namespace implementations:

class bs_string { /* ... */ };      // Bjarne's string
typedef int bs_bool;                // Bjarne's Boolean type
```

```
class joe_string;                    // Joe's string
enum joe_bool { joe_false, joe_true }; // Joe's bool
```

Be careful when choosing a prefix. Existing C and C++ libraries are littered with such prefixes.

B.3.4 Allocation Errors

In pre-exception-handling-C++, operator ***new*** returned ***0*** to indicate allocation failure. Standard C++'s ***new*** throws ***bad_alloc*** by default.

In general, it is best to convert to the standard. In this case, this means modify the code to catch ***bad_alloc*** rather than test for ***0***. In either case, coping with memory exhaustion beyond giving an error message is hard on many systems.

However, when converting from testing ***0*** to catching ***bad_alloc*** is impractical, you can sometimes modify the program to revert to the pre-exception-handling behavior. If no ***_new_handler*** is installed, using the ***nothrow*** allocator will cause a ***0*** to be returned in case of allocation failure:

```
X* p1 = new X;              // throws bad_alloc if no memory
X* p2 = new(nothrow) X;    // returns 0 if no memory
```

B.3.5 Templates

The standard introduced new template features and clarified the rules for several existing ones.

If your implementation doesn't support partial specialization, use a separate name for the template that would otherwise have been a specialization. For example:

```
template<class T> class plist : private list<void*> {  // should have been list<T*>
        // ...
};
```

If your implementation doesn't support member templates, some techniques become infeasible. In particular, member templates allow the programmer to specify construction and conversion with a flexibility that cannot be matched without them (§13.6.2). Sometimes, providing a nonmember function that constructs an object is an alternative. Consider:

```
template<class T> class X {
        // ...
        template<class A> X(const A& a);
};
```

In the absence of member templates, we must restrict ourselves to specific types:

```
template<class T> class X {
        // ...
        X(const A1& a);
        X(const A2& a);
        // ...
};
```

Most early implementations generated definitions for all member functions defined within a template class when that template class was instantiated. This could lead to errors in unused member

functions (§C.13.9.1). The solution is to place the definition of the member functions after the class declaration. For example, rather than

```
template<class T> class Container {
    // ...
public:
    void sort() { /* use < */ }        // in-class definition
};

class Glob { /* no < for Glob */ };

Container<Glob> cg; // some pre-standard implementations try to define Container<Glob>::sort()
```

use

```
template<class T> class Container {
    // ...
public:
    void sort();
};

template<class T> void Container<T>::sort() { /* use < */ }    // out-of-class definition

class Glob { /* no < for Glob */ };

Container<Glob> cg; // no problem as long as cg.sort() isn't called
```

Early implementations of C++ did not handle the use of members defined later in a class. For example:

```
template<class T> class Vector {
public:
    T& operator[](size_t i) { return v[i]; }   // v declared below
    // ...
private:
    T* v;                                      // oops: not found!
    size_t sz;
};
```

In such cases, either sort the member declarations to avoid the problem or place the definition of the member function after the class declaration.

Some pre-standard-C++ implementations do not accept default arguments for templates (§13.4.1). In that case, every template parameter must be given an explicit argument. For example:

```
template<class Key, class T, class LT = less<T> > class map {
    // ...
};

map<string, int> m;                    // Oops: default template arguments not implemented
map< string, int, less<string> > m2;   // workaround: be explicit
```

B.3.6 For-Statement Initializers

Consider:

```
void f(vector<char>& v, int m)
{
    for (int i= 0; i<v.size() && i<=m; ++i) cout << v[i];

    if (i == m) {    // error: i referred to after end of for-statement
        // ...
    }
}
```

Such code used to work because in the original definition of C++, the scope of the controlled variable extended to the end of the scope in which the *for-statement* appears. If you find such code, simply declare the controlled variable before the *for-statement*:

```
void f2(vector<char>& v, int m)
{
    int i= 0;   // i needed after the loop
    for (; i<v.size() && i<=m; ++i) cout << v[i];

    if (i == m) {
        // ...
    }
}
```

B.4 Advice

[1] For learning C++, use the most up-to-date and complete implementation of Standard C++ that you can get access to; §B.3.

[2] The common subset of C and C++ is not the best initial subset of C++ to learn; §1.6, §B.3.

[3] For production code, remember that not every C++ implementation is completely up-to-date. Before using a major new feature in production code, try it out by writing small programs to test the standards conformance and performance of the implementations you plan to use; for example, see §8.5[6-7], §16.5[10], §B.5[7].

[4] Avoid deprecated features such as global ***static***s; also avoid C-style casts; §6.2.7, §B.2.3.

[5] ''implicit ***int***'' has been banned, so explicitly specify the type of every function, variable, ***const***, etc.; §B.2.2.

[6] When converting a C program to C++, first make sure that function declarations (prototypes) and standard headers are used consistently; §B.2.2.

[7] When converting a C program to C++, rename variables that are C++ keywords; §B.2.2.

[8] When converting a C program to C++, cast the result of ***malloc***( ) to the proper type or change all uses of ***malloc***( ) to uses of ***new***; §B.2.2.

[9] When converting from ***malloc***( ) and ***free***( ) to ***new*** and ***delete***, consider using ***vector***, ***push_back***( ), and ***reserve***( ) instead of ***realloc***( ); §3.8, §16.3.5.

[10] When converting a C program to C++, remember that there are no implicit conversions from ***int***s to enumerations; use explicit type conversion where necessary; §4.8.

[11] A facility defined in namespace *std* is defined in a header without a suffix (e.g. *std*::*cout* is declared in <*iostream*>). Older implementations have standard library facilities in the global namespace and declared in headers with a *.h* suffix (e.g. ::*cout* declared in <*iostream.h*>); §9.2.2, §B.3.1.

[12] If older code tests the result of *new* against *0*, it must be modified to catch *bad_alloc* or to use *new*(*nothrow*); §B.3.4.

[13] If your implementation doesn't support default template arguments, provide arguments explicitly; *typedef*s can often be used to avoid repetition of template arguments (similar to the way the typedef *string* saves you from saying *basic_string*< *char*, *char_traits*<*char*>, *allocator*<*char*> >); §B.3.5.

[14] Use <*string*> to get *std*::*string* (<*string.h*> holds the C-style string functions); §9.2.2, §B.3.1.

[15] For each standard C header <*X.h*> that places names in the global namespace, the header <*cX*> places the names in namespace *std*; §B.3.1.

[16] Many systems have a "*String.h*" header defining a string type. Note that such strings differ from the standard library *string*.

[17] Prefer standard facilities to non-standard ones; §20.1, §B.3, §C.2.

[18] Use *extern* "*C*" when declaring C functions; §9.2.4.

B.5 Exercises

1. (∗2.5) Take a C program and convert it to a C++ program; list the kinds of non-C++ constructs used and determine if they are valid ANSI C constructs. First convert the program to strict ANSI C (adding prototypes, etc.), then to C++. Estimate the time it would take to convert a 100,000 line C program to C++.
2. (∗2.5) Write a program to help convert C programs to C++ by renaming variables that are C++ keywords, replacing calls of *malloc*( ) by uses of *new*, etc. Hint: don't try to do a perfect job.
3. (∗2) Replace all uses of *malloc*( ) in a C-style C++ program (maybe a recently converted C program) to uses of *new*. Hint: §B.4[8-9].
4. (∗2.5) Minimize the use of macros, global variables, uninitialized variables, and casts in a C-style C++ program (maybe a recently converted C program).
5. (∗3) Take a C++ program that is the result of a crude conversion from C and critique it as a C++ program considering locality of information, abstraction, readability, extensibility, and potential for reuse of parts. Make one significant change to the program based on that critique.
6. (∗2) Take a small (say, 500 line) C++ program and convert it to C. Compare the original with the result for size and probable maintainability.
7. (∗3) Write a small set of test programs to determine whether a C++ implementation has ''the latest'' standard features. For example, what is the scope of a variable defined in a *for-statement* initializer? (§B.3.6), are default template arguments supported? (§B.3.5), are member templates supported? (§13.6.2), and is argument-based lookup supported? (§8.2.6). Hint: §B.2.4.
8. (∗2.5) Take a C++ program that use <*X.h*> headers and convert it to using <*X*> and <*cX*> headers. Minimize the use of *using-directive*s.

Appendix C

Technicalities

Deep in the fundamental
heart of mind and Universe,
there is a reason.
– Slartibartfast

What the standard promises — character sets — integer literals — constant expressions — promotions and conversions — multidimensional arrays — fields and unions — memory management — garbage collection — namespaces — access control — pointers to data members — templates — ***static*** members — ***friends*** — templates as template parameters — template argument deduction — ***typename*** and ***template*** qualification — instantiation — name binding — templates and namespaces — explicit instantiation — advice.

C.1 Introduction and Overview

This chapter presents technical details and examples that do not fit neatly into my presentation of the main C++ language features and their uses. The details presented here can be important when you are writing a program and essential when reading code written using them. However, I consider them technical details that should not be allowed to distract from the student's primary task of learning to use C++ well or the programmer's primary task of expressing ideas as clearly and as directly as possible in C++.

C.2 The Standard

Contrary to common belief, strictly adhering to the C++ language and library standard doesn't guarantee good code or even portable code. The standard doesn't say whether a piece of code is good

or bad; it simply says what a programmer can and cannot rely on from an implementation. One can write perfectly awful standard-conforming programs, and most real-world programs rely on features not covered by the standard.

Many important things are deemed *implementation-defined* by the standard. This means that each implementation must provide a specific, well-defined behavior for a construct and that behavior must be documented. For example:

```
unsigned char c1 = 64;        // well-defined: a char has at least 8 bits and can always hold 64
unsigned char c2 = 1256;      // implementation-defined: truncation if a char has only 8 bits
```

The initialization of ***c1*** is well-defined because a ***char*** must be at least 8 bits. However, the behavior of the initialization of ***c2*** is implementation-defined because the number of bits in a ***char*** is implementation-defined. If the ***char*** has only 8 bits, the value *1256* will be truncated to *232* (§C.6.2.1). Most implementation-defined features relate to differences in the hardware used to run a program.

When writing real-world programs, it is usually necessary to rely on implementation-defined behavior. Such behavior is the price we pay for the ability to operate effectively on a large range of systems. For example, the language would have been much simpler if all characters had been 8 bits and all integers 32 bits. However, 16-bit and 32-bit character sets are not uncommon – nor are integers too large to fit in 32 bits. For example, many computers now have disks that hold more that *32G* bytes, so 48-bit or 64-bit integers can be useful for representing disk addresses.

To maximize portability, it is wise to be explicit about what implementation-defined features we rely on and to isolate the more subtle examples in clearly marked sections of a program. A typical example of this practice is to present all dependencies on hardware sizes in the form of constants and type definitions in some header file. To support such techniques, the standard library provides ***numeric_limits*** (§22.2).

Undefined behavior is nastier. A construct is deemed *undefined* by the standard if no reasonable behavior is required by an implementation. Typically, some obvious implementation technique will cause a program using an undefined feature to behave very badly. For example:

```
const int size = 4*1024;
char page[size];

void f()
{
    page[size+size] = 7; // undefined
}
```

Plausible outcomes of this code fragment include overwriting unrelated data and triggering a hardware error/exception. An implementation is not required to choose among plausible outcomes. Where powerful optimizers are used, the actual effects of undefined behavior can become quite unpredictable. If a set of plausible and easily implementable alternatives exist, a feature is deemed implementation-defined rather than undefined.

It is worth spending considerable time and effort to ensure that a program does not use something deemed undefined by the standard. In many cases, tools exist to help do this.

C.3 Character Sets

The examples in this book are written using the U.S. variant of the international 7-bit character set ISO 646-1983 called ASCII (ANSI3.4-1968). This can cause three problems for people who use C++ in an environment with a different character set:

[1] ASCII contains punctuation characters and operator symbols – such as `]`, `{`, and `!` – that are not available in some character sets.

[2] We need a notation for characters that do not have a convenient character representation (e.g., newline and ‘‘the character with value 17’’).

[3] ASCII doesn’t contain characters, such as – ζ , æ, and Π – that are used for writing languages other than English.

C.3.1 Restricted Character Sets

The ASCII special characters `[`, `]`, `{`, `}`, `|`, and `\` occupy character set positions designated as alphabetic by ISO. In most European national ISO-646 character sets, these positions are occupied by letters not found in the English alphabet. For example, the Danish national character set uses them for the vowels ***Æ***, ***æ***, ***Ø***, ***ø***, ***Å***, and ***å***. No significant amount of text can be written in Danish without them.

A set of trigraphs is provided to allow national characters to be expressed in a portable way using a truly standard minimal character set. This can be useful for interchange of programs, but it doesn’t make it easier for people to read programs. Naturally, the long-term solution to this problem is for C++ programmers to get equipment that supports both their native language and C++ well. Unfortunately, this appears to be infeasible for some, and the introduction of new equipment can be a frustratingly slow process. To help programmers stuck with incomplete character sets, C++ provides alternatives:

Keywords		Digraphs		Trigraphs	
and	`&&`	`<%`	`{`	`??=`	`#`
and_eq	`&=`	`%>`	`}`	`??(`	`[`
bitand	`&`	`<:`	`[`	`??<`	`{`
bitor	`|`	`:>`	`]`	`??/`	`\`
compl	`~`	`%:`	`#`	`??)`	`]`
not	`!`	`%:%:`	`##`	`??>`	`}`
or	`||`			`??'`	`^`
or_eq	`|=`			`??!`	`|`
xor	`^`			`??-`	`~`
xor_eq	`^=`			`???`	`?`
not_eq	`!=`				

Programs using the keywords and digraphs are far more readable than the equivalent programs written using trigraphs. However, if characters such as `{` are not available, trigraphs are necessary for putting ‘‘missing’’ characters into strings and character constants. For example, ´`{`´ becomes ´`??<`´.

Some people prefer the keywords such as ***and*** to their traditional operator notation.

C.3.2 Escape Characters

A few characters have standard names that use the backslash \ as an escape character:

Name	ASCII Name	C++ Name
newline	NL (LF)	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
alert	BEL	`\a`
backslash	\	`\\`
question mark	?	`\?`
single quote	'	`\'`
double quote	"	`\"`
octal number	*ooo*	`\`*ooo*
hex number	*hhh*	`\x`*hhh* ...

Despite their appearance, these are single characters.

It is possible to represent a character as a one-, two-, or three-digit octal number (\ followed by octal digits) or as a hexadecimal number (*x* followed by hexadecimal digits). There is no limit to the number of hexadecimal digits in the sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. For example:

Octal	Hexadecimal	Decimal	ASCII
`'\6'`	`'\x6'`	6	ACK
`'\60'`	`'\x30'`	48	`'0'`
`'\137'`	`'\x05f'`	95	`'_'`

This makes it possible to represent every character in the machine's character set and, in particular, to embed such characters in character strings (see §5.2.2). Using any numeric notation for characters makes a program nonportable across machines with different character sets.

It is possible to enclose more than one character in a character literal, for example ´***ab***´. Such uses are archaic, implementation-dependent, and best avoided.

When embedding a numeric constant in a string using the octal notation, it is wise always to use three digits for the number. The notation is hard enough to read without having to worry about whether or not the character after a constant is a digit. For hexadecimal constants, use two digits. Consider these examples:

```
char v1[] = "a\xah\129";      // 6 chars: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";      // 5 chars: 'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";      // 4 chars: 'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";     // 5 chars: 'a' '\xad' '\012' '7' '\0'
```

C.3.3 Large Character Sets

A C++ program may be written and presented to the user in character sets that are much richer than the 127 character ASCII set. Where an implementation supports larger character sets, identifiers, comments, character constants, and strings may contain characters such as å, β, and Γ. However, to be portable the implementation must map these characters into an encoding using only characters available to every C++ user. In principle, this translation into the C++ basic source character set (the set used in this book) occurs before the compiler does any other processing. Therefore, it does not affect the semantics of the program.

The standard encoding of characters from large character sets into the smaller set supported directly by C++ is presented as sequences of four or eight hexadecimal digits:

universal-character-name:
    \U *X X X X X X X X*
    \u *X X X X*

Here, ***X*** represents a hexadecimal digit. For example, ***\u1e2b***. The shorter notation ***\uXXXX*** is equivalent to ***\U0000XXXX***. A number of hexadecimal digits different from four or eight is a lexical error.

A programmer can use these character encodings directly. However, they are primarily meant as a way for an implementation that internally uses a small character set to handle characters from a large character set seen by the programmer.

If you rely on special environments to provide an extended character set for use in identifiers, the program becomes less portable. A program is hard to read unless you understand the natural language used for identifiers and comments. Consequently, for programs used internationally it is usually best to stick to English and ASCII.

C.3.4 Signed and Unsigned Characters

It is implementation-defined whether a plain ***char*** is considered signed or unsigned. This opens the possibility for some nasty surprises and implementation dependencies. For example:

```
char c = 255;    // 255 is ''all ones,'' hexadecimal 0xFF
int i = c;
```

What will be the value of ***i***? Unfortunately, the answer is undefined. On all implementations I know of, the answer depends on the meaning of the ''all ones'' ***char*** bit pattern when extended into an ***int***. On a SGI Challenge machine, a ***char*** is unsigned, so the answer is ***255***. On a Sun SPARC or an IBM PC, where a ***char*** is signed, the answer is ***-1***. In this case, the compiler might warn about the conversion of the literal ***255*** to the ***char*** value ***-1***. However, C++ does not offer a general mechanism for detecting this kind of problem. One solution is to avoid plain ***char*** and use the specific ***char*** types only. Unfortunately, some standard library functions, such as ***strcmp( )***, take plain ***char***s only (§20.4.1).

A ***char*** must behave identically to either a ***signed char*** or an ***unsigned char***. However, the three ***char*** types are distinct, so you can't mix pointers to different ***char*** types. For example:

```
void f(char c, signed char sc, unsigned char uc)
{
    char* pc = &uc;              // error: no pointer conversion
    signed char* psc = pc;       // error: no pointer conversion
    unsigned char* puc = pc;     // error: no pointer conversion
    psc = puc;                   // error: no pointer conversion
}
```

Variables of the three ***char*** types can be freely assigned to each other. However, assigning a too-large value to a signed ***char*** (§C.6.2.1) is still undefined. For example:

```
void f(char c, signed char sc, unsigned char uc)
{
    c = 255;   // undefined if plain chars are signed and have 8 bits

    c = sc;    // ok
    c = uc;    // undefined if plain chars are signed and if uc's value is too large
    sc = uc;   // undefined if uc's value is too large
    uc = sc;   // ok: conversion to unsigned
    sc = c;    // undefined if plain chars are unsigned and if c's value is too large
    uc = c;    // ok: conversion to unsigned
}
```

None of these potential problems occurs if you use plain ***char*** throughout.

C.4 Types of Integer Literals

In general, the type of an integer literal depends on its form, value, and suffix:

- If it is decimal and has no suffix, it has the first of these types in which its value can be represented: ***int***, ***long int***, ***unsigned long int***.
- If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: ***int***, ***unsigned int***, ***long int***, ***unsigned long int***.
- If it is suffixed by ***u*** or ***U***, its type is the first of these types in which its value can be represented: ***unsigned int***, ***unsigned long int***.
- If it is suffixed by ***l*** or ***L***, its type is the first of these types in which its value can be represented: ***long int***, ***unsigned long int***.
- If it is suffixed by ***ul***, ***lu***, ***uL***, ***Lu***, ***Ul***, ***lU***, ***UL***, or ***LU***, its type is ***unsigned long int***.

For example, ***100000*** is of type ***int*** on a machine with 32-bit ***int***s but of type ***long int*** on a machine with 16-bit ***int***s and 32-bit ***long***s. Similarly, ***0XA000*** is of type ***int*** on a machine with 32-bit ***int***s but of type ***unsigned int*** on a machine with 16-bit ***int***s. These implementation dependencies can be avoided by using suffixes: ***100000L*** is of type ***long int*** on all machines and ***0XA000U*** is of type ***unsigned int*** on all machines.

C.5 Constant Expressions

In places such as array bounds (§5.2), case labels (§6.3.2), and initializers for enumerators (§4.8), C++ requires a *constant expression*. A constant expression evaluates to an integral or enumeration constant. Such an expression is composed of literals (§4.3.1, §4.4.1, §4.5.1), enumerators (§4.8), and ***const***s initialized by constant expressions. In a template, an integer template parameter can also be used (§C.13.3). Floating literals (§4.5.1) can be used only if explicitly converted to an integral type. Functions, class objects, pointers, and references can be used as operands to the ***sizeof*** operator (§6.2) only.

Intuitively, constant expressions are simple expressions that can be evaluated by the compiler before the program is linked (§9.1) and starts to run.

C.6 Implicit Type Conversion

Integral and floating-point types (§4.1.1) can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information. Unfortunately, value-destroying conversions are also performed implicitly. This section provides a description of conversion rules, conversion problems, and their resolution.

C.6.1 Promotions

The implicit conversions that preserve values are commonly referred to as *promotions*. Before an arithmetic operation is performed, *integral promotion* is used to create ***int***s out of shorter integer types. Note that these promotions will *not* promote to ***long*** (unless the operand is a ***wchar_t*** or an enumeration that is already larger than an ***int***). This reflects the original purpose of these promotions in C: to bring operands to the ''natural'' size for arithmetic operations.

The integral promotions are:

- A ***char***, ***signed char***, ***unsigned char***, ***short int***, or ***unsigned short int*** is converted to an ***int*** if ***int*** can represent all the values of the source type; otherwise, it is converted to an ***unsigned int***.
- A ***wchar_t*** (§4.3) or an enumeration type (§4.8) is converted to the first of the following types that can represent all the values of its underlying type: ***int***, ***unsigned int***, ***long***, or ***unsigned long***.
- A bit-field (§C.8.1) is converted to an ***int*** if ***int*** can represent all the values of the bit-field; otherwise, it is converted to ***unsigned int*** if ***unsigned int*** can represent all the values of the bit-field. Otherwise, no integral promotion applies to it.
- A ***bool*** is converted to an ***int***; ***false*** becomes ***0*** and ***true*** becomes ***1***.

Promotions are used as part of the usual arithmetic conversions (§C.6.3).

C.6.2 Conversions

The fundamental types can be converted into each other in a bewildering number of ways. In my opinion, too many conversions are allowed. For example:

```
void f(double d)
{
    char c = d;    // beware: double-precision floating-point to char conversion
}
```

When writing code, you should always aim to avoid undefined behavior and conversions that quietly throw away information. A compiler can warn about many questionable conversions. Fortunately, many compilers actually do.

C.6.2.1 Integral Conversions

An integer can be converted to another integer type. An enumeration value can be converted to an integer type.

If the destination type is ***unsigned***, the resulting value is simply as many bits from the source as will fit in the destination (high-order bits are thrown away if necessary). More precisely, the result is the least unsigned integer congruent to the source integer modulo *2* to the ***n***th, where *n* is the number of bits used to represent the unsigned type. For example:

```
unsigned char uc = 1023; // binary 1111111111: uc becomes binary 11111111; that is, 255
```

If the destination type is ***signed***, the value is unchanged if it can be represented in the destination type; otherwise, the value is implementation-defined:

```
signed char sc = 1023;    // implementation-defined
```

Plausible results are *255* and *-1* (§C.3.4).

A Boolean or enumeration value can be implicitly converted to its integer equivalent (§4.2, §4.8).

C.6.2.2 Floating-Point Conversions

A floating-point value can be converted to another floating-point type. If the source value can be exactly represented in the destination type, the result is the original numeric value. If the source value is between two adjacent destination values, the result is one of those values. Otherwise, the behavior is undefined. For example:

```
float f = FLT_MAX;        // largest float value
double d = f;             // ok: d == f
float f2 = d;             // ok: f2 == f
double d3 = DBL_MAX;      // largest double value
float f3 = d3;            // undefined if FLT_MAX<DBL_MAX
```

C.6.2.3 Pointer and Reference Conversions

Any pointer to an object type can be implicitly converted to a ***void**** (§5.6). A pointer (reference) to a derived class can be implicitly converted to a pointer (reference) to an accessible and unambiguous base (§12.2). Note that a pointer to function or a pointer to member cannot be implicitly converted to a ***void****.

A constant expression (§C.5) that evaluates to ***0*** can be implicitly converted to any pointer or pointer to member type (§5.1.1). For example:

```
int* p =
            !    !   !     !   !          !
           ! !  !   !     !   !          !
          !  ! !    !     !   !          !
         !    !     !!!!!!  !!!!!    !!!!1;
```

A ***T**** can be implicitly converted to a ***const T**** (§5.4.1). Similarly, a ***T&*** can be implicitly converted to a ***const T&***.

C.6.2.4 Pointer-to-Member Conversions

Pointers and references to members can be implicitly converted as described in §15.5.1.

C.6.2.5 Boolean Conversions

Pointers, integral, and floating-point values can be implicitly converted to ***bool*** (§4.2). A nonzero value converts to ***true***; a zero value converts to ***false***. For example:

```
void f(int* p, int i)
{
    bool is_not_zero = p;      // true if p!=0
    bool b2 = i;               // true if i!=0
}
```

C.6.2.6 Floating-Integral Conversions

When a floating-point value is converted to an integer value, the fractional part is discarded. In other words, conversion from a floating-point type to an integer type truncates. For example, the value of ***int(1.6)*** is ***1***. The behavior is undefined if the truncated value cannot be represented in the destination type. For example:

```
int i = 2.7;        // i becomes 2
char b = 2000.7;    // undefined for 8-bit chars: 2000 cannot be represented as an 8-bit char
```

Conversions from integer to floating types are as mathematically correct as the hardware allows. Loss of precision occurs if an integral value cannot be represented exactly as a value of the floating type. For example,

```
int i = float(1234567890);
```

left ***i*** with the value ***1234567936*** on a machine, where both ***int***s and ***float***s are represented using 32 bits.

Clearly, it is best to avoid potentially value-destroying implicit conversions. In fact, compilers can detect and warn against some obviously dangerous conversions, such as floating to integral and ***long int*** to ***char***. However, general compile-time detection is impractical, so the programmer must be careful. When ‘‘being careful’’ isn’t enough, the programmer can insert explicit checks. For example:

```
class check_failed { };

char checked(int i)
{
    char c = i;                          // warning: not portable (§C.6.2.1)
    if (i != c) throw check_failed();
    return c;
}
void my_code(int i)
{
    char c = checked(i);
    // ...
}
```

To truncate in a way that is guaranteed to be portable requires the use of ***numeric_limits*** (§22.2).

C.6.3 Usual Arithmetic Conversions

These conversions are performed on the operands of a binary operator to bring them to a common type, which is then used as the type of the result:

[1] If either operand is of type ***long double***, the other is converted to ***long double***.
– Otherwise, if either operand is ***double***, the other is converted to ***double***.
– Otherwise, if either operand is ***float***, the other is converted to ***float***.
– Otherwise, integral promotions (§C.6.1) are performed on both operands.

[2] Then, if either operand is ***unsigned long***, the other is converted to ***unsigned long***.
– Otherwise, if one operand is a ***long int*** and the other is an ***unsigned int***, then if a ***long int*** can represent all the values of an ***unsigned int***, the ***unsigned int*** is converted to a ***long int***; otherwise, both operands are converted to ***unsigned long int***.
– Otherwise, if either operand is ***long***, the other is converted to ***long***.
– Otherwise, if either operand is ***unsigned***, the other is converted to ***unsigned***.
– Otherwise, both operands are ***int***.

C.7 Multidimensional Arrays

It is not uncommon to need a vector of vectors, a vector of vector of vectors, etc. The issue is how to represent these multidimensional vectors in C++. Here, I first show how to use the standard library ***vector*** class. Next, I present multidimensional arrays as they appear in C and C++ programs using only built-in facilities.

C.7.1 Vectors

The standard ***vector*** (§16.3) provides a very general solution:

```
vector< vector<int> > m;
```

This creates a vector of vectors of integers that initially contains no elements. We could initialize it to a three-by-five matrix like this:

```
void init_m()
{
    m.resize(3);                        // m now holds 3 empty vectors
    for (int i = 0; i<m.size(); i++) {
        m[i].resize(5);                 // now each of m's vectors holds 5 ints
        for (int j = 0; j<m[i].size(); j++) m[i][j] = 10*i+j;
    }
}
```

or graphically:

Each ***vector*** implementation holds a pointer to its elements plus the number of elements. The elements are typically held in an array. For illustration, I gave each ***int*** an initial value representing its coordinates.

It is not necessary for the ***vector<int>***s in the ***vector< vector<int> >*** to have the same size.

Accessing an element is done by indexing twice. For example, ***m[i][j]*** is the ***j***th element of the ***i***th vector. We can print ***m*** like this:

```
void print_m()
{
    for (int i = 0; i<m.size(); i++) {
        for (int j = 0; j<m[i].size(); j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

which gives:

```
0   1   2   3   4
10  11  12  13  14
20  21  22  23  24
```

C.7.2 Arrays

The built-in arrays are a major source of errors – especially when they are used to build multidimensional arrays. For novices, they are also a major source of confusion. Wherever possible, use ***vector***, ***list***, ***valarray***, ***string***, etc.

Multidimensional arrays are represented as arrays of arrays. A three-by-five array is declared like this:

```
int ma[3][5]; // 3 arrays with 5 ints each
```

For arrays, the dimensions must be given as part of the definition. We can initialize ***ma*** like this:

```
void init_ma()
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) ma[i][j] = 10*i+j;
    }
}
```

or graphically:

ma:															
	00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

The array ***ma*** is simply 15 ***ints*** that we access as if it were 3 arrays of 5 ***int***s. In particular, there is no single object in memory that is the matrix ***ma*** – only the elements are stored. The dimensions ***3*** and ***5*** exist in the compiler source only. When we write code, it is our job to remember them somehow and supply the dimensions where needed. For example, we might print ***ma*** like this:

```
void print_ma()
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) cout << ma[i][j] << '\t';
        cout << '\n';
    }
}
```

The comma notation used for array bounds in some languages cannot be used in C++ because the comma (**,**) is a sequencing operator (§6.2.2). Fortunately, most mistakes are caught by the compiler. For example:

```
int bad[3,5];          // error: comma not allowed in constant expression
int good[3][5];        // 3 arrays with 5 ints each
int ouch = good[1,4];  // error: int initialized by int* (good[1,4] means good[4], which is an int*)
int nice = good[1][4];
```

C.7.3 Passing Multidimensional Arrays

Consider defining a function to manipulate a two-dimensional matrix. If the dimensions are known at compile time, there is no problem:

```
void print_m35(int m[3][5])
{
    for (int i = 0; i<3; i++) {
        for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

A matrix represented as a multidimensional array is passed as a pointer (rather than copied; §5.3). The first dimension of an array is irrelevant to the problem of finding the location of an element; it simply states how many elements (here *3*) of the appropriate type (here *int*[*5*]) are present. For example, look at the previous representation of *ma* and note that by our knowing only that the second dimension is *5*, we can locate *ma*[*i*][*5*] for any *i*. The first dimension can therefore be passed as an argument:

```
void print_mi5(int m[][5], int dim1)
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<5; j++) cout << m[i][j] << '\t';
        cout << '\n';
    }
}
```

The difficult case is when both dimensions need to be passed. The ''obvious solution'' simply does not work:

```
void print_mij(int m[][], int dim1, int dim2) // doesn't behave as most people would think
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<dim2; j++) cout << m[i][j] << '\t';      // surprise!
        cout << '\n';
    }
}
```

First, the argument declaration *m*[][] is illegal because the second dimension of a multidimensional array must be known in order to find the location of an element. Second, the expression *m*[*i*][*j*] is (correctly) interpreted as *(*(*m*+*i*)+*j*), although that is unlikely to be what the programmer intended. A correct solution is:

```
void print_mij(int* m, int dim1, int dim2)
{
    for (int i = 0; i<dim1; i++) {
        for (int j = 0; j<dim2; j++) cout << m[i*dim2+j] << '\t'; // obscure
        cout << '\n';
    }
}
```

The expression used for accessing the members in *print_mij*( ) is equivalent to the one the compiler generates when it knows the last dimension.

To call this function, we pass a matrix as an ordinary pointer:

```
int main()
{
    int v[3][5] = { {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24} };

    print_m35(v);
    print_mi5(v,3);
    print_mij(&v[0][0],3,5);
}
```

Note the use of `&v[0][0]` for the last call; `v[0]` would do because it is equivalent, but `v` would be a type error. This kind of subtle and messy code is best hidden. If you must deal directly with multidimensional arrays, consider encapsulating the code relying on it. In that way, you might ease the task of the next programmer to touch the code. Providing a multidimensional array type with a proper subscripting operator saves most users from having to worry about the layout of the data in the array (§22.4.6).

The standard ***vector*** (§16.3) doesn't suffer from these problems.

C.8 Saving Space

When programming nontrivial applications, there often comes a time when you want more memory space than is available or affordable. There are two ways of squeezing more space out of what is available:

[1] Put more than one small object into a byte.

[2] Use the same space to hold different objects at different times.

The former can be achieved by using *fields*, and the latter by using *unions*. These constructs are described in the following sections. Many uses of fields and unions are pure optimizations, and these optimizations are often based on nonportable assumptions about memory layouts. Consequently, the programmer should think twice before using them. Often, a better approach is to change the way data is managed, for example, to rely more on dynamically allocated store (§6.2.6) and less on preallocated (static) storage.

C.8.1 Fields

It seems extravagant to use a whole byte (a ***char*** or a ***bool***) to represent a binary variable – for example, an on/off switch – but a ***char*** is the smallest object that can be independently allocated and addressed in C++ (§5.1). It is possible, however, to bundle several such tiny variables together as *fields* in a ***struct***. A member is defined to be a field by specifying the number of bits it is to occupy. Unnamed fields are allowed. They do not affect the meaning of the named fields, but they can be used to make the layout better in some machine-dependent way:

```
struct PPN {                // R6000 Physical Page Number
    unsigned int PFN : 22;  // Page Frame Number
    int : 3;                // unused
    unsigned int CCA : 3;   // Cache Coherency Algorithm
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

This example also illustrates the other main use of fields: to name parts of an externally imposed layout. A field must be of an integral or enumeration type (§4.1.1). It is not possible to take the address of a field. Apart from that, however, it can be used exactly like other variables. Note that a ***bool*** field really can be represented by a single bit. In an operating system kernel or in a debugger, the type ***PPN*** might be used like this:

```
void part_of_VM_system(PPN* p)
{
        // ...

        if (p->dirty) { // contents changed
                // copy to disc
                p->dirty = 0;
        }

        // ...
}
```

Surprisingly, using fields to pack several variables into a single byte does not necessarily save space. It saves data space, but the size of the code needed to manipulate these variables increases on most machines. Programs have been known to shrink significantly when binary variables were converted from bit fields to characters! Furthermore, it is typically much faster to access a ***char*** or an ***int*** than to access a field. Fields are simply a convenient shorthand for using bitwise logical operators (§6.2.4) to extract information from and insert information into part of a word.

C.8.2 Unions

A ***union*** is a ***struct*** in which all members are allocated at the same address so that the ***union*** occupies only as much space as its largest member. Naturally, a ***union*** can hold a value for only one member at a time. For example, consider a symbol table entry that holds a name and a value:

```
enum Type { S, I };

struct Entry {
        char* name;
        Type t;
        char* s;    // use s if t==S
        int i;      // use i if t==I
};

void f(Entry* p)
{
        if (p->t == S) cout << p->s;
        // ...
}
```

The members ***s*** and ***i*** can never be used at the same time, so space is wasted. It can be easily recovered by specifying that both should be members of a ***union***, like this:

```
union Value {
        char* s;
        int i;
};
```

The language doesn't keep track of which kind of value is held by a ***union***, so the programmer must still do that:

```
struct Entry {
    char* name;
    Type t;
    Value v;   // use v.s if t==S; use v.i if t==I
};

void f(Entry* p)
{
    if (p->t == S) cout << p->v.s;
    // ...
}
```

Unfortunately, the introduction of the ***union*** forced us to rewrite code to say ***v.s*** instead of plain ***s***. This can be avoided by using an *anonymous union*, which is a union that doesn't have a name and consequently doesn't define a type. Instead, it simply ensures that its members are allocated at the same address:

```
struct Entry {
    char* name;
    Type t;
    union {
        char* s;   // use s if t==S
        int i;     // use i if t==I
    };
};

void f(Entry* p)
{
    if (p->t == S) cout << p->s;
    // ...
}
```

This leaves all code using an ***Entry*** unchanged.

Using a ***union*** so that its value is always read using the member through which it was written is a pure optimization. However, it is not always easy to ensure that a ***union*** is used in this way only, and subtle errors can be introduced through misuse. To avoid errors, one can encapsulate a ***union*** so that the correspondence between a type field and access to the ***union*** members can be guaranteed (§10.6[20]).

Unions are sometimes misused for ''type conversion.'' This misuse is practiced mainly by programmers trained in languages that do not have explicit type conversion facilities, where cheating is necessary. For example, the following ''converts'' an ***int*** to an ***int****** simply by assuming bitwise equivalence:

```
union Fudge {
    int i;
    int* p;
};
```

```
int* cheat(int i)
{
    Fudge a;
    a.i = i;
    return a.p;     // bad use
}
```

This is not really a conversion at all. On some machines, an ***int*** and an ***int****** do not occupy the same amount of space, while on others, no integer can have an odd address. Such use of a ***union*** is dangerous and nonportable, and there is an explicit and portable way of specifying type conversion (§6.2.7).

Unions are occasionally used deliberately to avoid type conversion. One might, for example, use a ***Fudge*** to find the representation of the pointer ***0***:

```
int main()
{
    Fudge foo;
    foo.p = 0;
    cout << "the integer value of the pointer 0 is " << foo.i << '\n';
}
```

C.8.3 Unions and Classes

Many nontrivial ***union***s have some members that are much larger than the most frequently-used members. Because the size of a ***union*** is at least as large as its largest member, space is wasted. This waste can often be eliminated by using a set of derived classes instead of a ***union***.

A class with a constructor, destructor, or copy operation cannot be the type of a ***union*** member (§10.4.12) because the compiler would not know which member to destroy.

C.9 Memory Management

There are three fundamental ways of using memory in C++:

Static memory, in which an object is allocated by the linker for the duration of the program. Global and namespace variables, ***static*** class members (§10.2.4), and ***static*** variables in functions (§7.1.2) are allocated in static memory. An object allocated in static memory is constructed once and persists to the end of the program. It always has the same address. Static objects can be a problem in programs using threads (shared-address space concurrency) because they are shared and require locking for proper access.

Automatic memory, in which function arguments and local variables are allocated. Each entry into a function or a block gets its own copy. This kind of memory is automatically created and destroyed; hence the name automatic memory. Automatic memory is also said ‘‘to be on the stack.’’ If you absolutely must be explicit about this, C++ provides the redundant keyword ***auto***.

Free store, from which memory for objects is explicitly requested by the program and where a program can free memory again once it is done with it (using ***new*** and ***delete***). When a program needs more free store, ***new*** requests it from the operating system. Typically, the free

store (also called *dynamic memory* or *the heap*) grows throughout the lifetime of a program because no memory is ever returned to the operating system for use by other programs.

As far as the programmer is concerned, automatic and static storage are used in simple, obvious, and implicit ways. The interesting question is how to manage the free store. Allocation (using ***new***) is simple, but unless we have a consistent policy for giving memory back to the free store manager, memory will fill up – especially for long-running programs.

The simplest strategy is to use automatic objects to manage corresponding objects in free store. Consequently, many containers are implemented as handles to elements stored in the free store (§25.7). For example, an automatic ***String*** (§11.12) manages a sequence of characters on the free store and automatically frees that memory when it itself goes out of scope. All of the standard containers (§16.3, Chapter 17, Chapter 20, §22.4) can be conveniently implemented in this way.

C.9.1 Automatic Garbage Collection

When this regular approach isn't sufficient, the programmer might use a memory manager that finds unreferenced objects and reclaims their memory in which to store new objects. This is usually called *automatic garbage collection*, or simply *garbage collection*. Naturally, such a memory manager is called a *garbage collector*.

The fundamental idea of garbage collection is that an object that is no longer referred to in a program will not be accessed again, so its memory can be safely reused for some new object. For example:

```
void f()
{
    int* p = new int;
    p = 0;
    char* q = new char;
}
```

Here, the assignment ***p=0*** makes the ***int*** unreferenced so that its memory can be used for some other new object. Thus, the ***char*** might be allocated in the same memory as the ***int*** so that ***q*** holds the value that ***p*** originally had.

The standard does not require that an implementation supply a garbage collector, but garbage collectors are increasingly used for C++ in areas where their costs compare favorably to those of manual management of free store. When comparing costs, consider the run time, memory usage, reliability, portability, monetary cost of programming, monetary cost of a garbage collector, and predictability of performance.

C.9.1.1 Disguised Pointers

What should it mean for an object to be unreferenced? Consider:

```
void f()
{
    int* p = new int;
    long i1 = reinterpret_cast<long>(p)&0xFFFF0000;
    long i2 = reinterpret_cast<long>(p)&0x0000FFFF;
    p = 0;
```

```
        // point #1: no pointer to the int exists here

        p = reinterpret_cast<int*>(i1|i2);
        // now the int is referenced again
}
```

Often, pointers stored as non-pointers in a program are called ''disguised pointers.'' In particular, the pointer originally held in ***p*** is disguised in the integers ***i1*** and ***i2***. However, a garbage collector need not be concerned about disguised pointers. If the garbage collector runs at point ***#1***, the memory holding the ***int*** can be reclaimed. In fact, such programs are not guaranteed to work even if a garbage collector is not used because the use of ***reinterpret_cast*** to convert between integers and pointers is at best implementation-defined.

A ***union*** that can hold both pointers and non-pointers presents a garbage collector with a special problem. In general, it is not possible to know whether such a ***union*** contains a pointer. Consider:

```
union U {       // union with both pointer and non-pointer members
        int* p;
        int i;
};

void f(U u, U u2, U u3)
{
        u.p = new int;
        u2.i = 999999;
        u.i = 8;
        // ...
}
```

The safe assumption is that any value that appears in such a ***union*** is a pointer value. A clever garbage collector can do somewhat better. For example, it may notice that (for a given implementation) ***int***s are not allocated with odd addresses and that no objects are allocated with an address as low as ***8***. Noticing this will save the garbage collector from having to assume that objects containing locations ***999999*** and ***8*** are used by ***f*( )**.

C.9.1.2 Delete

If an implementation automatically collects garbage, the ***delete*** and ***delete*[ ]** operators are no longer needed to free memory for potential reuse. Thus, a user relying on a garbage collector could simply refrain from using these operators. However, in addition to freeing memory, ***delete*** and ***delete*[ ]** invoke destructors.

In the presence of a garbage collector,

```
delete p;
```

invokes the destructor for the object pointed to by ***p*** (if any). However, reuse of the memory can be postponed until it is collected. Recycling lots of objects at once can help limit fragmentation (§C.9.1.4). It also renders harmless the otherwise serious mistake of deleting an object twice in the important case where the destructor simply deletes memory.

As always, access to an object after it has been deleted is undefined.

C.9.1.3 Destructors

When an object is about to be recycled by a garbage collector, two alternatives exist:

[1] Call the destructor (if any) for the object.

[2] Treat the object as raw memory (don't call its destructor).

By default, a garbage collector should choose option (2) because objects created using ***new*** and never ***delete***d are never destroyed. Thus, one can see a garbage collector as a mechanism for simulating an infinite memory.

It is possible to design a garbage collector to invoke the destructors for objects that have been specifically ''registered'' with the collector. However, there is no standard way of ''registering'' objects. Note that it is always important to destroy objects in an order that ensures that the destructor for one object doesn't refer to an object that has been previously destroyed. Such ordering isn't easily achieved by a garbage collector without help from the programmer.

C.9.1.4 Memory Fragmentation

When a lot of objects of varying sizes are allocated and freed, the memory *fragments*. That is, much of memory is consumed by pieces of memory that are too small to use effectively. The reason is that a general allocator cannot always find a piece of memory of the exact right size for an object. Using a slightly larger piece means that a smaller fragment of memory remains. After running a program for a while with a naive allocator, it is not uncommon to find half the available memory taken up with fragments too small ever to get reused.

Several techniques exist for coping with fragmentation. The simplest is to request only larger chunks of memory from the allocator and use each such chunk for objects of the same size (§15.3, §19.4.2). Because most allocations and deallocations are of small objects of types such as tree nodes, links, etc., this technique can be very effective. An allocator can sometimes apply similar techniques automatically. In either case, fragmentation is further reduced if all of the larger ''chunks'' are of the same size (say, the size of a page) so that they themselves can be allocated and reallocated without fragmentation.

There are two main styles of garbage collectors:

[1] A *copying collector* moves objects in memory to compact fragmented space.

[2] A *conservative collector* allocates objects to minimize fragmentation.

From a C++ point of view, conservative collectors are preferable because it is very hard (probably impossible in real programs) to move an object and modify all pointers to it correctly. A conservative collector also allows C++ code fragments to coexist with code written in languages such as C. Traditionally, copying collectors have been favored by people using languages (such as Lisp and Smalltalk) that deal with objects only indirectly through unique pointers or references. However, modern conservative collectors seem to be at least as efficient as copying collectors for larger programs, in which the amount of copying and the interaction between the allocator and a paging system become important. For smaller programs, the ideal of simply never invoking the collector is often achievable – especially in C++, where many objects are naturally automatic.

C.10 Namespaces

This section presents minor points about namespaces that look like technicalities, yet frequently surface in discussions and in real code.

C.10.1 Convenience vs. Safety

A *using-declaration* adds a name to a local scope. A *using-directive* does not; it simply renders names accessible in the scope in which they were declared. For example:

```
namespace X {
    int i, j, k;
}

int k;

void f1()
{
    int i = 0;
    using namespace X; // make names from X accessible
    i++;               // local i
    j++;               // X::j
    k++;               // error: X::k or global k ?
    ::k++;             // the global k
    X::k++;            // X's k
}

void f2()
{
    int i = 0;
    using X::i;    // error: i declared twice in f2()
    using X::j;
    using X::k;    // hides global k

    i++;
    j++;           // X::j
    k++;           // X::k
}
```

A locally declared name (declared either by an ordinary declaration or by a *using-declaration*) hides nonlocal declarations of the same name, and any illegal overloadings of the name are detected at the point of declaration.

Note the ambiguity error for ***k++*** in ***f1()***. Global names are not given preference over names from namespaces made accessible in the global scope. This provides significant protection against accidental name clashes, and – importantly – ensures that there are no advantages to be gained from polluting the global namespace.

When libraries declaring *many* names are made accessible through *using-directive*s, it is a significant advantage that clashes of unused names are not considered errors.

The global scope is just another namespace. The global namespace is odd only in that you don't have to mention its name in an explicit qualification. That is, ***::k*** means ''look for ***k*** in the global namespace and in namespaces mentioned in ***using-directive***s in the global namespace,''

whereas *X*::*k* means ''the *k* declared in namespace *X* and namespaces mentioned in *using-directive*s in *X*'' (§8.2.8).

I hope to see a radical decrease in the use of global names in new programs using namespaces compared to traditional C and C++ programs. The rules for namespaces were specifically crafted to give no advantages to a ''lazy'' user of global names over someone who takes care not to pollute the global scope.

C.10.2 Nesting of Namespaces

One obvious use of namespaces is to wrap a complete set of declarations and definitions in a separate namespace:

```
namespace X {
    // all my declarations
}
```

The list of declarations will, in general, contain namespaces. Thus, nested namespaces are allowed. This is allowed for practical reasons, as well as for the simple reason that constructs ought to nest unless there is a strong reason for them not to. For example:

```
void h();

namespace X {
    void g();
    // ...
    namespace Y {
        void f();
        void ff();
        // ...
    }
}
```

The usual scope and qualification rules apply:

```
void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f();        // error: no f() in X
    Y::f();     // ok
}

void h()
{
    f();            // error: no global f()
    Y::f();         // error: no global Y
    X::f();         // error: no f() in X
    X::Y::f();      // ok
}
```

C.10.3 Namespaces and Classes

A namespace is a named scope. A class is a type defined by a named scope that describes how objects of that type can be created and used. Thus, a namespace is a simpler concept than a class and ideally a class would be defined as a namespace with a few extra facilities included. This is almost the case. A namespace is open (§8.2.9.3), but a class is closed. This difference stems from the observation that a class needs to define the layout of an object and that is best done in one place. Furthermore, *using-declaration*s and *using-directive*s can be applied to classes only in a very restricted way (§15.2.2).

Namespaces are preferred over classes when all that is needed is encapsulation of names. In this case, the class apparatus for type checking and for creating objects is not needed; the simpler namespace concept suffices.

C.11 Access Control

This section presents a few technical examples illustrating access control to supplement those presented in §15.3.

C.11.1 Access to Members

Consider:

```
class X {
// private by default:
        int priv;
protected:
        int prot;
public:
        int publ;
        void m();
};
```

The member `X::m()` has unrestricted access:

```
void X::m()
{
        priv = 1;   // ok
        prot = 2;   // ok
        publ = 3;   // ok
}
```

A member of a derived class has access to public and protected members (§15.3):

```
class Y : public X {
        void mderived();
};
```

```
void Y::mderived()
{
    priv = 1;   // error: priv is private
    prot = 2;   // ok: prot is protected and mderived() is a member of the derived class Y
    publ = 3;   // ok: publ is public
}
```

A global function can access only the public members:

```
void f(Y* p)
{
    p->priv = 1;    // error: priv is private
    p->prot = 2;    // error: prot is protected and f() is not a friend or a member of X or Y
    p->publ = 3;    // ok: publ is public
}
```

C.11.2 Access to Base Classes

Like a member, a base class can be declared ***private***, ***protected***, or ***public***. Consider:

```
class X {
public:
    int a;
    // ...
};

class Y1 : public X { };
class Y2 : protected X { };
class Y3 : private X { };
```

Because ***X*** is a public base of ***Y1***, any function can (implicitly) convert a ***Y1**** to an ***X**** where needed just as it can access the public members of class ***X***. For example:

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // ok: X is a public base class of Y1
    py1->a = 7;     // ok

    px = py2;       // error: X is a protected base of Y2
    py2->a = 7;     // error

    px = py3;       // error: X is a private base of Y3
    py3->a = 7;     // error
}
```

Consider:

```
class Y2 : protected X { };
class Z2 : public Y2 { void f(Y1*, Y2*, Y3*); };
```

Because ***X*** is a protected base of ***Y2***, only members and friends of ***Y2*** and members and friends of ***Y2***'s derived classes (e.g., ***Z2***) can (implicitly) convert a ***Y2**** to an ***X**** where needed, just as they can access the public and protected members of class ***X***. For example:

```
void Z2::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // ok: X is a public base class of Y1
    py1->a = 7;     // ok

    px = py2;       // ok: X is a protected base of Y2, and Z2 is derived from Y2
    py2->a = 7;     // ok

    px = py3;       // error: X is a private base of Y3
    py3->a = 7;     // error
}
```

Consider finally:

```
class Y3 : private X { void f(Y1*, Y2*, Y3*); };
```

Because ***X*** is a private base of ***Y3***, only members and friends of ***Y3*** can (implicitly) convert a ***Y3**** to an ***X**** where needed, just as they can access the public and protected members of class ***X***. For example:

```
void Y3::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // ok: X is a public base class of Y1
    py1->a = 7;     // ok

    px = py2;       // error: X is a protected base of Y2
    py2->a = 7;     // error

    px = py3;       // ok: X is a private base of Y3, and Y3::f() is a member of Y3
    py3->a = 7;     // ok
}
```

C.11.3 Access to Member Class

The members of a member class have no special access to members of an enclosing class. Similarly members of an enclosing class have no special access to members of a nested class; the usual access rules (§10.2.2) shall be obeyed. For example:

```
class Outer {
    typedef int T;
    int i;
public:
    int i2;
    static int s;

    class Inner {
        int x;
        T y; // error: Outer::T is private
    public:
        void f(Outer* p, int v);
    };
```

```
        int g(Inner* p);
};

void Outer::Inner::f(Outer* p, int v)
{
        p->i = v;          // error: Outer::i is private
        p->i2 = v;         // ok: Outer::i2 is public
}

int Outer::g(Inner* p)
{
        p->f(this,2);      // ok: Inner::f() is public
        return p->x;       // error: Inner::x is private
}
```

However, it is often useful to grant a member class access to its enclosing class. This can be done by making the member a ***friend***. For example:

```
class Outer {
        typedef int T;
        int i;
public:
        class Inner;              // forward declaration of member class
        friend class Inner;       // grant access to Outer::Inner

        class Inner {
                int x;
                T y;          // ok: Inner is a friend
        public:
                void f(Outer* p, int v);
        };
};

void Outer::Inner::f(Outer* p, int v)
{
        p->i = v;  // ok: Inner is a friend
}
```

C.11.4 Friendship

Friendship is neither inherited nor transitive. For example:

```
class A {
        friend class B;
        int a;
};

class B {
        friend class C;
};
```

```
class C {
    void f(A* p)
    {
        p->a++;   // error: C is not a friend of A, despite being a friend of a friend of A
    }
};

class D : public B {
    void f(A* p)
    {
        p->a++;   // error: D is not a friend of A, despite being derived from a friend of A
    }
};
```

C.12 Pointers to Data Members

Naturally, the notion of pointer to member (§15.5) applies to data members and to member functions with arguments and return types. For example:

```
struct C {
    char* val;
    int i;
    void print(int x) { cout << val << x << '\n' ; }
    void f1();
    int f2();
    C(char* v) { val = v; }
};

typedef void (C::*PMFI)(int);        // pointer to member function of C taking an int
typedef char* C::*PM;                // pointer to char* data member of C

void f(C& z1, C& z2)
{
    C* p = &z2;
    PMFI pf = &C::print;
    PM pm = &C::val;

    z1.print(1);
    (z1.*pf)(2);
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);

    pf = &C::f1;     // error: return type mismatch
    pf = &C::f2;     // error: argument type mismatch
    pm = &C::i;      // error: type mismatch
    pm = pf;         // error: type mismatch
}
```

The type of a pointer to function is checked just like any other type.

C.13 Templates

A class template specifies how a class can be generated given a suitable set of template arguments. Similarly, a function template specifies how a function can be generated given a suitable set of template arguments. Thus, a template can be used to generate types and executable code. With this expressive power comes some complexity. Most of this complexity relates to the variety of contexts involved in the definition and use of templates.

C.13.1 Static Members

A class template can have ***static*** members. Each class generated from the template has its own copy of the static members. Static members must be separately defined and can be specialized. For example:

```
template<class T> class X {
    // ...
    static T def_val;
    static T* new_X(T a = def_val);
};

template<class T> T X<T>::def_val(0,0);
template<class T> T* X<T>::new_X(T a) { /* ... */ }

template<> int X<int>::def_val<int> = 0;
template<> int* X<int>::new_X<int>(int i) { /* ... */ }
```

If you want to share an object or function among all members of every class generated from a template, you can place it in a non-templatized base class. For example:

```
struct B {
    static B* nil;    // to be used as common null pointer for every class derived from B
};

template<class T> class X : public B {
    // ...
};

B* B::nil = 0;
```

C.13.2 Friends

Like other classes, a template class can have friends. For example, comparison operators are typically friends, so we can rewrite class ***Basic_ops*** from §13.6 like this:

```
template <class C> class Basic_ops { // basic operators on containers
    friend bool operator==(const C&, const C&); // compare elements
    friend bool operator!=(const C&, const C&);
    // ...
};
```

```
template<class T> class Math_container : public Basic_ops< Math_container<T> > {
    // ...
};
```

Like a member, a friend declared within a template is itself a template and is defined using the template parameters of its class. For example:

```
template <class C> bool operator==(const C& a, const C& b)
{
    if (a.size() != b.size()) return false;
    for (int i = 0; i<a.size(); ++i)
        if (a[i] != b[i]) return false;
    return true;
}
```

Friends do not affect the scope in which the template class is defined, nor do they affect the scope in which the template is used. Instead, friend functions and operators are found using a lookup based on their argument types (§11.2.4, §11.5.1). Like a member function, a friend function is instantiated (§C.13.9.1) only if it is called.

C.13.3 Templates as Template Parameters

Sometimes it is useful to pass templates – rather than classes or objects – as template arguments. For example:

```
template<class T, template<class> class C> class Xrefd {
    C<T> mems;
    C<T*> refs;
    // ...
};

Xrefd<Entry, vector> x1;    // store cross references for Entries in a vector

Xrefd<Record, set> x2;      // store cross references for Records in a set
```

To use a template as a template parameter, you specify its required arguments. The template parameters of the template parameter need to be known in order to use the template parameter. The point of using a template as a template parameter is usually that we want to instantiate it with a variety of argument types (such as ***T*** and ***T***** in the previous example). That is, we want to express the member declarations of a template in terms of another template, but we want that other template to be a parameter so that it can be specified by users.

The common case in which a template needs a container to hold elements of its own argument type is often better handled by passing the container type (§13.6, §17.3.1).

Only class templates can be template arguments.

C.13.4 Deducing Function Template Arguments

A compiler can deduce a type template argument, ***T*** or ***TT***, and a non-type template argument, ***I***, from a template function argument with a type composed of the following constructs:

```
T                                   const T                          volatile T
T*                                  T&                               T[constant_expression]
type[I]                             class_template_name<T>           class_template_name<I>
TT<T>                               T<I>                             T<>
T type::*                           T T::*                           type T::*
T (*)(args)                         type (T::*)(args)                T (type::*)(args)
type (type::*)(args_TI)             T (T::*)(args_TI)                type (T::*)(args_TI)
T (type::*)(args_TI)                type (*)(args_TI)
```

Here, ***args_TI*** is a parameter list from which a ***T*** or an ***I*** can be determined by recursive application of these rules and ***args*** is a parameter list that does not allow deduction. If not all parameters can be deduced in this way, a call is ambiguous. For example:

```
template<class T, class U> void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
    f(p,g);    // T is char, U is int
    f(p,h);    // error: can't deduce U
}
```

Looking at the arguments of the first call of ***f*( )**, we easily deduce the template arguments. Looking at the second call of ***f*( )**, we see that ***h*( )** doesn't match the pattern ***U*(*)(*U*)** because ***h*( )**'s argument and return types differ.

If a template parameter can be deduced from more than one function argument, the same type must be the result of each deduction. Otherwise, the call is an error. For example:

```
template<class T> void f(T i, T* p);

void g(int i)
{
    f(i,&i);                  // ok
    f(i,"Remember!");         // error, ambiguous: T is int or T is char?
}
```

C.13.5 Typename and Template

To make generic programming easier and more general, the standard library containers provide a set of standard functions and types (§16.3.1). For example:

```
template<class T> class vector {
public:
    typedef T value_type;
    typedef T* iterator;

    iterator begin();
    iterator end();

    // ...
};
```

```
template<class T> class list {
    class link {
        // ...
    };
public:
    typedef T value_type;
    typedef link* iterator;

    iterator begin();
    iterator end();

    // ...
};
```

This allows us to write:

```
void f1(vector<T>& v)
{
    vector<T>::iterator i = v.begin();
    // ...
}

void f2(list<T>& v)
{
    list<T>::iterator i = v.begin();
    // ...
}
```

However, this does not allow us to write:

```
template<class C> void f4(C& v)
{
    C::iterator i = v.begin(); // error
    // ...
}
```

Unfortunately, the compiler isn't required to be psychic, so it doesn't know that ***C::iterator*** is the name of a type. In the previous example, the compiler could look at the declaration of ***vector<>*** to determine that the ***iterator*** in ***vector<T>::iterator*** was a type. That is not possible when the qualifier is a type parameter. Naturally, a compiler could postpone all checking until instantiation time where all information is available and could then accept such examples. However, that would be a nonstandard language extension.

Consider an example stripped of clues as to its meaning:

```
template<class T> void f5(T& v)
{
    T::x(y); // error?
}
```

Is ***T::x*** a function called with a nonlocal variable ***y*** as its argument? Or, are we declaring a variable ***y*** with the type ***T::x*** perversely using redundant parentheses? We could imagine a context in which ***X::x(y)*** was a function call and ***Y::x(y)*** was a declaration.

The resolution is simple: unless otherwise stated, an identifier is assumed to refer to something that is not a type or a template. If we want to state that something should be treated as a type, we can do so using the ***typename*** keyword:

```
template<class C> void f4(C& v)
{
    typename C::iterator i = v.begin();
    // ...
}
```

The ***typename*** keyword can be placed in front of a qualified name to state that the entity named is a type. In this, it resembles ***struct*** and ***class***.

The ***typename*** keyword can also be used as an alternative to ***class*** in template declarations. For example:

```
template<typename T> void f(T);
```

Being an indifferent typist and always short of screen space, I prefer the shorter:

```
template<class T> void f(T);
```

C.13.6 Template as a Qualifier

The need for the ***typename*** qualifier arises because we can refer both to members that are types and to members that are non-types. We can also have members that are templates. In rare cases, the need to distinguish the name of a template member from other member names can arise. Consider a possible interface to a general memory manager:

```
class Memory { // some Allocator
public:
    template<class T> T* get_new();
    template<class T> void release(T&);
    // ...
};

template<class Allocator> void f(Allocator& m)
{
    int* p1 = m.get_new<int>();              // syntax error: int after less-than operator
    int* p2 = m.template get_new<int>();     // explicit qualification
    // ...
    m.release(p1); // template argument deduced: no explicit qualification needed
    m.release(p2);
}
```

Explicit qualification of ***get_new()*** is necessary because its template parameter cannot be deduced. In this case, the ***template*** prefix must be used to inform the compiler (and the human reader) that ***get_new*** is a member template so that explicit qualification with the desired type of element is possible. Without the qualification with ***template***, we would get a syntax error because the **<** would be assumed to be a less-than operator. The need for qualification with ***template*** is rare because most template parameters are deduced.

C.13.7 Instantiation

Given a template definition and a use of that template, it is the implementation's job to generate correct code. From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used. From a template function, a function needs to be generated. This process is commonly called *template instantiation*.

The generated classes and functions are called *specializations*. When there is a need to distinguish between generated specializations and specializations explicitly written by the programmer (§13.5), these are referred to as *generated specializations* and *explicit specializations*, respectively. An explicit specialization is sometimes referred to as a *user-defined specialization*, or simply a *user specialization*.

To use templates in nontrivial programs, a programmer must understand how names used in a template definition are bound to declarations and how source code can be organized (§13.7).

By default, the compiler generates classes and functions from the templates used in accordance with the name-binding rules (§C.13.8). That is, a programmer need not state explicitly which versions of which templates must be generated. This is important because it is not easy for a programmer to know exactly which versions of a template are needed. Often, templates that the programmer hasn't even heard of are used in the implementation of libraries, and sometimes templates that the programmer does know of are used with unknown template argument types. In general, the set of generated functions needed can be known only by recursive examination of the templates used in application code libraries. Computers are better suited than humans for doing such analysis.

However, it is sometimes important for a programmer to be able to state specifically where code should be generated from a template (§C.13.10). By doing so, the programmer gains detailed control over the context of the instantiation. In most compilation environments, this also implies control over exactly when that instantiation is done. In particular, explicit instantiation can be used to force compilation errors to occur at predictable times rather than occurring whenever an implementation determines the need to generate a specialization. A perfectly predictable build process is essential to some users.

C.13.8 Name Binding

It is important to define template functions so that they have as few dependencies as possible on nonlocal information. The reason is that a template will be used to generate functions and classes based on unknown types and in unknown contexts. Every subtle context dependency is likely to surface as a debugging problem for some programmer – and that programmer is unlikely to want to know the implementation details of the template. The general rule of avoiding global names as far as possible should be taken especially seriously in template code. Thus, we try to make template definitions as self-contained as possible and to supply much of what would otherwise have been global context in the form of template parameters (e.g., traits; §13.4, §20.2.1).

However, some nonlocal names must be used. In particular, it is more common to write a set of cooperating template functions than to write just one self-contained function. Sometimes, such functions can be class members, but not always. Sometimes, nonlocal functions are the best choice. Typical examples of that are ***sort*( )**'s calls to ***swap*( )** and ***less*( )** (§13.5.2). The standard library algorithms provide a large-scale example (Chapter 18).

Operations with conventional names and semantics, such as +, *, [], and *sort*(), are another source of nonlocal name use in a template definition. Consider:

```
#include<vector>

bool tracing;

// ...

template<class T> T sum(std::vector<T>& v)
{
    T t = 0;
    if (tracing) cerr << "sum(" << &v << ")\n";
    for (int i = 0; i<v.size(); i++) t = t + v[i];
    return t;
}

// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
    Quad c = sum(v);
}
```

The innocent-looking template function *sum*() depends on the + operator. In this example, + is defined in *<quad.h>*:

```
Quad operator+(Quad, Quad);
```

Importantly, nothing related to complex numbers is in scope when *sum*() is defined and the writer of *sum*() cannot be assumed to know about class *Quad*. In particular, the + may be defined later than *sum*() in the program text, and even later in time.

The process of finding the declaration for each name explicitly or implicitly used in a template is called *name binding*. The general problem with template name binding is that three contexts are involved in a template instantiation and they cannot be cleanly separated:

[1] The context of the template definition
[2] The context of the argument type declaration
[3] The context of the use of the template

C.13.8.1 Dependent Names

When defining a function template, we want to assure that enough context is available for the template definition to make sense in terms of its actual arguments without picking up ''accidental'' stuff from the environment of a point of use. To help with this, the language separates names used in a template definition into two categories:

[1] Names that depend on a template argument. Such names are bound at some point of instantiation (§C.13.8.3). In the *sum*() example, the definition of + can be found in the instantiation context because it takes operands of the template argument type.

[2] Names that don't depend on a template argument. Such names are bound at the point of definition of the template (§C.13.8.2). In the *sum*( ) example, the template *vector* is defined in the standard header <*vector*> and the Boolean *tracing* is in scope when the definition of *sum*( ) is encountered by the compiler.

The simplest definition of ''*N* depends on a template parameter *T*'' would be ''*N* is a member of *T*.'' Unfortunately, this doesn't quite suffice; addition of *Quad*s (§C.13.8) is a counter-example. Consequently, a function call is said to *depend on* a template argument if and only if one of these conditions hold:

[1] The type of the actual argument depends on a template parameter *T* according to the type deduction rules (§13.3.1). For example, *f*(*T*(*1*)), *f*(*t*), *f*(*g*(*t*)), and *f*(&*t*), assuming that *t* is a *T*.

[2] The function called has a formal parameter that depends on *T* according to the type deduction rules (§13.3.1). For example, *f*(*T*), *f*(*list*<*T*>&), and *f*(*const T**).

Basically, the name of a function called is dependent if it is obviously dependent by looking at its arguments or at its formal parameters.

A call that by coincidence has an argument that matches an actual template parameter type is not dependent. For example:

```
template<class T> T f(T a)
{
    return g(1);    // error: no g() in scope and g(1) doesn't depend on T
}
void g(int);
int z = f(2);
```

It doesn't matter that for the call *f*(*2*), *T* happens to be *int* and *g*( )'s argument just happens to be an *int*. Had *g*(*1*) been considered dependent, its meaning would have been most subtle and mysterious to the reader of the template definition. If a programmer wants *g*(*int*) to be called, *g*(*int*)'s definition should be placed before the definition of *f*( ) so that *g*(*int*) is in scope when *f*( ) is analyzed. This is exactly the same rule as for non-template function definitions.

Note that only names of functions used in calls can be dependent names according to this definition. Names of variables, class members, types, etc., in a template definition must be declared (possibly in terms of template parameters) before they are used.

C.13.8.2 Point of Definition Binding

When the compiler sees a template definition, it determines which names are dependent (§C.13.8.1). If a name is dependent, looking for its declaration must be postponed until instantiation time (§C.13.8.3).

Names that do not depend on a template argument must be in scope (§4.9.4) at the point of definition. For example:

```
int x;
```

```
template<class T> T f(T a)
{
    x++;        // ok
    y++;        // error: no y in scope, and y doesn't depend on T
    return a;
}

int y;

int z = f(2);
```

If a declaration is found, that declaration is used even if a ‘‘better’’ declaration might be found later. For example:

```
void g(double);

template<class T> class X : public T {
public:
    void f() { g(2); }   // call g(double);
    // ...
};

void g(int);

class Z { };

void h(X<Z> x)
{
    x.f();
}
```

When a definition for `X<Z>::f()` is generated, `g(int)` is not considered because it is declared after `X`. It doesn't matter that `X` is not used until after the declaration of `g(int)`. Also, a call that isn't dependent cannot be hijacked in a base class:

```
class Y { public: void g(int); };

void h(X<Y> x)
{
    x.f();
}
```

Again, `X<Y>::f()` will call `g(double)`. If the programmer had wanted the `g()` from the base class `T` to be called, the definition of `f()` should have said so:

```
template<class T> class XX : public T {
    void f() { T::g(2); }     // calls T::g()
    // ...
};
```

This is, of course, an application of the rule of thumb that a template definition should be as self-contained as possible.

C.13.8.3 Point of Instantiation Binding

Each use of a template for a given set of template arguments defines a point of instantiation. That point is in the nearest global or namespace scope enclosing its use, just before the declaration that contains that use. For example:

```
template<class T> void f(T a) { g(a); }

void g(int);

void h()
{
    extern g(double);
    f(2);
}
```

Here, the point of instantiation for `f<int>()` is just before `h()`, so the `g()` called in `f()` is the global `g(int)` rather than the local `g(double)`. The definition of ''instantiation point'' implies that a template parameter can never be bound to a local name or a class member. For example:

```
void f()
{
    struct X { /* ... */ };        // local structure
    vector<X> v;                   // error: cannot use local structure as template parameter
    // ...
}
```

Nor can an unqualified name used in a template ever be bound to a local name. Finally, even if a template is first used within a class, unqualified names used in the template will not be bound to members of that class. Ignoring local names is essential to prevent a lot of nasty macro-like behavior. For example:

```
template<class T> void sort(vector<T>& v)
{
    sort(v.begin(), v.end());        // use standard library sort()
}

class Container {
    vector<int> v;   // elements
    // ...
public:
    void sort()      // sort elements
    {
        sort(v);     // invokes sort(vector<int>&) rather than Container::sort()
    }
    // ...
};
```

If the point of instantiation for a template defined in a namespace is in another namespace, names from both namespaces are available for name binding. As always, overload resolution is used to choose between names from different namespaces (§8.2.9.2).

Note that a template used several times with the same set of template arguments has several points of instantiation. If the bindings of independent names differ, the program is illegal. However, this is a difficult error for an implementation to detect, especially if the points of instantiation are in different translation units. It is best to avoid subtleties in name binding by minimizing the use of nonlocal names in templates and by using header files to keep use contexts consistent.

C.13.8.4 Templates and Namespaces

When a function is called, its declaration can be found even if it is not in scope, provided it is declared in the same namespace as one of its arguments (§8.2.6). This is very important for functions called in template definitions because it is the mechanism by which dependent functions are found during instantiation.

A template specialization may be generated at any point of instantiation (§C.13.8.3), any point subsequent to that in a translation unit, or in a translation unit specifically created for generating specializations. This reflects three obvious strategies an implementation can use for generating specializations:

[1] Generate a specialization the first time a call is seen.

[2] At the end of a translation unit, generate all specializations needed for that translation unit.

[3] Once every translation unit of a program has been seen, generate all specializations needed for the program.

All three strategies have strengths and weaknesses, and combinations of these strategies are also possible.

In any case, the binding of independent names is done at a point of template definition. The binding of dependent names is done by looking at

[1] the names in scope at the point where the template is defined, plus

[2] the names in the namespace of an argument of a dependent call (global functions are considered in the namespace of built-in types).

For example:

```
namespace N {
    class A { /* ... */ };

    char f(A);
}

char f(int);

template<class T> char g(T t) { return f(t); }

char c = g(N::A());          // causes N::f(N::A) to be called
```

Here, `f(t)` is clearly dependent, so we can't bind `f` to `f(N::A)` or `f(int)` at the point of definition. To generate a specialization for `g<N::A>(N::A)`, the implementation looks in namespace `N` for functions called `f()` and finds `N::f(N::A)`.

A program is illegal, if it is possible to construct two different meanings by choosing different points of instantiation or different contents of namespaces at different possible contexts for generating the specialization. For example:

```
namespace N {
    class A { /* ... */ };

    char f(A, int);
}

template<class T, class T2> char g(T t, T2 t2) { return f(t,t2); }

char c = g(N::A(), 'a');        // error (alternative resolutions of f(t) possible)

namespace N {                   // add to namespace N (§8.2.9.3)
    void f(A, char);
}
```

We could generate the specialization at the point of instantiation and get `f(N::A, int)` called. Alternatively, we could wait and generate the specialization at the end of the translation unit and get `f(N::A, char)` called. Consequently, the call `g(N::A(), 'a')` is an error.

It is sloppy programming to call an overloaded function in between two of its declarations. Looking at a large program, a programmer would have no reason to suspect a problem. In this particular case, a compiler could catch the ambiguity. However, similar problems can occur in separate translation units, and then detection becomes much harder. An implementation is not obliged to catch problems of this kind.

Most problems with alternative resolutions of function calls involve built-in types. Consequently, most remedies rely on more-careful use of arguments of built-in types.

As usual, use of global functions can make matters worse. The global namespace is considered the namespace associated with built-in types, so global functions can be used to resolve dependent calls that take built-in types. For example:

```
int f(int);

template<class T> T g(T t) { return f(t); }

char c = g('a');     // error: alternative resolutions of f(t) are possible

char f(char);
```

We could generate the specialization `g<char>(char)` at the point of instantiation and get `f(int)` called. Alternatively, we could wait and generate the specialization at the end of the translation unit and get `f(char)` called. Consequently, the call `g('a')` is an error.

C.13.9 When Is a Specialization Needed?

It is necessary to generate a specialization of a class template only if the class' definition is needed. In particular, to declare a pointer to some class, the actual definition of a class is not needed. For example:

```
class X;
X* p;       // ok: no definition of X needed
X a;        // error: definition of X needed
```

When defining template classes, this distinction can be crucial. A template class is *not* instantiated unless its definition is actually needed. For example:

```
template<class T> class Link {
    Link* suc; // ok: no definition of Link needed (yet)
    // ...
};

Link<int>* pl;  // no instantiation of Link<int> needed

Link<int> lnk;  // now we need to instantiate Link<int>
```

The point of instantiation is where a definition is first needed.

C.13.9.1 Template Function Instantiation

An implementation instantiates a template function only if that function has been used. In particular, instantiation of a class template does not imply the instantiation of all of its members or even of all of the members defined in the template class declaration. This allows the programmer an important degree of flexibility when defining a template class. Consider:

```
template<class T> class List {
    // ...
    void sort();
};

class Glob { /* no comparison operators */ };

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // use operations on lb, but not lb.sort()
}
```

Here, ***List<string>::sort()*** is instantiated, but ***List<Glob>::sort()*** isn't. This both reduces the amount of code generated and saves us from having to redesign the program. Had ***List<Glob>::sort()*** been generated, we would have had to either add the operations needed by ***vector::sort()*** to ***Glob***, redefine ***sort()*** so that it wasn't a member of ***List***, or use some other container for ***Glob***s.

C.13.10 Explicit Instantiation

An explicit instantiation request is a declaration of a specialization prefixed by the keyword ***template*** (not followed by <):

```
template class vector<int>;                  // class
template int& vector<int>::operator[](int);  // member
template int convert<int,double>(double);    // function
```

A template declaration starts with ***template<***, whereas plain ***template*** starts an instantiation request. Note that ***template*** prefixes a complete declaration; just stating a name is not sufficient:

```
template vector<int>::operator[];   // syntax error
template convert<int,double>;       // syntax error
```

As in template function calls, the template arguments that can be deduced from the function arguments can be omitted (§13.3.1). For example:

```
template int convert<int,double>(double);      // ok (redundant)
template int convert<int>(double);             // ok
```

When a class template is explicitly instantiated, every member function is also instantiated.

Note that an explicit instantiation can be used as a constraints check (§13.6.2). For example:

```
template<class T> class Calls_foo {
    void constraints(T t) { foo(t); }     // call from every constructor
    // ...
};

template class Calls_foo<int>;                 // error: foo(int) undefined
template Calls_foo<Shape*>::constraints();     // error: foo(Shape*) undefined
```

The link-time and recompilation efficiency impact of instantiation requests can be significant. I have seen examples in which bundling most template instantiations into a single compilation unit cut the compile time from a number of hours to the equivalent number of minutes.

It is an error to have two definitions for the same specialization. It does not matter if such multiple specializations are user-defined (§13.5), implicitly generated (§C.13.7), or explicitly requested. However, a compiler is not required to diagnose multiple instantiations in separate compilation units. This allows a smart implementation to ignore redundant instantiations and thereby avoid problems related to composition of programs from libraries using explicit instantiation (§C.13.7). However, implementations are not required to be smart. Users of ''less smart'' implementations must avoid multiple instantiations. However, the worst that will happen if they don't is that their program won't load; there will be no silent changes of meaning.

The language does not require that a user request explicit instantiation. Explicit instantiation is an optional mechanism for optimization and manual control of the compile-and-link process (§C.13.7).

C.14 Advice

[1] Focus on software development rather than technicalities; §C.1.
[2] Adherence to the standard does not guarantee portability; §C.2.
[3] Avoid undefined behavior (including proprietary extensions); §C.2.
[4] Localize implementation-defined behavior; §C.2.
[5] Use keywords and digraphs to represent programs on systems where `{ } [ ] |` are missing and trigraphs if `\` or `!` are missing; §C.3.1.
[6] To ease communication, use the ANSI characters to represent programs; §C.3.3.
[7] Prefer symbolic escape characters to numeric representation of characters; §C.3.2.
[8] Do not rely on signedness or unsignedness of ***char***; §C.3.4.
[9] If in doubt about the type of an integer literal, use a suffix; §C.4.
[10] Avoid value-destroying implicit conversions; §C.6.
[11] Prefer ***vector*** over array; §C.7.
[12] Avoid ***union***s; §C.8.2.

[13] Use fields to represent externally-imposed layouts; §C.8.1.
[14] Be aware of the tradeoffs between different styles of memory management; §C.9.
[15] Don't pollute the global namespace; §C.10.1.
[16] Where a scope (module) rather than a type is needed, prefer a ***namespace*** over a ***class***; §C.10.3.
[17] Remember to define ***static*** class template members; §C.13.1.
[18] Use ***typename*** to disambiguate type members of a template parameter; §C.13.5.
[19] Where explicit qualification by template arguments is necessary, use ***template*** to disambiguate template class members; §C.13.6.
[20] Write template definitions with minimal dependence on their instantiation context; §C.13.8.
[21] If template instantiation takes too long, consider explicit instantiation; §C.13.10.
[22] If the order of compilation needs to be perfectly predictable, consider explicit instantiation; §C.13.10.

Appendix D

Locales

When in Rome,
do as the Romans do.
– proverb

Handling cultural differences — class ***locale*** — named locales — constructing locales — copying and comparing locales — the ***global***( ) and ***classic***( ) locales — comparing strings — class ***facet*** — accessing facets in a locale — a simple user-defined facet — standard facets — string comparison — numeric I/O — money I/O — date and time I/O — low-level time operations — a ***Date*** class — character classification — character code conversion — message catalogs — advice — exercises.

D.1 Handling Cultural Differences

A ***locale*** is an object that represents a set of cultural preferences, such as how strings are compared, the way numbers appear as human-readable output, and the way characters are represented in external storage. The notion of a locale is extensible so that a programmer can add new ***facet***s to a ***locale*** representing locale-specific entities not directly supported by the standard library, such as postal codes (zip codes) and phone numbers. The primary use of ***locale***s in the standard library is to control the appearance of information put to an ***ostream*** and the format accepted by an ***istream***.

Section §21.7 describes how to change ***locale*** for a stream; this appendix describes how a ***locale*** is constructed out of ***facet***s and explains the mechanisms through which a ***locale*** affects its stream. This appendix also describes how ***facet***s are defined, lists the standard ***facet***s that define specific properties of a stream, and presents techniques for implementing and using ***locale***s and ***facet***s. The standard library facilities for representing data and time are discussed as part of the presentation of date I/O.

The discussion of locales and facets is organized like this:

§D.1 introduces the basic ideas for representing cultural differences using locales.
§D.2 presents the ***locale*** class.
§D.3 presents the ***facet*** class.
§D.4 gives an overview of the standard ***facet***s and presents details of each:
- §D.4.1 String comparison
- §D.4.2 Input and output of numeric values
- §D.4.3 Input and output of monetary values
- §D.4.4 Input and output of dates and time
- §D.4.5 Character classification
- §D.4.6 Character code conversions
- §D.4.7 Message catalogs

The notion of a locale is not primarily a C++ notion. Most operating systems and application environments have a notion of locale. Such a notion is – in principle – shared among all programs on a system, independently of which programming language they are written in. Thus, the C++ standard library notion of a locale can be seen as a standard and portable way for C++ programs to access information that has very different representations on different systems. Among other things, a C++ ***locale*** is a common interface to system information that is represented in incompatible ways on different systems.

D.1.1 Programming Cultural Differences

Consider writing a program that needs to be used in several countries. Writing a program in a style that allows that is often called ''internationalization'' (emphasizing the use of a program in many countries) or ''localization'' (emphasizing the adaptation of a program to local conditions). Many of the entities that a program manipulates will conventionally be displayed differently in those countries. We can handle this by writing our I/O routines to take this into account. For example:

```
void print_date(const Date& d) // print in the appropriate format
{
    switch(where_am_I) {      // user-defined style indicator
    case DK:                  // e.g., 7. marts 1999
        cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
        break;
    case UK:                  // e.g., 7 / 3 / 1999
        cout << d.day() << " / " << d.month() << " / " << d.year();
        break;
    case US:                  // e.g., 3/7/1999
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
    // ...
    }
}
```

This style of code does the job. However, it's rather ugly, and we have to use this style consistently to ensure that all output is properly adjusted to local conventions. Worse, if we want to add a new way of writing a date, we must modify the code. We could imagine handling this problem by

creating a class hierarchy (§12.2.4). However, the information in a ***Date*** is independent of the way we want to look at it. Consequently, we don't want a hierarchy of ***Date*** types: for example, ***US_date***, ***UK_date***, and ***JP_date***. Instead, we want a variety of ways of displaying ***Date***s: for example, US-style output, UK-style output, and Japanese-style output; see §D.4.4.5.

Other problems arise with the ''let the user write I/O functions that take care of cultural differences'' approach:

[1] An application programmer cannot easily, portably, and efficiently change the appearance of built-in types without the help of the standard library.

[2] Finding every I/O operation (and every operation that prepares data for I/O in a locale-sensitive manner) in a large program is not always feasible.

[3] Sometimes, we cannot rewrite a program to take care of a new convention – and even if we could, we'd prefer a solution that didn't involve a rewrite.

[4] Having each user design and implement a solution to the problems of different cultural convention is wasteful.

[5] Different programmers will handle low-level cultural preferences in different ways, so programs dealing with the same information will differ for non-fundamental reasons. Thus, programmers maintaining code from a number of sources will have to learn a variety of programming conventions. This is tedious and error prone.

Consequently, the standard library provides an extensible way of handling cultural conventions. The iostreams library (§21.7) relies on this framework to handle both built-in and user-defined types. For example, consider a simple loop copying (***Date***, ***double***) pairs that might represent a series of measurements or a set of transactions:

```
void cpy(istream& is, ostream& os)    // copy (Date,double) stream
{
    Date d;
    double volume;

    while (is >> d >> volume) os << d << ´ ´<< volume << ´\n´;
}
```

Naturally, a real program would do something with the records, and ideally also be a bit more careful about error handling.

How would we make this program read a file that conformed to French conventions (where comma is the character used to represent the decimal point in a floating-point number; for example, ***12,5*** means twelve and a half) and write it according to American conventions? We can define ***locale***s and I/O operations so that ***cpy***( ) can be used to convert between conventions:

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2)
{
    fin.imbue(locale("en_US"));          // American English
    fout.imbue(locale("fr"));            // French
    cpy(fin,fout);                       // read American English, write French

    fin2.imbue(locale("fr"));            // French
    fout2.imbue(locale("en_US"));        // American English
    cpy(fin2,fout2);                     // read French, write American English
}
```

Given streams,

```
Apr 12, 1999  1000.3
Apr 13, 1999  345.45
Apr 14, 1999  9688.321
 ...

3 juillet 1950  10,3
3 juillet 1951  134,45
3 juillet 1952  67,9
 ...
```

this program would produce:

```
12 avril 1999 1000,3
13 avril 1999 345,45
14 avril 1999 9688,321
 ...

July 3, 1950 10.3
July 3, 1951 134.45
July 3, 1952 67.9
 ...
```

Much of the rest of this appendix is devoted to describing the mechanisms that make this possible and explaining how to use them. Please note that most programmers will have little reason to deal with the details of ***locale***s. Many programmers will never explicitly manipulate a ***locale***, and most who do will just retrieve a standard locale and imbue a stream with it (§21.7). However, the mechanisms provided to compose those ***locale***s and to make them trivial to use constitute a little programming language of their own.

If a program or a system is successful, it will be used by people with needs and preferences that the original designers and programmers didn't anticipate. Most successful programs will be run in countries where (natural) languages and character sets differ from those familiar to the original designers and programmers. Wide use of a program is a sign of success, so designing and programming for portability across linguistic and cultural borders is to prepare for success.

The concept of localization (internationalization) is simple. However, practical constraints make the design and implementation of ***locale*** quite intricate:

[1] A ***locale*** encapsulates cultural conventions, such as the appearance of a date. Such conventions vary in many subtle and unsystematic ways. These conventions have nothing to do with programming languages, so a programming language cannot standardize them.

[2] The concept of a ***locale*** must be extensible, because it is not possible to enumerate every cultural convention that is important to every C++ user.

[3] A ***locale*** is used in I/O operations from which people demand run-time efficiency.

[4] A ***locale*** must be invisible to the majority of programmers who want to benefit from stream I/O ''doing the right thing'' without having to know exactly what that is or how it is achieved.

[5] A ***locale*** must be available to designers of facilities that deal with cultural-sensitive information beyond the scope of the stream I/O library.

Designing a program doing I/O requires a choice between controlling formatting through ''ordinary

code'' and the use of ***locale***s. The former (traditional) approach is feasible where we can ensure that every input operation can be easily converted from one convention to another. However, if the appearance of built-in types needs to vary, if different character sets are needed, or if we need to choose among an extensible set of I/O conventions, the ***locale*** mechanism begins to look attractive.

A ***locale*** is composed of ***facet***s that control individual aspects, such as the character used for punctuation in the output of a floating-point value (***decimal_point***( ); §D.4.2) and the format used to read a monetary value (***moneypunct***; §D.4.3). A ***facet*** is an object of a class derived from class ***locale*** : : ***facet*** (§D.3). We can think of a ***locale*** as a container of ***facet***s (§D.2, §D.3.1).

D.2 The *locale* Class

The ***locale*** class and its associated facilities are presented in <***locale***>:

```
class std::locale {
public:
    class facet;                    // type used to represent aspects of a locale; §D.3
    class id;                       // type used to identify a locale; §D.3
    typedef int category;           // type used to group/categorize facets

    static const category           // the actual values are implementation defined
        none = 0,
        collate = 1,
        ctype = 1<<1,
        monetary = 1<<2,
        numeric = 1<<3,
        time = 1<<4,
        messages = 1<<5,
        all = collate | ctype | monetary | numeric | time | messages;

    locale() throw();                   // copy of global locale (§D.2.1)
    locale(const locale& x) throw();    // copy of x
    explicit locale(const char* p);     // copy of locale named p (§D.2.1)

    ~locale() throw();

    locale(const locale& x, const char* p, category c);       // copy of x plus facets from p's c
    locale(const locale& x, const locale& y, category c);     // copy of x plus facets from y's c

    template <class Facet> locale(const locale& x, Facet* f); // copy of x plus facet f
    template <class Facet> locale combine(const locale& x);   // copy of *this plus Facet from x

    const locale& operator=(const locale& x) throw();

    bool operator==(const locale&) const;     // compare locales
    bool operator!=(const locale&) const;

    string name() const;                      // name of this locale (§D.2.1)

    template <class Ch, class Tr, class A>    // compare strings using this locale
    bool operator()(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) const;
```

```
        static locale global(const locale&);        // set global locale and return old global locale
        static const locale& classic();             // get ''classic'' C-style locale
    private:
        // representation
    };
```

A *locale* can be thought of as an interface to a *map<id, facet*>*; that is, something that allows us to use a *locale::id* to find a corresponding object of a class derived from *locale::facet*. A real implementation of *locale* is an efficient variant of this idea. The layout will be something like this:

Here, *collate<char>* and *numpunct<char>* are standard library facets (§D.4). As all facets, they are derived from *locale::facet*.

A *locale* is meant to be copied freely and cheaply. Consequently, a *locale* is almost certainly implemented as a handle to the specialized *map<id, facet*>* that constitutes the main part of its implementation. The *facet*s must be quickly accessible in a *locale*. Consequently, the specialized *map<id, facet*>* will be optimized to provide array-like fast access. The *facet*s of a *locale* are accessed by using the *use_facet<Facet>(loc)* notation; see §D.3.1.

The standard library provides a rich set of *facet*s. To help the programmer manipulate *facet*s in logical groups, the standard *facet*s are grouped into categories, such as *numeric* and *collate* (§D.4).

A programmer can replace *facet*s from existing categories (§D.4, §D.4.2.1). However, it is not possible to add new categories; there is no way for a programmer to define a new category. The notion of ''category'' applies to standard library facets only, and it is not extensible. Thus, a facet need not belong to any category, and many user-defined facets do not.

By far the dominant use of *locale*s is implicitly, in stream I/O. Each *istream* and *ostream* has its own *locale*. The *locale* of a stream is by default the global *locale* (§D.2.1) at the time of the stream's creation. The *locale* of a stream can be set by the *imbue( )* operation and we can extract a copy of a stream's *locale* using *getloc( )* (§21.6.3).

D.2.1 Named Locales

A *locale* is constructed from another *locale* and from *facet*s. The simplest way of making a locale is to copy an existing one. For example:

```
locale loc0;                          // copy of the current global locale (§D.2.3)

locale loc1 = locale();               // copy of the current global locale (§D.2.3)
locale loc2("");                      // copy of ''the user's preferred locale''

locale loc3("C");                     // copy of the "C" locale
locale loc4 = locale::classic();      // copy of the "C" locale

locale loc5("POSIX");                 // copy of the implementation-defined "POSIX" locale
```

The meaning of ***locale***`("C")` is defined by the standard to be the ''classic'' C locale; this is the locale that has been used throughout this book. Other ***locale*** names are implementation defined.

The ***locale***`("")` is deemed to be ''the user's preferred locale.'' This locale is set by extralinguistic means in a program's execution environment.

Most operating systems have ways of setting a locale for a program. Often, a locale suitable to the person using a system is chosen when that person first encounters a system. For example, I would expect a person who configures a system to use Argentine Spanish as its default setting will find ***locale***`("")` to mean ***locale***`("es_AR")`. A quick check on one of my systems revealed 51 locales with mnemonic names, such as ***POSIX***, ***de***, ***en_UK***, ***en_US***, ***es***, ***es_AR***, ***fr***, ***sv***, ***da***, ***pl***, and ***iso_8859_1***. POSIX recommends a format of a lowercase language name, optionally followed by an uppercase country name, optionally followed by an encoding specifier; for example, ***jp_JP.jit***. However, these names are not standardized across platforms. On another system, among many other locale names, I found ***g***, ***uk***, ***us***, ***s***, ***fr***, ***sw***, and ***da***. The C++ standard does not define the meaning of a ***locale*** for a given country or language, though there may be platform-specific standards. Consequently, to use named ***locale***s on a given system, a programmer must refer to system documentation and experiment.

It is generally a good idea to avoid embedding ***locale*** name strings in the program text. Mentioning a file name or a system constant in the program text limits the portability of a program and often forces a programmer who wants to adapt a program to a new environment to find and change such values. Mentioning a locale name string has similar unpleasant consequences. Instead, locales can be picked up from the program's execution environment (for example, using ***locale***`("")`), or the program can request an expert user to specify alternative locales by entering a string. For example:

```
void user_set_locale(const string& question_string)
{
    cout << question_string;    // e.g., "If you want to use a different locale, please enter its name"
    string s;
    cin >> s;
    locale::global(locale(s.c_str()));  // set global locale as specified by user
}
```

It is usually better to let a non-expert user pick from a list of alternatives. A routine for doing this would need to know where and how a system kept its locales.

If the string argument doesn't refer to a defined ***locale***, the constructor throws the ***runtime_error*** exception (§14.10). For example:

```
void set_loc(locale& loc, const char* name)
try
{
    loc = locale(name);
}
catch (runtime_error) {
    cerr << "locale \"" << name << "\" isn´t defined\n";
    // ...
}
```

If a *locale* has a name string, *name*( ) will return it. If not, *name*( ) will return *string*("*"). A name string is primarily a way to refer to a *locale* stored in the execution environment. Secondarily, a name string can be used as a debugging aid. For example:

```
void print_locale_names(const locale& my_loc)
{
    cout << "name of current global locale: " << locale( ).name( ) << "\n";
    cout << "name of classic C locale: " << locale::classic( ).name( ) << "\n";
    cout << "name of ``user´s preferred locale´´: " << locale("").name( ) << "\n";
    cout << "name of my locale: " << my_loc.name( ) << "\n";
}
```

Locales with identical name strings different from the default *string*("*") compare equal. However, == or != provide more direct ways of comparing locales.

The copy of a *locale* with a name string gets the same name as that *locale* (if it has one), so many *locale*s can have the same name string. That's logical because *locale*s are immutable, so all of these objects define the same set of cultural conventions.

A call *locale*(*loc*,"*Foo*",*cat*) makes a locale that is like *loc* except that it takes the facets from the category *cat* of *locale*("*Foo*"). The resulting locale has a name string if and only if *loc* has one. The standard doesn't specify exactly which name string the new locale gets, but it is supposed to be different from *loc*'s. One obvious implementation would be to compose the new string out of *loc*'s name string and "*Foo*". For example, if *loc*'s name string is *en_UK*, the new locale may have "*en_UK*:*Foo*" as its name string.

The name strings for a newly created *locale* can be summarized like this:

Locale	Name String
locale("Foo")	"Foo"
locale(loc)	*loc.name*( )
locale(loc,"Foo",cat)	New name string if *loc* has a name string; otherwise, *string*("*")
locale(loc,loc2,cat)	New name string if *loc* and *loc2* have strings; otherwise, *string*("*")
locale(loc,Facet)	*string*("*")
loc.combine(loc2)	*string*("*")

There are no facilities for a programmer to specify a C-style string as a name for a newly created *locale* in a program. Name strings are either defined in the program's execution environment or created as combinations of such names by *locale* constructors.

D.2.1.1 Constructing New Locales

A new locale is made by taking an existing ***locale*** and adding or replacing ***facet***s. Typically, a new ***locale*** is a minor variation on an existing one. For example:

```
void f(const locale& loc, const My_money_io* mio)  // My_money_io defined in §D.4.3.1
{
    locale loc1(locale("POSIX"), loc, locale::monetary);    // use monetary facets from loc
    locale loc2 = locale(locale::classic(), mio);           // classic plus mio
    // ...
}
```

Here, ***loc1*** is a copy of the ***POSIX*** locale modified to use ***loc***'s monetary facets (§D.4.3). Similarly, ***loc2*** is a copy of the ***C*** locale modified to use a ***My_money_io*** (§D.4.3.1). If a ***Facet**** argument (here, ***My_money_io***) is ***0***, the resulting ***locale*** is simply a copy of the ***locale*** argument.

When using

```
locale(const locale& x, Facet* f);
```

the ***f*** argument must identify a specific facet type. A plain ***facet**** is not sufficient. For example:

```
void g(const locale::facet* mio1, const My_money_io* mio2)
{
    locale loc3 = locale(locale::classic(), mio1);  // error: type of facet not known
    locale loc4 = locale(locale::classic(), mio2);  // ok: type of facet known
    // ...
}
```

The reason is that the ***locale*** uses the type of the ***Facet**** argument to determine the type of the facet at compile time. Specifically, the implementation of ***locale*** uses a facet's identifying type, ***facet::id*** (§D.3), to find that facet in the locale (§D.3.1).

Note that the

```
template <class Facet> locale(const locale& x, Facet* f);
```

constructor is the only mechanism offered within the language for the programmer to supply a ***facet*** to be used through a ***locale***. Other ***locale***s are supplied by implementers as named locales (§D.2.1). These named locales can be retrieved from the program's execution environment. A programmer who understands the implementation-specific mechanism used for that might be able to add new ***locale***s that way (§D.6[11,12]).

The set of constructors for ***locale*** is designed so that the type of every ***facet*** is known either from type deduction (of the ***Facet*** template parameter) or because it came from another ***locale*** (that knew its type). Specifying a ***category*** argument specifies the type of ***facet***s indirectly, because the ***locale*** knows the type of the ***facet***s in the categories. This implies that the ***locale*** class can (and does) keep track of the types of ***facet*** types so that it can manipulate them with minimal overhead.

The ***locale::id*** member type is used by ***locale*** to identify ***facet*** types (§D.3).

It is sometimes useful to construct a ***locale*** that is a copy of another except for a ***facet*** copied from yet another ***locale***. The ***combine()*** template member function does that. For example:

```
void f(const locale& loc, const locale& loc2)
{
    locale loc3 = loc.combine< My_money_io >(loc2);
    // ...
}
```

The resulting *loc3* behaves like *loc* except that it uses a copy of *My_money_io* (§D.4.3.1) from *loc2* to format monetary I/O. If *loc2* doesn't have a *My_money_io* to give to the new *locale*, *combine*( ) will throw a *runtime_error* (§14.10). The result of *combine*( ) has no name string.

D.2.2 Copying and Comparing Locales

A *locale* can be copied by initialization and by assignment. For example:

```
void swap(locale& x, locale& y)        // just like std::swap()
{
    locale temp = x;
    x = y;
    y = temp;
}
```

The copy of a *locale* compares equal to the original, but the copy is an independent and separate object. For example:

```
void f(locale* my_locale)
{
    locale loc = locale::classic();  // "C" locale

    if (loc != locale::classic()) {
        cerr << "implementation error: send bug report to vendor\n";
        exit(1);
    }

    if (&loc != &locale::classic()) cout << "no surprise: addresses differ\n";

    locale loc2 = locale(loc, my_locale, locale::numeric);

    if (loc == loc2) {
        cout << "my numeric facets are the same as classic()´s numeric facets\n";
        // ...
    }

    // ...
}
```

If *my_locale* has a numeric punctuation facet, *my_numpunct<char>*, that is different from *classic*( )'s standard *numpunct<char>*, the resulting *locale*s can be represented like this:

There is no way of modifying a ***locale***. Instead, the ***locale*** operations provide ways of making new ***locales*** from existing ones. The fact that a ***locale*** is immutable after it has been created is essential for run-time efficiency. This allows someone using a ***locale*** to call virtual functions of a ***facet*** and to cache the values returned. For example, an ***istream*** can know what character is used to represent the decimal point and how ***true*** is represented, without calling ***decimal_point***( ) each time it reads a number and ***truename***( ) each time it reads to a ***bool*** (§D.4.2). Only a call of ***imbue***( ) for the stream (§21.6.3) can cause such calls to return a different value.

D.2.3 The *global()* and the *classic()* Locales

The notion of a current locale for a program is provided by ***locale***( ), which yields a copy of the current locale, and ***locale***::***global***(*x*), which sets the current locale to *x*. The current locale is commonly referred to as the ''global locale,'' reflecting its probable implementation as a global (or ***static***) object.

The global locale is implicitly used when a stream is initialized. That is, every new stream is imbued (§21.1, §21.6.3) with a copy of ***locale***( ). Initially, the global locale is the standard C locale, ***locale***::***classic***( ).

The ***locale***::***global***( ) static member function allows a programmer to specify a locale to be used as the global locale. A copy of the previous global locale is returned by ***global***( ). This allows a user to restore the global locale. For example:

```
void f(const locale& my_loc)
{
    ifstream fin1(some_name);                    // fin1 is imbued with the global locale
    locale& old_global = locale::global(my_loc); // set new global locale
    ifstream fin2(some_other_name);              // fin2 is imbued with my_loc
    // ...
    locale::global(old_global);                  // restore old global locale
}
```

If a locale *x* has a name string, *locale*::*global*(*x*) also sets the C global locale. This implies that if a C++ program calls a locale-sensitive function from the C standard library, the treatment of locale will be consistent throughout a mixed C and C++ program.

If a locale *x* does not have a name string, it is undefined whether *locale*::*global*(*x*) affects the C global locale. This implies that a C++ program cannot reliably and portably set the C locale to a locale that wasn't retrieved from the execution environment. There is no standard way for a C program to set the C++ global locale (except by calling a C++ function to do so). In a mixed C and C++ program, having the C global locale differ from *global*() is error prone.

Setting the global locale does not affect existing I/O streams; those still use the locales that they were imbued with before the global locale was reset. For example, *fin1* is unaffected by the manipulation of the global locale that caused *fin2* to be imbued with *my_loc*.

Changing the global locale suffers the same problems as all other techniques relying on changing global data: It is essentially impossible to know what is affected by a change. It is therefore best to reduce use of *global*() to a minimum and to localize those changes in a few sections of code that obey a simple strategy for the changes. The ability to imbue (§21.6.3) individual streams with specific *locale*s makes that easier. However, too many explicit uses of *locale*s and *facet*s scattered throughout a program will also become a maintenance problem.

D.2.4 Comparing Strings

Comparing two strings according to a *locale* is possibly the most common *explicit* use of a *locale*. Consequently, this operation is provided directly by *locale* so that users don't have to build their own comparison function from the *collate* facet (§D.4.1). To be directly useful as a predicate (§18.4.2), this comparison function is defined as *locale*'s *operator*()(). For example:

```
void f(vector<string>& v, const locale& my_locale)
{
    sort(v.begin(), v.end());                  // sort using < to compare elements
    // ...
    sort(v.begin(), v.end(), my_locale);       // sort according to the rules of my_locale
    // ...
}
```

By default, the standard library *sort*() uses < for the numerical value of the implementation character set to determine collation order (§18.7, §18.6.3.1).

Note that *locale*s compare *basic_string*s rather than C-style strings.

D.3 Facets

A *facet* is an object of a class derived from *locale*'s member class *facet*:

```
class std::locale::facet {
protected:
    explicit facet(size_t r = 0);      // "r==0": the locale controls the lifetime of this facet
    virtual ~facet();                  // note: protected destructor
```

```
private:
    facet(const facet&);              // not defined
    void operator=(const facet&);     // not defined

    // representation
};
```

The copy operations are ***private*** and are left undefined to prevent copying (§11.2.2).

The ***facet*** class is designed to be a base class and has no public functions. Its constructor is ***protected*** to prevent the creation of ''plain ***facet***'' objects, and its destructor is virtual to ensure proper destruction of derived-class objects.

A ***facet*** is intended to be managed through pointers by ***locale***s. A ***0*** argument to the ***facet*** constructor means that ***locale*** should delete the ***facet*** when the last reference to it goes away. Conversely, a nonzero constructor argument ensures that ***locale*** never deletes the ***facet***. A nonzero argument is meant for the rare case in which the lifetime of a facet is controlled directly by the programmer rather than indirectly through a locale. For example, we could try to create objects of the standard facet type ***collate_byname<char>*** (§D.4.1.1) like this:

```
void f(const string& s1, const string& s2)
{
    // normal case: (default) argument 0 means that locale is responsible for deletion:
    collate<char>* p = new collate_byname<char>("pl");
    locale loc(locale(),p);

    // rare case: argument 1 means that user is responsible for deletion:
    collate<char>* q = new collate_byname<char>("ge",1);

    collate_byname<char> bug1("sw");        // error: cannot destroy local variable
    collate_byname<char> bug2("no",1);      // error: cannot destroy local variable

    // ...

    // q cannot be deleted: collate_byname<char>'s destructor is protected
    // no delete p; locale manages deletion of *p
}
```

That is, standard facets are useful when managed by locales, as base classes, and only rarely in other ways.

A ***_byname()*** facet is a facet from a named locale in the the execution environment (§D.2.1).

For a ***facet*** to be found in a ***locale*** by ***has_facet()*** and ***use_facet()*** (§D.3.1), each kind of facet must have an ***id***:

```
class std::locale::id {
public:
    id();
private:
    id(const id&);                    // not defined
    void operator=(const id&);        // not defined

    // representation
};
```

The copy operations are declared private and are left undefined to prevent copying (§11.2.2).

The intended use of ***id*** is for the user to define a ***static*** member of type ***id*** of each class supplying a new ***facet*** interface (for example, see §D.4.1). The ***locale*** mechanisms use ***id***s to identify facets (§D.2, §D.3.1). In the obvious implementation of a ***locale***, an ***id*** is used as an index into a vector of pointers to facets, thereby implementing an efficient ***map<id, facet*>***.

Data used to define a (derived) ***facet*** is defined in the derived class rather than in the base class ***facet*** itself. This implies that the programmer defining a ***facet*** has full control over the data and that arbitrary amounts of data can be used to implement the concept represented by a ***facet***.

Note that all member functions of a user-defined ***facet*** should be ***const*** members. Generally, a facet is intended to be immutable (§D.2.2).

D.3.1 Accessing Facets in a Locale

The facets of a ***locale*** are accessed through the template function ***use_facet***, and we can inquire whether a ***locale*** has a specific ***facet***, using the template function ***has_facet***:

```
template <class Facet> bool has_facet(const locale&) throw();
template <class Facet> const Facet& use_facet(const locale&);   // may throw bad_cast
```

Think of these template functions as doing a lookup in their ***locale*** argument for their template parameter ***Facet***. Alternatively, think of ***use_facet*** as a kind of explicit type conversion (cast) of a ***locale*** to a specific ***facet***. This is feasible because a ***locale*** can have only one ***facet*** of a given type. For example:

```
void f(const locale& my_locale)
{
    char c = use_facet< numpunct<char> >(my_locale).decimal_point() // use standard facet
    // ...

    if (has_facet<Encrypt>(my_locale)) {      // does my_locale contain an Encrypt facet?
        const Encrypt& f = use_facet<Encrypt>(my_locale);   // retrieve Encrypt facet
        const Crypto c = f.get_crypto();                    // use Encrypt facet
        // ...
    }
    // ...
}
```

Note that ***use_facet*** returns a reference to a ***const*** facet, so we cannot assign the result of ***use_facet*** to a non-***const***. This makes sense because a facet is meant to be immutable and to have only ***const*** members.

If we try ***use_facet<X>(loc)*** and ***loc*** doesn't have an ***X*** facet, ***use_facet()*** throws ***bad_cast*** (§14.10). The standard ***facet***s are guaranteed to be available for all locales (§D.4), so we don't need to use ***has_facet*** for standard facets. For standard facets, ***use_facet*** will not throw ***bad_cast***.

How might ***use_facet*** and ***has_facet*** be implemented? Remember that we can think of a ***locale*** as a ***map<id, facet*>*** (§D.2). Given a ***facet*** type as the ***Facet*** template argument, the implementation of ***has_facet*** or ***use_facet*** can refer to ***Facet::id*** and use that to find the corresponding facet. A very naive implementation of ***has_facet*** and ***use_facet*** might look like this:

```
// pseudoimplementation: imagine that a locale has a map<id,facet*> called facet_map

template <class Facet> bool has_facet(const locale& loc) throw()
{
    const locale::facet* f = loc.facet_map[Facet::id];
    return f ? true : false;
}

template <class Facet> const Facet& use_facet(const locale& loc)
{
    const locale::facet* f = loc.facet_map[Facet::id];
    if (f) return static_cast<const Facet&>(*f);
    throw bad_cast();
}
```

Another way of looking at the ***facet::id*** mechanism is as an implementation of a form of compile-time polymorphism. A ***dynamic_cast*** can be used to get very similar results to what ***use_facet*** produces. However, the specialized ***use_facet*** can be implemented more efficiently than the more general ***dynamic_cast***.

An ***id*** really identifies an interface and a behavior rather than a class. That is, if two facet classes have exactly the same interface and implement the same semantics (as far as a ***locale*** is concerned), they should be identified by the same ***id***. For example, ***collate<char>*** and ***collate_byname<char>*** are interchangeable in a ***locale***, so both are identified by ***collate<char>::id*** (§D.4.1).

If we define a ***facet*** with a new interface – such as ***Encrypt*** in ***f()*** – we must define a corresponding ***id*** to identify it (see §D.3.2 and §D.4.1).

D.3.2 A Simple User-Defined Facet

The standard library provides standard facets for the most critical areas of cultural differences, such as character sets and I/O of numbers. To examine the facet mechanism in isolation from the complexities of widely used types and the efficiency concerns that accompany them, let me first present a ***facet*** for a trivial user-defined type:

```
enum Season { spring, summer, fall, winter };
```

This was just about the simplest user-defined type I could think of. The style of I/O outlined here can be used with little variation for most simple user-defined types.

```
class Season_io : public locale::facet {
public:
    Season_io(int i = 0) : locale::facet(i) { }

    ~Season_io() { }    // to make it possible to destroy Season_io objects (§D.3)

    virtual const string& to_str(Season x) const = 0;          // string representation of x

    // place Season corresponding to s in x:
    virtual bool from_str(const string& s, Season& x) const = 0;
```

```
        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
    };

    locale::id Season_io::id; // define the identifier object
```

For simplicity, this ***facet*** is limited to representations using ***char***.

The ***Season_io*** class provides a general and abstract interface for all ***Season_io*** facets. To define the I/O representation of a ***Season*** for a particular locale, we derive a class from ***Season_io***, defining ***to_str*( )** and ***from_str*( )** appropriately.

Output of a ***Season*** is easy. If the stream has a ***Season_io*** facet, we can use that to convert the value into a string. If not, we can output the ***int*** value of the ***Season***:

```
    ostream& operator<<(ostream& s, Season x)
    {
        const locale& loc = s.getloc(); // extract the stream's locale (§21.7.1)
        if (has_facet<Season_io>(loc)) return s << use_facet<Season_io>(loc).to_str(x);
        return s << int(x);
    }
```

Note that this << operator is implemented by invoking << for other types. This way, we benefit from the simplicity of using << compared to accessing the ***ostream***'s stream buffers directly, from the locale sensitivity of those << operators, and from the error handling provided for those << operators. Standard ***facet***s tend to operate directly on stream buffers (§D.4.2.2, §D.4.2.3) for maximum efficiency and flexibility, but for many simple user-defined types, there is no need to drop to the ***streambuf*** level of abstraction.

As is typical, input is a bit more complicated than output:

```
    istream& operator>>(istream& s, Season& x)
    {
        const locale& loc = s.getloc();          // extract the stream's locale (§21.7.1)

        if (has_facet<Season_io>(loc)) {      // read alphabetic representation
            const Season_io& f = use_facet<Season_io>(loc);
            string buf;
            if (!(s>>buf && f.from_str(buf,x))) s.setstate(ios_base::failbit);
            return s;
        }

        int i;                // read numeric representation
        s >> i;
        x = Season(i);
        return s;
    }
```

The error handling is simple and follows the error-handling style for built-in types. That is, if the input string didn't represent a ***Season*** in the chosen locale, the stream is put into the ***fail*** state. If exceptions are enabled, this implies that an ***ios_base::failure*** exception is thrown (§21.3.6).

Here is a trivial test program:

```
int main()        // trivial test
{
        Season x;

        // Use default locale (no Season_io facet) implies integer I/O:
        cin >> x;
        cout << x << endl;

        locale loc(locale(), new US_season_io);
        cout.imbue(loc);        // Use locale with Season_io facet
        cin.imbue(loc);         // Use locale with Season_io facet

        cin >> x;
        cout << x << endl;
}
```

Given the input

```
2
summer
```

this program responded:

```
2
summer
```

To get this, we must define ***US_season_io*** to define the string representation of the seasons and override the ***Season_io*** functions that convert between string representations and the enumerators:

```
class US_season_io : public Season_io {
        static const string seasons[];
public:
        const string& to_str(Season) const;
        bool from_str(const string&, Season&) const;

        // note: no US_season_io::id
};

const string US_season_io::seasons[] = { "spring", "summer", "fall", "winter" };

const string& US_season_io::to_str(Season x) const
{
        if (x<spring || winter<x) {
                static const string ss = "no-such-season";
                return ss;
        }
        return seasons[x];
}
```

```
bool US_season_io::from_str(const string& s, Season& x) const
{
    const string* beg = &seasons[spring];
    const string* end = &seasons[winter]+1;
    const string* p = find(beg,end,s);    // §3.8.1, §18.5.2
    if (p==end) return false;
    x = Season(p-beg);
    return true;
}
```

Note that because ***US_season_io*** is simply an implementation of the ***Season_io*** interface, I did not define an ***id*** for ***US_season_io***. In fact, if we want ***US_season_io*** to be used as a ***Season_io***, we must not give ***US_season_io*** its own ***id***. Operations on ***locale***s, such as ***has_facet*** (§D.3.1), rely on facets implementing the same concepts being identified by the same ***id*** (§D.3).

The only interesting implementation question was what to do if asked to output an invalid ***Season***. Naturally, that shouldn't happen. However, it is not uncommon to find an invalid value for a simple user-defined type, so it is realistic to take that possibility into account. I could have thrown an exception, but when dealing with simple output intended for humans to read, it is often helpful to produce an ''out of range'' representation for an out-of-range value. Note that for input, the error-handling policy is left to the >> operator, whereas for output, the facet function ***to_str*( )** implements an error-handling policy. This was done to illustrate the design alternatives. In a ''production design,'' the ***facet*** functions would either implement error handling for both input and output or just report errors for >> and << to handle.

This ***Season_io*** design relied on derived classes to supply the locale-specific strings. An alternative design would have ***Season_io*** itself retrieve those strings from a locale-specific repository (see §D.4.7). The possibility of having a single ***Season_io*** class to which the season strings are passed as constructor arguments is left as an exercise (§D.6[2]).

D.3.3 Uses of Locales and Facets

The primary use of ***locale***s within the standard library is in I/O streams. However, the ***locale*** mechanism is a general and extensible mechanism for representing culture-sensitive information. The ***messages*** facet (§D.4.7) is an example of a facet that has nothing to do with I/O streams. Extensions to the iostreams library and even I/O facilities that are not based on streams might take advantage of locales. Also, a user may use ***locale***s as a convenient way of organizing arbitrary culture-sensitive information.

Because of the generality of the ***locale***/***facet*** mechanism, the possibilities for user-defined ***facet***s are unlimited. Plausible candidates for representation as ***facet***s are dates, time zones, phone numbers, social security numbers (personal identification numbers), product codes, temperatures, general (unit,value) pairs, postal codes (zip codes), clothe sizes, and ISBN numbers.

As with every other powerful mechanism, ***facet***s should be used with care. That something can be represented as a ***facet*** doesn't mean that it is best represented that way. The key issues to consider when selecting a representation for cultural dependencies are – as ever – how the various decisions affect the difficulty of writing code, the ease of reading the resulting code, the maintainability of the resulting program, and the efficiency in time and space of the resulting I/O operations.

D.4 Standard Facets

In *<locale>*, the standard library provides these *facet*s for the *classic*( ) locale:

Standard Facets (in the *classic()* locale)			
	Category	**Purpose**	**Facets**
§D.4.1	*collate*	String comparison	*collate<Ch>*
§D.4.2	*numeric*	Numeric I/O	*numpunct<Ch>* *num_get<Ch>* *num_put<Ch>*
§D.4.3	*monetary*	Money I/O	*moneypunct<Ch>* *moneypunct<Ch,true>* *money_get<Ch>* *money_put<Ch>*
§D.4.4	*time*	Time I/O	*time_get<Ch>* *time_put<Ch>*
§D.4.5	*ctype*	Character classification	*ctype<Ch>* *codecvt<Ch,char,mbstate_t>*
§D.4.7	*messages*	Message retrieval	*messages<Ch>*

In this table, *Ch* is as shorthand for *char* or *wchar_t*. A user who needs standard I/O to deal with another character type *X* must provide suitable versions of facets for *X*. For example, *codecvt<X, char, mbstate_t>* (§D.4.6) might be needed to control conversions between *X* and *char*. The *mbstate_t* type is used to represent the shift states of a multibyte character representation (§D.4.6); *mbstate_t* is defined in *<cwchar>* and *<wchar.h>*. The equivalent to *mbstate_t* for an arbitrary character type *X* is *char_traits<X>::state_type*.

In addition, the standard library provides these *facet*s in *<locale>*:

Standard Facets			
	Category	**Purpose**	**Facets**
§D.4.1	*collate*	String comparison	*collate_byname<Ch>*
§D.4.2	*numeric*	Numeric I/O	*numpunct_byname<Ch>* *num_get<C,In>* *num_put<C,Out>*
§D.4.3	*monetary*	Money I/O	*moneypunct_byname<Ch,International>* *money_get<C,In>* *money_put<C,Out>*
§D.4.4	*time*	Time I/O	*time_put_byname<Ch,Out>*
§D.4.5	*ctype*	Character classification	*ctype_byname<Ch>*
§D.4.7	*messages*	Message retrieval	*messages_byname<Ch>*

When instantiating a facet from this table, *Ch* can be *char* or *wchar_t*; *C* can be any character type (§20.1). *International* can be *true* or *false*; *true* means that a four-character ‘‘international’’

representation of a currency symbol is used (§D.4.3.1). The ***mbstate_t*** type is used to represent the shift states of a multibyte character representation (§D.4.6); ***mbstate_t*** is defined in ***<cwchar>*** and ***<wchar.h>***.

In and ***Out*** are input iterators and output iterators, respectively (§19.1, §19.2.1). Providing the ***_put*** and ***_get*** facets with these template arguments allows a programmer to provide facets that access nonstandard buffers (§D.4.2.2). Buffers associated with iostreams are stream buffers, so the iterators provided for those are ***ostreambuf_iterator***s (§19.2.6.1, §D.4.2.2). Consequently, the function ***failed()*** (§19.2.6.1) is available for error handling.

An ***F_byname*** facet is derived from the facet ***F***. ***F_byname*** provides the identical interface to ***F***, except that it adds a constructor taking a string argument naming a ***locale*** (see §D.4.1). The ***F_byname(name)*** provides the appropriate semantics for ***F*** defined in ***locale(name)***. The idea is to pick a version of a standard facet from a named ***locale*** (§D.2.1) in the program's execution environment. For example:

```
void f(vector<string>& v, const locale& loc)
{
    locale d1(loc,new collate_byname<char>("da"));   // use Danish string comparison
    locale dk(d1,new ctype_byname<char>("da"));      // use Danish character classification
    sort(v.begin(),v.end(),dk);
    // ...
}
```

This new ***dk*** locale will use Danish-style strings but will retain the default conventions for numbers. Note that because ***facet***'s second argument is by default ***0***, the ***locale*** manages the lifetime of a ***facet*** created using operator ***new*** (§D.3).

Like the ***locale*** constructors that take string arguments, the ***_byname*** constructors access the program's execution environment. This implies that they are very slow compared to constructors that do not need to consult the environment. It is almost always faster to construct a locale and then to access its facets than it is to use ***_byname*** facets in many places in a program. Thus, reading a facet from the environment once and then using the copy in main memory repeatedly is usually a good idea. For example:

```
locale dk("da");     // read the Danish locale (incl. all of its facets) once
                     // then use the dk locale and its facets as needed

void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col = use_facet< collate<char> >(dk);
    const collate<char>& ctyp = use_facet< ctype<char> >(dk);

    locale d1(loc,col);  // use Danish string comparison
    locale d2(d1,ctyp);  // use Danish character classification and Danish string comparison

    sort(v.begin(),v.end(),d2);
    // ...
}
```

The notion of categories gives a simpler way of manipulating standard facets in locales. For example, given the ***dk*** locale, we can construct a ***locale*** that reads and compares strings according to the

rules of Danish (that give three extra vowels compared to English) but that retains the syntax of numbers used in C++:

```
locale dk_us(locale::classic(), dk, collate|ctype);    // Danish letters, American numbers
```

The presentations of individual standard ***facet***s contains more examples of ***facet*** use. In particular, the discussion of ***collate*** (§D.4.1) brings out many of the common structural aspects of ***facet***s.

Note that the standard ***facet***s often depend on each other. For example, ***num_put*** depends on ***numpunct***. Only if you have a detailed knowledge of individual ***facet***s can you successfully mix and match facets or add new versions of the standard facets. In other words, beyond the simple operations mentioned in §21.7, the ***locale*** mechanisms are not meant to be directly used by novices.

The design of an individual facet is often messy. The reason is partially that facets have to reflect messy cultural conventions outside the control of the library designer, and partially that the C++ standard library facilities have to remain largely compatible with what is offered by the C standard library and various platform-specific standards. For example, POSIX provides locale facilities that it would be unwise for a library designer to ignore.

On the other hand, the framework provided by locales and facets is very general and flexible. A facet can be designed to hold any data, and the facet's operations can provide any desired operation based on that data. If the behavior of a new facet isn't overconstrained by convention, its design can be simple and clean (§D.3.2).

D.4.1 String Comparison

The standard ***collate*** facet provides ways of comparing arrays of characters of type ***Ch***:

```
template <class Ch>
class std::collate : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit collate(size_t r = 0);

    int compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const
        { return do_compare(b,e,b2,e2); }

    long hash(const Ch* b, const Ch* e) const { return do_hash(b,e); }
    string_type transform(const Ch* b, const Ch* e) const { return do_transform(b,e); }

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~collate();    // note: protected destructor

    virtual int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    virtual string_type do_transform(const Ch* b, const Ch* e) const;
    virtual long do_hash(const Ch* b, const Ch* e) const;
};
```

Like all facets, ***collate*** is publically derived from ***facet*** and provides a constructor that takes an argument that tells whether class ***locale*** controls the lifetime of the facet (§D.3).

Note that the destructor is protected. The *collate* facet isn't meant to be used directly. Rather, it is intended as a base class for all (derived) collation classes and for *locale* to manage (§D.3). Application programmers, implementation providers, and library vendors will write the string comparison facets to be used through the interface provided by *collate*.

The *compare*( ) function does the basic string comparison according to the rules defined for a particular *collate*; it returns *1* if the first string is lexicographically greater than the second, *0* if the strings are identical, and *-1* if the second string is greater than the first. For example:

```
void f(const string& s1, const string& s2, collate<char>& cmp)
{
    const char* cs1 = s1.data();    // because compare() operates on char[]s
    const char* cs2 = s2.data();
    switch (cmp.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size()) {
    case 0:         // identical strings according to cmp
        // ...
        break;
    case -1:        // s1 < s2
        // ...
        break;
    case 1:         // s1 > s2
        // ...
        break;
    }
}
```

Note that the *collate* member functions compare arrays of *Ch* rather than *basic_string*s or zero-terminated C-style strings. In particular, a *Ch* with the numeric value *0* is treated as an ordinary character rather than as a terminator. Also, *compare*( ) differs from *strcmp*( ), returning exactly the values *-1*, *0*, and *1* rather than simply *0* and (arbitrary) negative and positive values (§20.4.1).

The standard library *string* is not *locale* sensitive. That is, it compares strings according to the rules of the implementation's character set (§C.2). Furthermore, the standard *string* does not provide a direct way of specifying comparison criteria (Chapter 20). To do a *locale*-sensitive comparison, we can use a *collate*'s *compare*( ). Notationally, it can be more convenient to use *collate*'s *compare*( ) indirectly through a *locale*'s *operator*( ) (§D.2.4). For example:

```
void f(const string& s1, const string& s2, const char* n)
{
    bool b = s1 == s2;              // compare using implementation's character set values

    const char* cs1 = s1.data();    // because compare() operates on char[]s
    const char* cs2 = s2.data();

    typedef collate<char> Col;

    const Col& glob = use_facet<Col>(locale());         // from the current global locale
    int i0 = glob.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

    const Col& my_coll = use_facet<Col>(locale("")); // from my preferred locale
    int i1 = my_coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());
```

```
        const Col& coll = use_facet<Col>(locale(n));              // from locale named n
        int i2 = coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

        int i3 = locale()(s1,s2);          // compare using the current global locale
        int i4 = locale("")(s1,s2);        // compare using my preferred locale
        int i5 = locale(n)(s1,s2);         // compare using the locale named n

        // ...
    }
```

Here, *i0==i3*, *i1==i4*, and *i2==i5*. It is not difficult to imagine cases in which *i2*, *i3*, and *i4* differ. Consider this sequence of words from a German dictionary:

Dialekt, Diät, dich, dichten, Dichtung

According to convention, nouns (only) are capitalized, but the ordering is not case sensitive.

A case-sensitive German sort would place all words starting with *D* before *d*:

Dialekt, Diät, Dichtung, dich, dichten

The *ä* (umlaut *a*) is treated as ''a kind of *a*,'' so it comes before *c*. However, in most common character sets, the numeric value of *ä* is larger than the numeric value of *c*. Consequently, *int(´c´)<int(´ä´)*, and the simple default sort based on numeric values gives:

Dialekt, Dichtung, Diät, dich, dichten

Writing a compare function that orders this sequence correctly according to the dictionary is an interesting exercise (§D.6[3]).

The *hash()* function calculates a hash value (§17.6.2.3). Obviously, this can be useful for building hash tables.

The *transform()* function produces a string that, when compared to other strings, gives the same result as would a comparison to the argument string. The purpose of *transform()* is to allow optimization of code in which one string is compared to many others. This is useful when implementing a search for one or more strings among a set of strings.

The public *compare()*, *hash()*, and *transform()* functions are implemented by calls to the protected virtual functions *do_compare()*, *do_hash()*, and *do_transform()*, respectively. These ''*do_* functions'' can be overridden in derived classes. This two-function strategy allows the library implementer who writes the non-virtual functions to provide some common functionality for all calls independently of what the user-supplied *do_* function might do.

The use of virtual functions preserves the polymorphic nature of the *facet* but could be costly. To avoid excess function calls, a *locale* can determine the exact *facet* used and cache any values it might need for efficient execution (§D.2.2).

The static member *id* of type *locale::id* is used to identify a *facet* (§D.3). The standard functions *has_facet* and *use_facet* depend on the correspondence between *id*s and *facet*s (§D.3.1). Two *facet*s providing exactly the same interface and semantics to *locale* should have the same *id*. For example, *collate<char>* and *collate_byname<char>* (§D.4.1.1) have the same *id*. Conversely, two *facet*s performing different functions (as far as *locale* is concerned) must have different *id*s. For example, *numpunct<char>* and *num_put<char>* have different *id*s (§D.4.2).

D.4.1.1 Named Collate

A *collate_byname* is a facet that provides a version of *collate* for a particular locale named by a constructor string argument:

```
template <class Ch>
class std::collate_byname : public collate<Ch> {
public:
    typedef basic_string<Ch> string_type;

    explicit collate_byname(const char*, size_t r = 0);   // construct from named locale

    // note: no id and no new functions

protected:
    ~collate_byname();   // note: protected destructor

    // override collate<Ch>'s virtual functions:

    int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    string_type do_transform(const Ch* b, const Ch* e) const;
    long do_hash(const Ch* b, const Ch* e) const;
};
```

Thus, a *collate_byname* can be used to pick out a *collate* from a locale named in the program's execution environment (§D.4). One obvious way of storing facets in an execution environment would be as data in a file. A less flexible alternative would be to represent a facet as program text and data in a *_byname* facet.

The *collate_byname<char>* class is an example of a *facet* that doesn't have its own *id* (§D.3). In a *locale*, *collate_byname<Ch>* is interchangeable with *collate<Ch>*. A *collate* and a *collate_byname* for the same locale differ only in the extra constructor offered by the *collate_byname* and in the semantics provided by the *collate_byname*.

Note that the *_byname* destructor is *protected*. This implies that you cannot have a *_byname* facet as a local variable. For example:

```
void f()
{
    collate_byname<char> my_coll("");   // error: cannot destroy my_coll
    // ...
}
```

This reflects the view that using locales and facets is something that is best done at a fairly high level in a program to affect large bodies of code. An example is setting the global locale (§D.2.3) or imbuing a stream (§21.6.3, §D.1). If necessary, we could derive a class with a public destructor from a *_byname* class and create local variables of that class.

D.4.2 Numeric Input and Output

Numeric output is done by a *num_put* facet writing into a stream buffer (§21.6.4). Conversely, numeric input is done by a *num_get* facet reading from a stream buffer. The format used by *num_put* and *num_get* is defined by a ''numerical punctuation'' facet, *numpunct*.

D.4.2.1 Numeric Punctuation

The ***numpunct*** facet defines the I/O format of built-in types, such as ***bool***, ***int***, and ***double***:

```
template <class Ch>
class std::numpunct : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit numpunct(size_t r = 0);

    Ch decimal_point() const;          // '.' in classic()
    Ch thousands_sep() const;          // ',' in classic()
    string grouping() const;           // "" in classic(), meaning no grouping

    string_type truename() const;      // "true" in classic()
    string_type falsename() const;     // "false" in classic()

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~numpunct();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The characters of the string returned by ***grouping***( ) are read as a sequence of small integer values. Each number specifies a number of digits for a group. Character 0 specifies the rightmost group (the least-significant digits), character 1 the group to the left of that, etc. Thus, "*\004\002\003*" describes a number, such as *123-45-6789* (provided you use ´-´ as the separation character). If necessary, the last number in a grouping pattern is used repeatedly, so "*\003*" is equivalent to "*\003\003\003*". As the name of the separation character, ***thousands_sep***( ), indicates, the most common use of grouping is to make large integers more readable. The ***grouping***( ) and ***thousands_sep***( ) functions define a format for both input and output of integers. They are not used for standard floating-point number I/O. Thus, we can not get *1234567.89* printed as *1,234,567.89* simply by defining ***grouping***( ) and ***thousands_sep***( ).

We can define a new punctuation style by deriving from ***numpunct***. For example, I could define facet ***My_punct*** to write integer values using spaces to group the digits by threes and floating-point values, using a European-style comma as the ''decimal point:''

```
class My_punct : public std::numpunct<char> {
public:
    typedef char char_type;
    typedef string string_type;

    explicit My_punct(size_t r = 0) : std::numpunct<char>(r) { }

protected:
    char do_decimal_point() const { return ´,´; }  // comma
    char do_thousands_sep() const { return ´ ´; }  // space
    string do_grouping() const { return "\003"; }  // 3-digit groups
};
```

```
void f()
{
    cout << "style A: " << 12345678 << " *** "<< 1234567.8 << '\n';

    locale loc(locale(), new My_punct);
    cout.imbue(loc);
    cout << "style B: " << 12345678 << " *** "<< 1234567.8 << '\n';
}
```

This produced:

```
style A: 12345678 *** 1.23457e+06
style B: 12 345 678 *** 1,23457e+06
```

Note that ***imbue*()** stores a copy of its argument in its stream. Consequently, a stream can rely on an imbued locale even after the original copy of that locale has been destroyed. If an iostream has its ***boolalpha*** flag set (§21.2.2, §21.4.1), the strings returned by ***truename*()** and ***falsename*()** are used to represent ***true*** and ***false***, respectively; otherwise, ***1*** and ***0*** are used.

A ***_byname*** version (§D.4, §D.4.1) of ***numpunct*** is provided:

```
template <class Ch>
class std::numpunct_byname : public numpunct<Ch> { /* ... */ };
```

D.4.2.2 Numeric Output

When writing to a stream buffer (§21.6.4), an ***ostream*** relies on the ***num_put*** facet:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::num_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit num_put(size_t r = 0);

    // put value "v" to buffer position "b" in stream "s":
    Out put(Out b, ios_base& s, Ch fill, bool v) const;
    Out put(Out b, ios_base& s, Ch fill, long v) const;
    Out put(Out b, ios_base& s, Ch fill, unsigned long v) const;
    Out put(Out b, ios_base& s, Ch fill, double v) const;
    Out put(Out b, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, ios_base& s, Ch fill, const void* v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~num_put();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The output iterator (§19.1, §19.2.1) argument, ***Out***, identifies where in an ***ostream***'s stream buffer (§21.6.4) ***put*()** places characters representing the numeric value on output. The value of ***put*()** is

that iterator positioned one past the last character position written.

Note that the default specialization of ***num_put*** (the one where the iterator used to access characters is of type ***ostreambuf_iterator<Ch>***) is part of the standard locales (§D.4). If you want to use another specialization, you'll have to make it yourself. For example:

```
template<class Ch>
class String_numput : public std::num_put<Ch,typename basic_string<Ch>::iterator> {
public:
    String_numput() : std::num_put<Ch,typename basic_string<Ch>::iterator>(1) { }
};

void f(int i, string& s, int pos)        // format i into s starting at pos
{
    String_numput<char> f;
    ios_base& xxx = cout;                    // use cout's formatting rules
    f.put(s.begin()+pos,xxx,' ',i);          // format i into s
}
```

The ***ios_base*** argument is used to get information about formatting state and locale. For example, if padding is needed, the ***fill*** character is used as required by the ***ios_base*** argument. Typically, the stream buffer written to through ***b*** is the buffer associated with an ***ostream*** for which ***s*** is the base. Note that an ***ios_base*** is not a simple object to construct. In particular, it controls many aspects of formatting that must be consistent to achieve acceptable output. Consequently, ***ios_base*** has no public constructor (§21.3.3).

A ***put*()** function also uses its ***ios_base*** argument to get the stream's ***locale*()**. That ***locale*** is used to determine punctuation (§D.4.2.1), the alphabetic representation of Booleans, and the conversion to ***Ch***. For example, assuming that ***s*** is ***put*()**'s ***ios_base*** argument, we might find code like this in a ***put*()** function:

```
const locale& loc = s.getloc();
// ...
wchar_t w = use_facet< ctype<char> >(loc).widen(c);                  // char to Ch conversion
// ...
string pnt = use_facet< numpunct<char> >(loc).decimal_point();       // default: '.'
// ...
string flse = use_facet< numpunct<char> >(loc).falsename();          // default: "false"
```

A standard facet, such as ***num_put<char>***, is typically used implicitly through a standard I/O stream function. Consequently, most programmers need not know about it. However, the use of such facets by standard library functions is interesting because they show how I/O streams work and how facets can be used. As ever, the standard library provides examples of interesting programming techniques.

Using ***num_put***, the implementer of ***ostream*** might write:

```
template<class Ch, class Tr>
ostream& std::basic_ostream<Ch,Tr>::operator<<(double d)
{
    sentry guard(*this);        // see §21.3.8
    if (!guard) return *this;
```

```
        try {
            if (use_facet< num_put<Ch> >(getloc()).put(*this, *this, this->fill(), d).failed())
                setstate(badbit);
        }
        catch (...) {
            handle_ioexception(*this);
        }
        return *this;
    }
```

A lot is going on here. The sentry ensures that all prefix and suffix operations are performed (§21.3.8). We get the ***ostream***'s ***locale*** by calling its member function ***getloc*()** (§21.7). We extract ***num_put*** from that ***locale*** using ***use_facet*** (§D.3.1). That done, we call the appropriate ***put*()** function to do the real work. An ***ostreambuf_iterator*** can be constructed from an ***ostream*** (§19.2.6), and an ***ostream*** can be implicitly converted to its base class ***ios_base*** (§21.2.1), so the two first arguments to ***put*()** are easily supplied.

A call of ***put*()** returns its output iterator argument. This output iterator is obtained from a ***basic_ostream***, so it is an ***ostreambuf_iterator***. Consequently, ***failed*()** (§19.2.6.1) is available to test for failure and to allow us to set the stream state appropriately.

I did not use ***has_facet***, because the standard facets (§D.4) are guaranteed to be present in every locale. If that guarantee is violated, ***bad_cast*** is thrown (§D.3.1).

The ***put*()** function calls the virtual ***do_put*()**. Consequently, user-defined code may be executed, and ***operator<<*()** must be prepared to handle an exception thrown by the overriding ***do_put*()**. Also, ***num_put*** may not exist for some character types, so ***use_facet*()** might throw ***std*::*bad_cast*** (§D.3.1). The behavior of a << for a built-in type, such as ***double***, is defined by the C++ standard. Consequently, the question is not what ***handle_ioexception*()** should do but rather how it should do what the standard prescribes. If ***badbit*** is set in this ***ostream***'s exception state (§21.3.6), the exception is simply rethrown. Otherwise, an exception is handled by setting the stream state and continuing. In either case, ***badbit*** must be set in the stream state (§21.3.3):

```
template<class Ch, class Tr>
void handle_ioexception(std::basic_ostream<Ch,Tr>& s)    // called from catch clause
{
    if (s.exceptions()&ios_base::badbit) {
        try {
            s.setstate(ios_base::badbit);    // might throw basic_ios::failure
        } catch(...) { }
        throw;    // rethrow
    }
    s.setstate(ios_base::badbit);
}
```

The *try-block* is needed because ***setstate*()** might throw ***basic_ios*::*failure*** (§21.3.3, §21.3.6). However, if ***badbit*** is set in the exception state, ***operator<<*()** must rethrow the exception that caused ***handle_ioexception*()** to be called (rather than simply throwing ***basic_ios*::*failure***).

The << for a built-in type, such as ***double***, must be implemented by writing directly to a stream buffer. When writing a << for a user-defined type, we can often avoid the resulting complexity by expressing the output of the user-defined type in terms of output of existing types (§D.3.2).

D.4.2.3 Numeric Input

When reading from a stream buffer (§21.6.4), an ***istream*** relies on the ***num_get*** facet:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::num_get : public locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit num_get(size_t r = 0);

    // read [b:e) into v, using formatting rules from s, reporting errors by setting r:
    In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~num_get();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

Basically, ***num_get*** is organized like ***num_put*** (§D.4.2.2). Since it reads rather than writes, ***get*( )** needs a pair of input iterators, and the argument designating the target of the read is a reference.

The ***iostate*** variable ***r*** is set to reflect the state of the stream. If a value of the desired type could not be read, ***failbit*** is set in ***r***; if the end of input was reached, ***eofbit*** is set in ***r***. An input operator will use ***r*** to determine how to set the state of its stream. If no error was encountered, the value read is assigned though ***v***; otherwise, ***v*** is left unchanged.

A sentry is used to ensure that the stream's prefix and suffix operations are performed (§21.3.8). In particular, the sentry is used to ensure that we try to read only if the stream is in a good state to start with.

The implementer of ***istream*** might write:

```
template<class Ch, class Tr>
istream& std::basic_istream<Ch,Tr>::operator>>(double& d)
{
    sentry guard(*this);        // see §21.3.8
    if (!guard) return *this;

    iostate state = 0;      // good
    istreambuf_iterator<Ch> eos;
    double dd;
```

```
        try {
            use_facet< num_get<Ch> >(getloc()).get(*this, eos, *this, state, dd);
            if (state==0 || state==eofbit) d = dd; // set value only if get() succeeded
            setstate(state);
        }
        catch (...) {
            handle_ioexception(*this);        // see §D.4.2.2
        }
        return *this;
    }
```

Exceptions enabled for the ***istream*** will be thrown by ***setstate***( ) in case of error (§21.3.6).

By defining a ***numpunct***, such as ***My_punct*** from §D.4.2, we can read using nonstandard punctuation. For example:

```
    void f()
    {
        cout << "style A: "
        int i1;
        double d1;
        cin >> i1 >> d1;            // read using standard ''12345678'' format

        locale loc(locale::classic(), new My_punct);
        cin.imbue(loc);
        cout << "style B: "
        int i2;
        double d2;
        cin >> i1 >> d2;            // read using the ''12 345 678'' format
    }
```

If we want to read really unusual numeric formats, we have to override ***do_get***( ). For example, we might define a ***num_get*** that read Roman numerals, such as ***XXI*** and ***MM*** (§D.6[15]).

D.4.3 Input and Output of Monetary Values

The formatting of monetary amounts is technically similar to the formatting of ''plain'' numbers (§D.4.2). However, the presentation of monetary amounts is even more sensitive to cultural differences. For example, a negative amount (a loss, a debit), such as ***-1.25***, should in some contexts be presented as a (positive) number in parentheses: ***(1.25)***. Similarly, color is in some contexts used to ease the recognition of negative amounts.

There is no standard ''money type.'' Instead, the money facets are meant to be used explicitly for numeric values that the programmer knows to represent monetary amounts. For example:

```
    class Money { // simple type to hold a monetary amount
        long int amount;
    public:
        Money(long int i) : amount(i) { }
        operator long int() const { return amount; }
    };
```

```
// ...

void f(long int i)
{
    cout << "value= " << i << " amount= " << Money(i) << endl;
}
```

The task of the monetary facets is to make it reasonably easy to write an output operator for ***Money*** so that the amount is printed according to local convention (see §D.4.3.2). The output would vary depending on ***cout***'s locale. Possible outputs are:

```
value= 1234567 amount= $12345.67
value= 1234567 amount= 12345,67 DKK
value= -1234567 amount= $-12345.67
value= -1234567 amount= -$12345.67
value= -1234567 amount= (CHF12345,67)
```

For money, accuracy to the smallest currency unit is usually considered essential. Consequently, I adopted the common convention of having the integer value represent the number of cents (pence, øre, fils, cents, etc.) rather than the number of dollars (pounds, kroner, dinar, euro, etc.). This convention is supported by ***moneypunct***'s ***frac_digits***( ) function (§D.4.3.1). Similarly, the appearance of the ''decimal point'' is defined by ***decimal_point***( ).

The facets ***money_get*** and ***money_put*** provide functions that perform I/O based on the format defined by the ***money_base*** facet.

A simple ***Money*** type can be used simply to control I/O formats or to hold monetary values. In the former case, we cast values of (other) types used to hold monetary amounts to ***Money*** before writing, and we read into ***Money*** variables before converting them to other types. It is less error prone to consistently hold monetary amounts in a ***Money*** type; that way, we cannot forget to cast a value to ***Money*** before writing it, and we don't get input errors by trying to read monetary values in locale-insensitive ways. However, it may be infeasible to introduce a ***Money*** type into a system that wasn't designed for that. In such cases, applying ***Money*** conversions (casts) to read and write operations is necessary.

D.4.3.1 Money Punctuation

The facet controlling the presentation of monetary amounts, ***moneypunct***, naturally resembles the facet for controlling plain numbers, ***numpunct*** (§D.4.2.1):

```
class std::money_base {
public:
    enum part { none, space, symbol, sign, value };     // parts of value layout
    struct pattern { char field[4]; };                  // layout specification
};
```

```
template <class Ch, bool International = false>
class std::moneypunct : public locale::facet, public money_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit moneypunct(size_t r = 0);

    Ch decimal_point() const;          // '.' in classic()
    Ch thousands_sep() const;          // ',' in classic()
    string grouping() const;           // "" in classic(), meaning "no grouping"

    string_type curr_symbol() const;   // "$" in classic()
    string_type positive_sign() const; // "" in classic()
    string_type negative_sign() const; // "-" in classic()

    int frac_digits() const;           // number of digits after the decimal point; 2 in classic()
    pattern pos_format() const;        // { symbol, sign, none, value } in classic()
    pattern neg_format() const;        // { symbol, sign, none, value } in classic()

    static const bool intl = International;     // use international monetary formats

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~moneypunct();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The facilities offered by ***moneypunct*** are intended primarily for use by implementers of ***money_put*** and ***money_get*** facets (§D.4.3.2, §D.4.3.3).

The ***decimal_point***( ), ***thousands_sep***( ), and ***grouping***( ) members behave as their equivalents in ***numpunct***.

The ***curr_symbol***( ), ***positive_sign***( ), and ***negative_sign***( ) members return the string to be used to represent the currency symbol (for example, $, ¥, ***FRF***, ***DKK***), the plus sign, and the minus sign, respectively. If the ***International*** template argument was ***true***, the ***intl*** member will also be ***true***, and ''international'' representations of the currency symbols will be used. Such an ''international'' representation is a four-character string. For example:

```
"USD "
"DKK "
"EUR "
```

The last character is a terminating zero. The three-letter currency identifier is defined by the ISO-4217 standard. When ***International*** is ***false***, a ''local'' currency symbol, such as $, £, and ¥, can be used.

A ***pattern*** returned by ***pos_format***( ) or ***neg_format***( ) is four ***part***s defining the sequence in which the numeric value, the currency symbol, the sign symbol, and whitespace occur. Most common formats are trivially represented using this simple notion of a pattern. For example:

```
+$ 123.45       // { sign, symbol, space, value } where positive_sign() returns "+"
$+123.45        // { symbol, sign, value, none } where positive_sign() returns "+"
$123.45         // { symbol, sign, value, none } where positive_sign() returns ""
$123.45-        // { symbol, value, sign, none }
-123.45 DKK     // { sign, value, space, symbol }
($123.45)       // { sign, symbol, value, none } where negative_sign() returns "()"
(123.45DKK)     // { sign, value, symbol, none } where negative_sign() returns "()"
```

Representing a negative number using parentheses is achieved by having ***negative_sign***( ) return a string containing the two characters ( ). The first character of a sign string is placed where ***sign*** is found in the pattern, and the rest of the sign string is placed after all other parts of the pattern. The most common use of this facility is to represent the financial community's convention of using parentheses for negative amounts, but other uses are possible. For example:

```
-$123.45            // { sign, symbol, value, none } where negative_sign() returns "–"
*$123.45 silly      // { sign, symbol, value, none } where negative_sign() returns "* silly"
```

The values ***sign***, ***value***, and ***symbol*** must each appear exactly once in a pattern. The remaining value can be either ***space*** or ***none***. Where ***space*** appears, at least one and possibly more white-space characters may appear in the representation. Where ***none*** appears, except at the end of a pattern, zero or more whitespace characters may appear in the representation.

Note that these strict rules ban some apparently reasonable patterns:

```
pattern pat = { sign, value, none, none };      // error: no symbol
```

The ***frac_digits***( ) function indicates where the ***decimal_point***( ) is placed. Often, monetary amounts are represented in the smallest currency unit (§D.4.3). This unit is typically one hundredth of the major unit (for example, a ¢ is one hundredth of a $), so ***frac_digits***( ) is often *2*.

Here is a simple format defined as a facet:

```
class My_money_io : public moneypunct<char,true> {
public:
    explicit My_money_io(size_t r = 0) :moneypunct<char,true>(r) { }

    char_type do_decimal_point() const { return '.'; }
    char_type do_thousands_sep() const { return ','; }
    string do_grouping() const { return "\003\003\003"; }

    string_type do_curr_symbol() const { return "USD "; }
    string_type do_positive_sign() const { return ""; }
    string_type do_negative_sign() const { return "()"; }

    int do_frac_digits() const { return 2; }    // two digits after decimal point

    pattern do_pos_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
```

```
        pattern do_neg_format() const
        {
            static pattern pat = { sign, symbol, value, none };
            return pat;
        }
    };
```

This facet is used in the ***Money*** input and output operations defined in §D.4.3.2 and §D.4.3.3.

A ***_byname*** version (§D.4, §D.4.1) of ***moneypunct*** is provided:

```
template <class Ch, bool Intl = false>
class std::moneypunct_byname : public moneypunct<Ch, Intl> { /* ... */ };
```

D.4.3.2 Money Output

The ***money_put*** facet writes monetary amounts according to the format specified by ***moneypunct***. Specifically, ***money_put*** provides ***put()*** functions that place a suitably formatted character representation into the stream buffer of a stream:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::money_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_put(size_t r = 0);

    // put value "v" into buffer position "b":
    Out put(Out b, bool intl, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, bool intl, ios_base& s, Ch fill, const string_type& v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~money_put();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The ***b***, ***s***, ***fill***, and ***v*** arguments are used as for ***num_put***'s ***put()*** functions (§D.4.2.2). The ***intl*** argument indicates whether a standard four-character ''international'' currency symbol or a ''local'' symbol is used (§D.4.3.1).

Given ***money_put***, we can define an output operator for ***Money*** (§D.4.3):

```
ostream& operator<<(ostream& s, Money m)
{
    ostream::sentry guard(s);          // see §21.3.8
    if (!guard) return s;
```

```
        try {
            const money_put<char>& f = use_facet< money_put<char> >(s.getloc());
            if (m==static_cast<long double>(m)) { // m can be represented as a long double
                if (f.put(s,true,s,s.fill(),m).failed()) s.setstate(ios_base::badbit);
            }
            else {
                ostringstream v;
                v << m;          // convert to string representation
                if (f.put(s,true,s,s.fill(),v.str()).failed()) s.setstate(ios_base::badbit);
            }
        }
        catch (...) {
            handle_ioexception(s);      // see §D.4.2.2
        }
        return s;
    }
```

If a ***long double*** doesn't have sufficient precision to represent the monetary value exactly, I convert the value to its string representation and output that using the ***put*()** that takes a ***string***.

D.4.3.3 Money Input

The ***money_get*** facet reads monetary amounts according to the format specified by ***moneypunct***. Specifically, ***money_get*** provides ***get*()** functions that extract a suitably formatted character representation from the stream buffer of a stream:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::money_get : public locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_get(size_t r = 0);

    // read [b:e) into v, using formatting rules from s, reporting errors by setting r:
    In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
protected:
    ~money_get();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The ***b***, ***e***, ***s***, ***fill***, and ***v*** arguments are used as for ***num_get***'s ***get*()** functions (§D.4.2.3). The ***intl*** argument indicates whether a standard four-character ''international'' currency symbol or a ''local'' symbol is used (§D.4.3.1).

A well-defined pair of ***money_get*** and ***money_put*** facets will provide output in a form that can be read back in without errors or loss of information. For example:

```
int main()
{
    Money m;
    while (cin>>m) cout << m << "\n";
}
```

The output of this simple program should be acceptable as its input. Furthermore, the output produced by a second run given the output from a first run should be identical to its input.

A plausible input operator for ***Money*** would be:

```
istream& operator>>(istream& s, Money& m)
{
    istream::sentry guard(s);        // see §21.3.8
    if (guard) try {
        ios_base::iostate state = 0;      // good
        istreambuf_iterator<char> eos;
        string str;

        use_facet< money_get<char> >(s.getloc()).get(s,eos,true,state,str);

        if (state==0 || state==ios_base::eofbit) { // set value only if get() succeeded
            long int i = strtol(str.c_str(),0,0); // for strtol(), see §20.4.1
            if (errno==ERANGE)
                state |= ios_base::failbit;
            else
                m = i;      // set value only if conversion to long int succeeded
            s.setstate(state);
        }
    }
    catch (...) {
        handle_ioexception(s);      // see §D.4.2.2
    }
    return s;
}
```

I use the ***get*()** that reads into a ***string*** because reading into a ***double*** and then converting to a ***long int*** could lead to loss of precision.

D.4.4 Date and Time Input and Output

Unfortunately, the C++ standard library does not provide a proper ***date*** type. However, from the C standard library, it inherits low-level facilities for dealing with dates and time intervals. These C facilities are the basis for C++'s facilities for dealing with time in a system-independent manner.

The following sections demonstrate how the presentation of date and time-of-day information can be made ***locale*** sensitive. In addition, they provide an example of how a user-defined type (***Date***) can fit into the framework provided by ***iostream*** (Chapter 21) and ***locale*** (§D.2). The implementation of ***Date*** shows techniques that are useful for dealing with time if you don't have a ***Date*** type available.

D.4.4.1 Clocks and Timers

At the lowest level, most systems have a fine-grained timer. The standard library provides a function *clock*( ) that returns an implementation-defined arithmetic type *clock_t*. The result of *clock*( ) can be calibrated by using the *CLOCKS_PER_SEC* macro. If you don't have access to a reliable timing utility, you might measure a loop like this:

```
int main(int argc, char* argv[])     // §6.1.7
{
    int n = atoi(argv[1]);           // §20.4.1

    clock_t t1 = clock();
    if (t1 == clock_t(-1)) {         // clock_t(-1) means "clock() didn't work"
        cerr << "sorry, no clock\n";
        exit(1);
    }

    for (int i = 0; i<n; i++) do_something(); // timing loop

    clock_t t2 = clock();
    if (t2 == clock_t(-1)) {
        cerr << "sorry, clock overflow\n";
        exit(2);
    }
    cout << "do_something( ) " << n << " times took "
        << double(t2-t1)/CLOCKS_PER_SEC << " seconds"
        << " (measurement granularity: " << CLOCKS_PER_SEC << " of a second)\n";
}
```

The explicit conversion *double*(*t2-t1*) before dividing is necessary because *clock_t* might be an integer. Exactly when the *clock*( ) starts running is implementation defined; *clock*( ) is meant to measure time intervals within a single run of a program. For values *t1* and *t2* returned by *clock*( ), *double*(*t2-t1*)/*CLOCKS_PER_SEC* is the system's best approximation of the time in seconds between the two calls.

If *clock*( ) isn't provided for a processor or if a time interval was too long to measure, *clock*( ) returns *clock_t*(*-1*).

The *clock*( ) function is meant to measure intervals from a fraction of a second to a few seconds. For example, if *clock_t* is a 32-bit signed *int* and *CLOCKS_PER_SEC* is 1,000,000 , we can use *clock*( ) to measure from 0 to just over 2,000 seconds (about half an hour) in microseconds.

Please note that getting meaningful measurements of a program can be tricky. Other programs running on a machine may severely affect the time used by a run, cache and pipelining effects are difficult to predict, and algorithms may have surprising dependencies on data. If you try to time something, make several runs and reject the results as flawed if the run times vary significantly.

To cope with longer time intervals and with calendar time, the standard library provides *time_t* for representing a point in time and a structure *tm* for separating a point in time into its conventional parts:

```
typedef implementation_defined time_t;      // implementation-defined arithmetic type (§4.1.1)
                                            // capable of representing a period of time,
                                            // often, a 32-bit integer

struct tm {
    int tm_sec;     // second of minute [0,61]; 60 and 61 to represent leap seconds
    int tm_min;     // minute of hour [0,59]
    int tm_hour;    // hour of day [0,23]
    int tm_mday;    // day of month [1,31]
    int tm_mon;     // month of year [0,11]; 0 means January (note: not [1:12])
    int tm_year;    // year since 1900; 0 means year 1900, and 102 means 2002
    int tm_wday;    // days since Sunday [0,6]; 0 means Sunday
    int tm_yday;    // days since January 1 [0,365]; 0 means January 1
    int tm_isdst;   // hours of daylight savings time
};
```

Note that the standard guarantees only that ***tm*** has the ***int*** members mentioned here. The standard does not guarantee that the members appear in this order or that there are no other fields.

The ***time_t*** and ***tm*** types and the basic facilities for using them are presented in ***<ctime>*** and ***<time.h>***. For example::

```
clock_t clock();                            // number of clock ticks since the start of the program

time_t time(time_t* pt);                    // current calendar time
double difftime(time_t t2, time_t t1);      // t2–t1 in seconds

tm* localtime(const time_t* pt);            // local time for the *pt
tm* gmtime(const time_t* pt);               // Grenwich Mean Time (GMT) tm for *pt, or 0
                                            // (officially called Coordinated Universal Time, UTC)

time_t mktime(tm* ptm);                     // time_t for *ptm, or time_t(-1)

char* asctime(const tm* ptm);               // C-style string representation for *ptm
                                            // for example, "Sun Sep 16 01:03:52 1973\n"
char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

Beware: both ***localtime( )*** and ***gmtime( )*** return a ***tm**** to a statically allocated object; a subsequent call of that function will change the value of that object. Either use such a return value immediately, or copy the ***tm*** into storage that you control. Similarly, ***asctime( )*** returns a pointer to a statically allocated character array.

A ***tm*** can represent dates in a range of at least tens of thousands of years (about [-32000,32000] for a minimally sized ***int***). However, ***time_t*** is most often a (signed) 32-bit ***long int***. Counting seconds, this makes ***time_t*** capable of representing a range just over 68 years on each side of a base year. This base year is most commonly 1970, with the exact base time being 0:00 of January 1 GMT (UTC). If ***time_t*** is a 32-bit signed integer, we'll run out of ''time'' in 2038 unless we upgrade ***time_t*** to a larger integer type, as is already done on some systems.

The ***time_t*** mechanism is meant primarily for representing ''near current time.'' Thus, we should not expect ***time_t*** to be able to represent dates outside the [1902,2038] range. Worse, not all implementations of the functions dealing with time handle negative values in the same way. For portability, a value that needs to be represented as both a ***tm*** and a ***time_t*** should be in the

[1970,2038] range. People who want to represent dates outside the 1970 to 2038 time frame must devise some additional mechanism to do so.

One consequence of this is that *mktime*( ) can fail. If the argument for *mktime*( ) cannot be represented as a *time_t*, the error indicator *time_t*(-1) is returned.

If we have a long-running program, we might time it like this:

```
int main(int argc, char* argv[])    // §6.1.7
{
    time_t t1 = time(0);
    do_a_lot(argc,argv);
    time_t t2 = time(0);
    double d = difftime(t2,t1);
    cout << "do_a_lot( ) took" << d << " seconds\n";
}
```

If the argument to *time*( ) is not *0*, the resulting time is also assigned to the *time_t* pointed to. If the calendar time is not available (say, on a specialized processor), the value *time_t*(-1) is returned. We could cautiously try to find today's date like this:

```
int main( )
{
    time_t t;

    if (time(&t) == time_t(-1)) {   // time_t(–1) means ''time() didn't work''
        cerr << "Bad time\n";
        exit(1);
    }

    tm* gt = gmtime(&t);
    cout << gt->tm_mon+1 << ´/´ << gt->tm_mday << ´/´ << 1900+gt->tm_year << endl;
}
```

D.4.4.2 A Date Class

As mentioned in §10.3, it is unlikely that a single *Date* type can serve all purposes. The uses of date information dictate a variety of representations, and calendar information before the 19th century is very dependent on historical vagaries. However, as an example, we could define a *Date* type along the lines from §10.3, using *time_t* as the implementation:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Bad_date {};

    Date(int dd, Month mm, int yy);
    Date( );

    friend ostream& operator<<(ostream& s, const Date& d);
```

```
        // ...
    private:
        time_t d;   // standard date and time representation
    };

    Date::Date(int dd, Month mm, int yy)
    {
        tm x = { 0 };
        if (dd<0 || 31<dd) throw Bad_date();     // oversimplified: see §10.3.1
        x.tm_mday = dd;
        if (mm<jan || dec<mm) throw Bad_date();
        x.tm_mon = mm-1;          // tm_mon is zero based
        x.tm_year = yy-1900;      // tm_year is 1900 based
        d = mktime(&x);
    }

    Date::Date()
    {
        d = time(0);     // default Date: today
        if (d == time_t(-1)) throw Bad_date();
    }
```

The task here is to define locale-sensitive implementations for *Date* << and >>.

D.4.4.3 Date and Time Output

Like *num_put* (§D.4.2), *time_put* provides *put*( ) functions for writing to buffers through iterators:

```
    template <class Ch, class Out = ostreambuf_iterator<Ch> >
    class std::time_put : public locale::facet {
    public:
        typedef Ch char_type;
        typedef Out iter_type;

        explicit time_put(size_t r = 0);

        // put t into s's stream buffer through b, using format fmt:
        Out put(Out b, ios_base& s, Ch fill, const tm* t,
                                        const Ch* fmt_b, const Ch* fmt_e) const;

        Out put(Out b, ios_base& s, Ch fill, const tm* t, char fmt, char mod = 0) const
            { return do_put(b,s,fill,t,fmt,mod); }

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~time_put();

        virtual Out do_put(Out, ios_base&, Ch, const tm*, char, char) const;
    };
```

A call *put*(*b*,*s*,*fill*,*t*,*fmt_b*,*fmt_e*) places the date information from *t* into *s*'s stream buffer through *b*. The *fill* character is used where needed for padding. The output format is specified by a *printf*( )-like format string [*fmt_b*,*fmt_e*). The *printf*-like (§21.8) format is used to produce an

actual output and may contain the following special-purpose format specifiers:

%a	abbreviated weekday name (e.g., Sat)
%A	full weekday name (e.g., Saturday)
%b	abbreviated month name (e.g., Feb)
%B	full month name (e.g., February)
%c	date and time (e.g., Sat Feb 06 21:46:05 1999)
%d	day of month [01,31] (e.g., 06)
%H	24-hour clock hour [00,23] (e.g., 21)
%I	12-hour clock hour [01,12] (e.g., 09)
%j	day of year [001,366] (e.g., 037)
%m	month of year [01,12] (e.g., 02)
%M	minute of hour [00,59] (e.g., 48)
%p	a.m./p.m. indicator for 12-hour clock (e.g., PM)
%S	second of minute [00,61] (e.g., 40)
%U	week of year [00,53] starting with Sunday (e.g., 05); the first Sunday starts week 1
%w	day of week [0,6]; 0 means Sunday (e.g., 6)
%W	week of year [00,53] starting with Monday (e.g., 05); the first Monday starts week 1
%x	date (e.g., 02/06/99)
%X	time (e.g., 21:48:40)
%y	year without century [00,99] (e.g., 99)
%Y	year (e.g., 1999)
%Z	time zone indicator (e.g., EST) if the time zone is known

This long list of very specialized formatting rules could be used as an argument for the use of extensible I/O systems. However, as with most specialized notations, it is adequate for its task and often even convenient.

In addition to these formatting directives, most implementations support ‘‘modifiers,’’ such as an integer specifying a field width (§21.8), ***%10X***. Modifiers for the time-and-date formats are not part of the C++ standard, but some platform standards, such as POSIX, require them. Consequently, modifiers can be difficult to avoid even if their use isn't perfectly portable.

The ***sprintf***-like (§21.8) function ***strftime*( )** from ***<ctime>*** or ***<time.h>*** produces output using the time and date format directives:

```
size_t strftime(char* s, size_t max, const char* format, const tm* tmp);
```

This function places a maximum of ***max*** characters from ****tmp*** and the ***format*** into ****s*** according the ***format***. For example:

```
int main()
{
    const int max = 20;  // sloppy: hope strftime() will never produce more than 20 characters
    char buf[max];
    time_t t = time(0);
    strftime(buf, max, "%A\n", localtime(&t));
    cout << buf;
}
```

On a Wednesday, this will print *Wednesday* in the default *classic*( ) locale (§D.2.3) and *onsdag* in a Danish locale.

Characters that are not part of a format specified, such as the newline in the example, are simply copied into the first argument (*s*).

When *put*( ) identifies a format character *f* (and optional modifier character *m*), it calls the virtual *do_put*( ) to do the actual formatting: *do_put*(*b*, *s*, *fill*, *t*, *f*, *m*).

A call *put*(*b*, *s*, *fill*, *t*, *f*, *m*) is a simplified form of *put*( ), where a format character (*f*) and a modifier character (*m*) are explicitly provided. Thus,

```
const char fmt[] = "%10X";
put(b,s,fill,t,fmt,fmt+sizeof(fmt));
```

can be abbreviated to

```
put(b,s,fill,t,´X´,10);
```

If a format contains multibyte characters, it must both begin and end in the default state (§D.4.6).

We can use *put*( ) to implement a *locale*-sensitive output operator for *Date*:

```
ostream& operator<<(ostream& s, const Date& d)
{
        ostream::sentry guard(s);              // see §21.3.8
        if (!guard) return s;

        tm* tmp = localtime(&d.d);
        try {
                if (use_facet< time_put<char> >(s.getloc()).put(s,s,s.fill(),tmp,´x´).failed())
                        s.setstate(ios_base::failbit);
        }
        catch (...) {
                handle_ioexception(s);      // see §D.4.2.2
        }
        return s;
}
```

Since there is no standard *Date* type, there is no default layout for date I/O. Here, I specified the *%x* format by passing the character *´x´* as the format character. Because the *%x* format is the default for *get_time*( ) (§D.4.4.4), that is probably as close to a standard as one can get. See §D.4.4.5 for an example of how to use alternative formats.

A *_byname* version (§D.4, §D.4.1) of *time_put* is also provided:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put_byname : public time_put<Ch,Out> { /* ... */ };
```

D.4.4.4 Date and Time Input

As ever, input is trickier than output. When we write code to output a value, we often have a choice among different formats. In addition, when we write input code, we must deal with errors and sometimes the possibility of several alternative formats.

The *time_get* facet implements input of time and date. The idea is that *time_get* of a *locale* can

read the times and dates produced by the ***locale***'s ***time_put***. However, there are no standard ***date*** and ***time*** classes, so a programmer can use a locale to produce output according to a variety of formats. For example, the following representations could all be produced by using a single output statement, using ***time_put*** (§D.4.4.5) from different locales:

```
January 15th 1999
Thursday 15th January 1999
15 Jan 1999AD
Thurs 15/1/99
```

The C++ standard encourages implementers of ***time_get*** to accept dates and time formats as specified by POSIX and other standards. The problem is that it is difficult to standardize the intent to read dates and times in whatever format is conventional in a given culture. It is wise to experiment to see what a given locale provides (§D.6[8]). If a format isn't accepted, a programmer can provide a suitable alternative ***time_get*** facet.

The standard time input ***facet, time_get***, is derived from ***time_base***:

```
struct std::time_base {
    enum dateorder {
        no_order, // no order, possibly more elements (such as day of week)
        dmy,      // day before month before year
        mdy,      // month before day before year
        ymd,      // year before month before day
        ydm       // year before day before month
    };
};
```

An implementer can use this enumeration to simplify the parsing on date formats.

Like ***num_get***, ***time_get*** accesses its buffer through a pair of input iterators:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class time_get : public locale::facet, public time_base {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit time_get(size_t r = 0);

    dateorder date_order() const { return do_date_order(); }

    // read [b,e) into d, using formatting rules from s, reporting errors by setting r:
    In get_time(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_year(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_weekday(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_monthname(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
protected:
    ~time_get();
    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The *get_time*( ) function calls *do_get_time*( ). The default *get_time*( ) reads time as produced by the *locale*'s *time_put*::*put*( ), using the %*X* format (§D.4.4). Similarly, the *get_date*( ) function calls *do_get_date*( ). The default *get_date*( ) reads a date as produced by the *locale*'s *time_put*::*put*( ), using the %*x* format (§D.4.4).

Thus, the simplest input operator for *Date*s is something like this:

```
istream& operator>>(istream& s, Date& d)
{
        istream::sentry guard(s);          // see §21.3.8
        if (!guard) return s;

        ios_base::iostate res = 0;
        tm x = { 0 };
        istreambuf_iterator<char,char_traits<char> > end;
        try {
                use_facet< time_get<char> >(s.getloc()).get_date(s,end,s,res,&x);
                if (res==0 || res==ios_base::eofbit)
                        d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900);
                else
                        s.setstate(res);
        }
        catch (...) {
                handle_ioexception(s);     // see §D.4.2.2
        }

        return s;
}
```

The call *get_date*(*s*,*end*,*s*,*res*,&*x*) relies on two implicit conversions from *istream*: As the first argument, *s* is used to construct an *istreambuf_iterator*. As third argument, *s* is converted to the *istream* base class *ios_base*.

This input operator will work correctly for dates in the range that can be represented by *time_t*.

A trivial test case would be:

```
int main()
try {
        Date today;
        cout << today << endl;             // write using %x format
        Date d(12, Date::may, 1998);

        cout << d << endl;
        Date dd;
        while (cin >> dd) cout << dd << endl;      // read dates produced by %x format
}
catch (Date::Bad_date) {
        cout << "exit: bad date caught\n";
}
```

A *_byname* version (§D.4, §D.4.1) of *time_get* is also provided:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::time_get_byname : public time_get<Ch,In> { /* ... */ };
```

D.4.4.5 A More Flexible Date Class

If you tried to use the ***Date*** class from §D.4.4.2 with the I/O from §D.4.4.3 and §D.4.4.4, you'd soon find it restrictive:

[1] It can handle only dates that can be represented by a ***time_t***; that typically means in the [1970,2038] range.
[2] It accepts dates only in the standard format – whatever that might be.
[3] Its reporting of input errors is unacceptable.
[4] It supports only streams of ***char*** – not streams of arbitrary character types.

A more interesting and more useful input operator would accept a wider range of dates, recognize a few common formats, and reliably report errors in a useful form. To do this, we must depart from the ***time_t*** representation:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    struct Bad_date {
        const char* why;
        Bad_date(const char* p) : why(p) { }
    };

    Date(int dd, Month mm, int yy, int day_of_week = 0);
    Date();

    void make_tm(tm* t) const;      // place tm representation of Date in *t
    time_t make_time_t() const;     // return time_t representation of Date

    int year() const { return y; }
    Month month() const { return m; }
    int day() const { return d; }

    // ...
private:
    char d;
    Month m;
    int y;
};
```

For simplicity, I reverted to the (d,m,y) representation (§10.2).

The constructor might be defined like this:

```
Date::Date(int dd, Month mm, int yy, int day_of_week)
    :d(dd), m(mm), y(yy)
{
    if (d==0 && m==Month(0) && y==0) return;     // Date(0,0,0) is the "null date"
    if (mm<jan || dec<mm) throw Bad_date("bad month");
```

```
        if (dd<1 || 31<dd)   // oversimplified; see §10.3.1
            throw Bad_date("bad day of month");
        if (day_of_week && day_in_week(yy,mm,dd)!=day_of_week)
            throw Bad_date("bad day of week");
}

Date::Date() :d(0), m(0), y(0) { } // a "null date"
```

The ***day_in_week*()** calculation is nontrivial and immaterial to the ***locale*** mechanisms, so I have left it out. If you need one, your system will have one somewhere.

Comparison operations are always useful for types such as ***Date***:

```
bool operator==(const Date& x, const Date& y)
{
        return x.year()==y.year() && x.month()==y.month() && x.day()==y.day();
}

bool operator!=(const Date& x, const Date& y)
{
        return !(x==y);
}
```

Having departed from the standard ***tm*** and ***time_t*** formats, we need conversion functions to cooperate with software that expects those types:

```
void Date::make_tm(tm* p) const      // put date into *p
{
        tm x = { 0 };
        *p = x;
        p->tm_year = y-1900;
        p->tm_mday = d;
        p->tm_mon = m-1;
}

time_t Date::make_time_t() const
{
        if (y<1970 || 2038<y)      // oversimplified
            throw Bad_date("date out of range for time_t");
        tm x;
        make_tm(&x);
        return mktime(&x);
}
```

D.4.4.6 Specifying a *Date* Format

C++ doesn't define a standard output format for dates (%***x*** is as close as we get; §D.4.4.3). However, even if a standard format existed, we would probably want to be able to use alternatives. This could be done by providing a ''default format'' and a way of changing it. For example:

```
class Date_format {
    static char fmt[];          // default format
    const char* curr;           // current format
    const char* curr_end;
public:
    Date_format() :curr(fmt), curr_end(fmt+strlen(fmt)) { }

    const char* begin() const { return curr; }
    const char* end() const { return curr_end; }

    void set(const char* p, const char* q) { curr=p; curr_end=q; }
    void set(const char* p) { curr=p; curr_end=curr+strlen(p); }

    static const char* default_fmt() { return fmt; }
};

const char Date_format::fmt[] = "%A, %B %d, %Y";  // e.g., Friday, February 5, 1999

Date_format date_fmt;
```

To be able to use that ***strftime*()** format (§D.4.4.3), I have refrained from parameterizing the ***Date_format*** class on the character type used. This implies that this solution allows only date notations for which the format can be expressed as a ***char*[]**. I also used a global format object (***date_fmt***) to provide a default ***Date*** format. Since the value of ***date_fmt*** can be changed, this provides a crude way of controlling ***Date*** formatting, similar to the way ***global*()** (§D.2.3) can be used to control formatting.

A more general solution is to add ***Date_in*** and ***Date_out*** facets to control reading and writing from a stream. That approach is presented in §D.4.4.7.

Given ***Date_format***, ***Date*::*operator*<<()** can be written like this:

```
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>& s, const Date& d)
// write according to user-specified format
{
    typename basic_ostream<Ch,Tr>::sentry guard(s);  // see §21.3.8
    if (!guard) return s;

    tm t;
    d.make_tm(&t);
    try {
        const time_put<Ch>& f = use_facet< time_put<Ch> >(s.getloc());
        if (f.put(s,s,s.fill(),&t,date_fmt.begin(),date_fmt.end()).failed())
            s.setstate(ios_base::failbit);
    }
    catch (...) {
        handle_ioexception(s);      // see §D.4.2.2
    }
    return s;
}
```

I could have used ***has_facet*** to verify that ***s***'s locale had a ***time_put<Ch>*** facet. However, here it seemed simpler to handle that problem by catching any exception thrown by ***use_facet*.**

Here is a simple test program that controls the output format through *date_fmt*:

```
int main()
try {

    while (cin >> dd && dd != Date()) cout << dd << endl;      // write using default date_fmt

    date_fmt.set("%Y/%m/%d");

    while (cin >> dd && dd != Date()) cout << dd << endl;      // write using "%Y/%m/%d"
}
catch (Date::Bad_date e) {
    cout << "bad date caught: " << e.why << endl;
}
```

D.4.4.7 A *Date* Input Facet

As ever, input is a bit more difficult than output. However, because the interface to low-level input is fixed by *get_date*() and because the *operator*>>() defined for *Date* in §D.4.4.4 didn't directly access the representation of a *Date*, we could use that *operator*>>() unchanged. Here is a templatized version to match the *operator*<<() from §D.4.4.6:

```
template<class Ch, class Tr>
istream<Ch,Tr>& operator>>(istream<Ch,Tr>& s, Date& d)
{
    typename istream<Ch,Tr>::sentry guard(s);
    if (guard) try {
        ios_base::iostate res = 0;
        tm x = { 0 };
        istreambuf_iterator<Ch,Tr> end;

        use_facet< time_get<Ch> >(s.getloc()).get_date(s,end,s,res,&x);

        if (res==0 || res==ios_base::eofbit)
            d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900,x.tm_wday);
        else
            s.setstate(res);
    }
    catch (...) {
        handle_ioexception(s);      // see §D.4.2.2
    }
    return s;
}
```

This *Date* input operator calls *get_date*() from the *istream*'s *time_get* facet (§D.4.4.4). Therefore, we can provide a different and more flexible form of input by defining a new facet derived from *time_get*:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class Date_in : public std::time_get<Ch,In> {
public:
    Date_in(size_t r = 0) : std::time_get<Ch>(r) { }
```

```
protected:
    In do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const;

private:
    enum Vtype { novalue, unknown, dayofweek, month };
    In getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const;
};
```

The *getval*( ) needs to read a year, a month, a day of the month, and optionally a day of the week and compose the result into a *tm*.

The names of the months and the names of the days of the week are locale specific. Consequently, we can't mention them directly in our input function. Instead, we recognize months and days by calling the functions that *time_get* provides for that: *get_monthname*( ) and *get_weekday*( ) (§D.4.4.4).

The year, the day of the month, and possibly the month are represented as integers. Unfortunately, a number does not indicate whether it denotes a day or a month, or whatever. For example, *7* could denote July, day 7 of a month, or even the year 2007. The real purpose of *time_get*'s *date_order*( ) is to resolve such ambiguities.

The strategy of *Date_in* is to read values, classify them, and then use *date_order*( ) to see whether (or how) the values entered make sense. The private *getval*( ) function does the actual reading from the stream buffer and the initial classification:

```
template<class Ch, class In>
In Date_in<Ch,In>::getval(In b, In e,
                          ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const
    // read part of Date: number, day_of_week, or month. Skip whitespace and punctuation.
{
    const ctype<Ch>& ct = use_facet< ctype<Ch> >(s.getloc());  // ctype is defined in §D.4.5
    Ch c;

    *res = novalue; // no value found

    for (;;) {// skip whitespace and punctuation
        if (b == e) return e;
        c = *b;
        if (!(ct.is(ctype_base::space,c) || ct.is(ctype_base::punct,c))) break;
        ++b;
    }

    if (ct.is(ctype_base::digit,c)) {    // read integer without regard for numpunct
        int i = 0;


        do { // turn digit from arbitrary character set into decimal value:
            static char const digits[] = "0123456789";
            i = i*10 + find(digits,digits+10,ct.narrow(c,´´))-digits;
            c = *++b;
        } while (ct.is(ctype_base::digit,c));
```

```
            *v = i;
            *res = unknown;        // an integer, but we don't know what it represents
            return b;
        }

        if (ct.is(ctype_base::alpha, c)) {   // look for name of month or day of week
            basic_string<Ch> str;
            while (ct.is(ctype_base::alpha, c)) {       // read characters into string
                str += c;
                if (++b == e) break;
                c = *b;
            }

            tm t;
            basic_stringstream<Ch> ss(str);
            typedef istreambuf_iterator<Ch> SI;          // iterator type for ss' buffer
            get_monthname(ss.rdbuf(), SI(), s, r, &t); // read from in-memory stream buffer

            if ((r&(ios_base::badbit|ios_base::failbit))==0) {
                *v= t.tm_mon;
                *res = month;
                return b;
            }

            r = 0;       // clear state before trying to read a second time
            get_weekday(ss.rdbuf(), SI(), s, r, &t);     // read from in-memory stream buffer

            if ((r&(ios_base::badbit|ios_base::failbit))==0) {
                *v = t.tm_wday;
                *res = dayofweek;
                return b;
            }
        }

        r |= ios_base::failbit;
        return b;
    }
```

The tricky part here is to distinguish months from weekdays. We read through input iterators, so we cannot read `[b,e)` twice, looking first for a month and then for a day. On the other hand, we cannot look at one character at a time and decide, because only ***get_monthname()*** and ***get_weekday()*** know which character sequences make up the names of the months and the names of the days of the week in a given locale. The solution I chose was to read strings of alphabetic characters into a ***string***, make a ***stringstream*** from that string, and then repeatedly read from that stream's ***streambuf***.

The error recording uses the state bits, such as ***ios_base::badbit***, directly. This is necessary because the more convenient functions for manipulating stream state, such as ***clear()*** and ***setstate()***, are defined in ***basic_ios*** rather than in its base ***ios_base*** (§21.3.3). If necessary, the >> operator then uses the error results reported by ***get_date()*** to reset the state of the input stream.

Given ***getval()***, we can read values first and then try to see whether they make sense later. The ***date_order()*** can be crucial:

```
template<class Ch, class In>
In Date_in<Ch,In>::do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const
// optional day of week followed by ymd, dmy, mdy, or ydm
{
    int val[3];                         // for day, month, and year values in some order
    Vtype res[3] = { novalue };         // for value classifications

    for (int i=0; b!=e && i<3; ++i) {   // read day, month, and year
        b = getval(b,e,s,r,&val[i],&res[i]);
        if (r) return b;                // oops: error
        if (res[i]==novalue) {          // couldn't complete date
            r |= ios_base::badbit;
            return b;
        }
        if (res[i]==dayofweek) {
            tmp->tm_wday = val[i];
            --i; // oops: not a day, month, or year
        }
    }

    time_base::dateorder order = date_order();    // now try to make sense of the values read

    if (res[0] == month) {              // mdy or error
        // ...
    }
    else if (res[1] == month) {         // dmy or ymd or error
        tmp->tm_mon = val[1];
        switch (order) {
        case dmy:
            tmp->tm_mday = val[0];
            tmp->tm_year = val[2];
            break;
        case ymd:
            tmp->tm_year = val[0];
            tmp->tm_mday = val[2];
            break;
        default:
            r |= ios_base::badbit;
            return b;
        }
    }
    else if (res[2] == month) {         // ydm or error
        // ...
    }
    else {                              // rely on dateorder or error
        // ...
    }

    tmp->tm_year -= 1900;   // adjust base year to suit tm convention
    return b;
}
```

I have omitted bits of code that do not add to the understanding of locales, dates, or the handling of input. Writing better and more general date input functions are left as exercises (§D.6[9-10]).

Here is a simple test program:

```
int main()
try {
        cin.imbue(loc(locale(), new Date_in));   // read Dates using Date_in

        while (cin >> dd && dd != Date()) cout << dd << endl;
}
catch (Date::Bad_date e) {
        cout << "bad date caught: " << e.why << endl;
}
```

Note that ***do_get_date()*** will accept meaningless dates, such as

```
Thursday October 7, 1998
```

and

```
1999/Feb/31
```

The checks for consistency of the year, month, day, and optional day of the week are done in ***Date***'s constructor. It is the ***Date*** class' job to know what constitutes a correct date, and it is not necessary for ***Date_in*** to share that knowledge.

It would be possible to have ***getval()*** or ***do_get_date()*** guess about the meaning of numeric values. For example,

```
12 May 1922
```

is clearly not the day 1922 of year 12. That is, we could ''guess'' that a numeric value that couldn't be a day of the specified month must be a year. Such ''guessing'' can be useful in specific constrained context. However, it in not a good idea in more general contexts. For example,

```
12 May 15
```

could be a date in the year 12, 15, 1912, 1915, 2012, or 2015. Sometimes, a better approach is to augment the notation with clues that disambiguate years and days. For example, ***1st*** and ***15th*** are clearly days of a month. Similarly, ***751BC*** and ***1453AD*** are explicitly identified as years.

D.4.5 Character Classification

When reading characters from input, it is often necessary to classify them to make sense of what is being read. For example, to read a number, an input routine needs to know which letters are digits. Similarly, §6.1.2 showed a use of standard character classification functions for parsing input.

Naturally, classification of characters depends on the alphabet used. Consequently, a facet ***ctype*** is provided to represent character classification in a locale.

The character classes as described by an enumeration called ***mask***:

```
class std::ctype_base {
public:
    enum mask {                 // the actual values are implementation defined
        space = 1,              // whitespace (in "C" locale: ' ', '\n', '\t', ...)
        print = 1<<1,           // printing characters
        cntrl = 1<<2,           // control characters
        upper = 1<<3,           // uppercase characters
        lower = 1<<4,           // lowercase characters
        alpha = 1<<5,           // alphabetic characters
        digit = 1<<6,           // decimal digits
        punct = 1<<7,           // punctuation characters
        xdigit = 1<<8,          // hexadecimal digits
        alnum=alpha|digit,      // alphanumeric characters
        graph=alnum|punct
    };
};
```

This ***mask*** doesn't depend on a particular character type. Consequently, this enumeration is placed in a (non-template) base class.

Clearly, ***mask*** reflects the traditional C and C++ classification (§20.4.1). However, for different character sets, different character values fall into different classes. For example, for the ASCII character set, the integer value *125* represents the character ´}´, which is a punctuation character (***punct***). However, in the Danish national character set, *125* represents the vowel ´å´, which in a Danish locale must be classified as an ***alpha***.

The classification is called a ''mask'' because the traditional efficient implementation of character classification for small character sets is a table in which each entry holds bits representing the classification. For example:

```
table[´a´] == lower|alpha|xdigit
table[´1´] == digit
table[´ ´] == space
```

Given that implementation, ***table***[***c***]&***m*** is nonzero if the character ***c*** is an ***m*** and ***0*** otherwise.

The ***ctype*** facet is defined like this:

```
template <class Ch>
class std::ctype : public locale::facet, public ctype_base {
public:
    typedef Ch char_type;
    explicit ctype(size_t r = 0);

    bool is(mask m, Ch c) const; // is "c" an "m"?

    // place classification for each Ch in [b:e) into v:
    const Ch* is(const Ch* b, const Ch* e, mask* v) const;

    const Ch* scan_is(mask m, const Ch* b, const Ch* e) const;   // find an m
    const Ch* scan_not(mask m, const Ch* b, const Ch* e) const; // find a non-m
```

```
        Ch toupper(Ch c) const;
        const Ch* toupper(Ch* b, const Ch* e) const; // convert [b:e)
        Ch tolower(Ch c) const;
        const Ch* tolower(Ch* b, const Ch* e) const;

        Ch widen(char c) const;
        const char* widen(const char* b, const char* e, Ch* b2) const;
        char narrow(Ch c, char def) const;
        const Ch* narrow(const Ch* b, const Ch* e, char def, char* b2) const;

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~ctype();

        // virtual ''do_'' functions for public functions (see §D.4.1)
    };
```

A call *is*(*m*, *c*) tests whether the character *c* belongs to the classification *m*. For example:

```
    int count_spaces(const string& s, const locale& loc)
    {
        const ctype<char>& ct = use_facet< ctype<char> >(loc);
        int i = 0;
        for(string::const_iterator p = s.begin(); p != s.end(); ++p)
            if (ct.is(ctype_base::space, *p)) ++i;     // whitespace as defined by ct
        return i;
    }
```

Note that it is also possible to use *is*( ) to check whether a character belongs to one of a number of classifications. For example:

```
        ct.is(ctype_base::space|ctype_base::punct, c); // is c whitespace or punctuation in ct?
```

A call *is*(*b*, *e*, *v*) determines the classification of each character in [*b*, *e*) and places it in the corresponding position in the array *v*.

A call *scan_is*(*m*, *b*, *e*) returns a pointer to the first character in [*b*, *e*) that is an *m*. If no character is classified as an *m*, *e* is returned. As ever for standard facets, the public member function is implemented by a call to its ''*do_*'' virtual function. A simple implementation might be:

```
    template <class Ch>
    const Ch* std::ctype<Ch>::do_scan_is(mask m, const Ch* b, const Ch* e) const
    {
        while (b!=e && !is(m, *b)) ++b;
        return b;
    }
```

A call *scan_not*(*m*, *b*, *e*) returns a pointer to the first character in [*b*, *e*) that is not an *m*. If all characters are classified as *m*, *e* is returned.

A call *toupper*(*c*) returns the uppercase version of *c* if such a version exists in the character set used and *c* itself otherwise.

A call *toupper*(*b*, *e*) converts each character in the range [*b*, *e*) to uppercase and returns *e*. A simple implementation might be:

```
template <class Ch>
const Ch* std::ctype<Ch>::to_upper(Ch* b, const Ch* e)
{
    for ( ; b!=e; ++b) *b = toupper(*b);
    return e;
}
```

The *tolower*( ) functions are similar to *toupper*( ) except that they convert to lowercase.

A call *widen*(*c*) transforms the character *c* into its corresponding *Ch* value. If *Ch*'s character set provides several characters corresponding to *c*, the standard specifies that ''the simplest reasonable transformation'' be used. For example,

```
wcout << use_facet< ctype<wchar_t> >(wcout.getloc()).widen('e');
```

will output a reasonable equivalent to the character *e* in *wcout*'s locale.

Translation between unrelated character representations, such as ASCII and EBCDIC, can also be done by using *widen*( ). For example, assume that an *ebcdic* locale exists:

```
char EBCDIC_e = use_facet< ctype<char> >(ebcdic).widen('e');
```

A call *widen*(*b*,*e*,*v*) takes each character in the range [*b*,*e*) and places a widened version in the corresponding position in the array *v*.

A call *narrow*(*ch*,*def*) produces a *char* value corresponding to the character *ch* from the *Ch* type. Again, ''the simplest reasonable transformation'' is to be used. If no such corresponding *char* exist, *def* is returned.

A call *narrow*(*b*,*e*,*def*,*v*) takes each character in the range [*b*,*e*) and places a narrowed version in the corresponding position in the array *v*.

The general idea is that *narrow*( ) converts from a larger character set to a smaller one and that *widen*( ) performs the inverse operation. For a character *c* from the smaller character set, we expect:

```
c == narrow(widen(c),0) // not guaranteed
```

This is true provided that the character represented by *c* has only one representation in ''the smaller character set.'' However, that is not guaranteed. If the characters represented by a *char* are not a subset of those represented by the larger character set (*Ch*), we should expect anomalies and potential problems with code treating characters generically.

Similarly, for a character *ch* from the larger character set, we might expect:

```
widen(narrow(ch,def)) == ch || widen(narrow(ch,def)) == widen(def)  // not guaranteed
```

However, even though this is often the case, it cannot be guaranteed for a character that is represented by several values in the larger character set but only once in the smaller character set. For example, a digit, such as *7*, often has several separate representations in a large character set. The reason for that is typically that a large character set has several conventional character sets as subsets and that the characters from the smaller sets are replicated for ease of conversion.

For every character in the basic source character set (§C.3.3), it is guaranteed that

```
widen(narrow(ch_lit,0)) == ch_lit
```

For example:

```
widen(narrow('x'),0) == 'x'
```

The *narrow*( ) and *widen*( ) functions respect character classifications wherever possible. For example, if *is*(*alpha*,*c*), then *is*(*alpha*,*narrow*(*c*,'*a*')) and *is*(*alpha*,*widen*(*c*)) wherever *alpha* is a valid mask for the locale used.

A major reason for using a *ctype* facet in general and for using *narrow*( ) and *widen*( ) functions in particular is to be able to write code that does I/O and string manipulation for any character set; that is, to make such code generic with respect to character sets. This implies that *iostream* implementations depend critically on these facilities. By relying on *<iostream>* and *<string>*, a user can avoid most direct uses of the *ctype* facet.

A *_byname* version (§D.4, §D.4.1) of *ctype* is provided:

```
template <class Ch> class std::ctype_byname : public ctype<Ch> { /* ... */ };
```

D.4.5.1 Convenience Interfaces

The most common use of the *ctype* facet is to inquire whether a character belongs to a given classification. Consequently, a set of functions is provided for that:

```
template <class Ch> bool isspace(Ch c, const locale& loc);
template <class Ch> bool isprint(Ch c, const locale& loc);
template <class Ch> bool iscntrl(Ch c, const locale& loc);
template <class Ch> bool isupper(Ch c, const locale& loc);
template <class Ch> bool islower(Ch c, const locale& loc);
template <class Ch> bool isalpha(Ch c, const locale& loc);
template <class Ch> bool isdigit(Ch c, const locale& loc);
template <class Ch> bool ispunct(Ch c, const locale& loc);
template <class Ch> bool isxdigit(Ch c, const locale& loc);
template <class Ch> bool isalnum(Ch c, const locale& loc);
template <class Ch> bool isgraph(Ch c, const locale& loc);
```

These functions are trivially implemented by using *use_facet*. For example:

```
template <class Ch>
inline bool isspace(Ch c, const locale& loc)
{
    return use_facet< ctype<Ch> >(loc).is(space,c);
}
```

The one-argument versions of these functions, presented in §20.4.2, are simply these functions for the current C global locale (not the global C++ locale, *locale*( )). Except for the rare cases in which the C global locale and the C++ global locale differ (§D.2.3), we can think of a one-argument version as the two-argument version applied to *locale*( ). For example:

```
inline int isspace(int i)
{
    return isspace(i,locale());    // almost
}
```

D.4.6 Character Code Conversion

Sometimes, the representation of characters stored in a file differs from the desired representation of those same characters in main memory. For example, Japanese characters are often stored in files in which indicators (''shifts'') tell to which of the four common character sets (kanji, katakana, hiragana, and romaji) a given sequence of characters belongs. This is a bit unwieldy because the meaning of each byte depends on its ''shift state,'' but it can save memory because only a kanji requires more than one byte for its representation. In main memory, these characters are easier to manipulate when represented in a multi-byte character set where every character has the same size. Such characters (for example, Unicode characters) are typically placed in wide characters (***wchar_t***; §4.3). Consequently, the ***codecvt*** facet provides a mechanism for converting characters from one representation to another as they are read or written. For example:

This code-conversion mechanism is general enough to provide arbitrary conversions of character representations. It allows us to write a program to use a suitable internal character representation (stored in ***char***, ***wchar_t***, or whatever) and to then accept a variety of input character stream representations by adjusting the locale used by iostreams. The alternative would be to modify the program itself or to convert input and output files from/to a variety of formats.

The ***codecvt*** facet provides conversion between different character sets when a character is moved between a stream buffer and external storage:

```
class std::codecvt_base {
public:
    enum result { ok, partial, error, noconv };      // result indicators
};

template <class I, class E, class State>
class std::codecvt : public locale::facet, public codecvt_base {
public:
    typedef I intern_type;
    typedef E extern_type;
    typedef State state_type;

    explicit codecvt(size_t r = 0);

    result in(State&, const E* from, const E* from_end, const E*& from_next, // read
                      I* to, I* to_end, I*& to_next) const;
```

```
        result out(State&, const I* from, const I* from_end, const I*& from_next, // write
                   E* to, E* to_end, E*& to_next) const;

        result unshift(State&, E* to, E* to_end, E*& to_next) const; // end character sequence

        int encoding() const throw();          // characterize basic encoding properties
        bool always_noconv() const throw();    // can we do I/O without code translation?

        int length(const State&, const E* from, const E* from_end, size_t max) const;
        int max_length() const throw();        // maximum possible length()

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~codecvt();

        // virtual ''do_'' functions for public functions (see §D.4.1)
    };
```

A *codecvt* facet is used by *basic_filebuf* (§21.5) to read or write characters. A *basic_filebuf* obtains this facet from the stream's locale (§21.7.1).

The *State* template argument is the type used to hold the shift state of the stream being converted. *State* can also be used to identify different conversions by specifying a specialization. The latter is useful because characters of a variety of character encodings (character sets) can be stored in objects of the same type. For example:

```
class JISstate { /* .. */ };

p = new codecvt<wchar_t, char, mbstate_t>;     // standard char to wide char
q = new codecvt<wchar_t, char, JISstate>;      // JIS to wide char
```

Without the different *State* arguments, there would be no way for the facet to know which encoding to assume for the stream of *char*s. The *mbstate_t* type from *<cwchar>* or *<wchar.h>* identifies the system's standard conversion between *char* and *wchar_t*.

A new *codecvt* can be also created as a derived class and identified by name. For example:

```
class JIScvt : public codecvt<wchar_t, char, mbstate_t> { /* ... */ };
```

A call *in*(*s*, *from*, *from_end*, *from_next*, *to*, *to_end*, *to_next*) reads each character in the range [*from*, *from_end*) and tries to convert it. If a character is converted, *in*() writes its converted form to the corresponding position in the [*to*, *to_end*) range; if not, *in*() stops at that point. Upon return, *in*() stores the position one-beyond-the-last character read in *from_next* and the position one-beyond-the-last character written in *to_next*. The *result* value returned by *in*() indicates how much work was done:

ok:	all characters in the [*from*, *from_end*) range converted
partial:	not all characters in the [*from*, *from_end*) range were converted
error:	*in*() encountered a character it couldn't convert
noconv:	no conversion was needed

Note that a *partial* conversion is not necessarily an error. Possibly more characters have to be read before a multibyte character is complete and can be written, or maybe the output buffer has to be emptied to make room for more characters.

The *s* argument of type *State* indicates the state of the input character sequence at the start of the call of *in*( ). This is significant when the external character representation uses shift states. Note that *s* is a (non-*const*) reference argument: At the end of the call, *s* holds the state of shift state of the input sequence. This allows a programmer to deal with *partial* conversions and to convert a long sequence using several calls to *in*( ).

A call *out*(*s*, *from*, *from_end*, *from_next*, *to*, *to_end*, *to_next*) converts [*from*, *from_end*) from the internal to the external representation in the same way the *in*( ) converts from the external to the internal representation.

A character stream must start and end in a ''neutral'' (unshifted) state. Typically, that state is *State*( ). A call *unshift*(*s*, *to*, *to_end*, *to_next*) looks at *s* and places characters in [*to*, *to_end*) as needed to bring a sequence of characters back to that unshifted state. The result of *unshift*( ) and the use of *to_next* are done just like *out*( ).

A call *length*(*s*, *from*, *from_end*, *max*) returns the number of characters that *in*( ) could convert from [*from*, *from_end*).

A call *encoding*( ) returns

- *–1* if the encoding of the external character set uses state (for example, uses shift and unshift character sequences)
- *0* if the encoding uses varying number of bytes to represent individual characters (for example, a character representation might use a bit in a byte to indicate whether one or two bytes are used to represents that character)
- *n* if every character of the external character representation is *n* bytes

A call *always_noconv*( ) returns *true* if no conversion is required between the internal and the external character sets and *false* otherwise. Clearly, *always_noconv*( )==*true* opens the possibility for the implementation to provide the maximally efficient implementation that simply doesn't invoke the conversion functions.

A call *max_length*( ) returns the maximum value that *length*( ) can return for a valid set of arguments.

The simplest code conversion that I can think of is one that converts input to uppercase. Thus, this is about as simple as a *codecvt* can be and still perform a service:

```
class Cvt_to_upper : public codecvt<char, char, mbstate_t> {     // convert to uppercase
    explicit Cvt_to_upper(size_t r = 0) : codecvt(r) { }

protected:
    // read external representation write internal representation:
    result do_in(State& s, const char* from, const char* from_end, const char*& from_next,
                 char* to, char* to_end, char*& to_next) const;

    // read internal representation write external representation:
    result do_out(State& s, const char* from, const char* from_end, const char*& from_next,
                  char* to, char* to_end, char*& to_next) const
    {
        return codecvt<char, char, mbstate_t>::do_out
            (s, from, from_end, from_next, to, to_end, to_next);
    }
```

```
        result do_unshift(State&, E* to, E* to_end, E*& to_next) const { return ok; }

        int do_encoding() const throw() { return 1; }
        bool do_always_noconv() const throw() { return false; }

        int do_length(const State&, const E* from, const E* from_end, size_t max) const;
        int do_max_length() const throw();        // maximum possible length()
    };

    codecvt<char,char,mbstate_t>::result
    Cvt_to_upper::do_in(State& s, const char* from, const char* from_end,
                        const char*& from_next, char* to, char* to_end, char*& to_next) const
    {
        // ... §D.6[16] ...
    }

    int main()        // trivial test
    {
        locale ulocale(locale(), new Cvt_to_upper);

        cin.imbue(ulocale);

        char ch;
        while (cin>>ch) cout << ch;
    }
```

A *_byname* version (§D.4, §D.4.1) of *codecvt* is provided:

```
    template <class I, class E, class State>
    class std::codecvt_byname : public codecvt<I,E,State> { /* ... */ };
```

D.4.7 Messages

Naturally, most end users prefer to use their native language to interact with a program. However, we cannot provide a standard mechanism for expressing *locale*-specific general interactions. Instead, the library provides a simple mechanism for keeping a *locale*-specific set of strings from which a programmer can compose simple messages. In essence, *messages* implements a trivial read-only database:

```
    class std::messages_base {
    public:
        typedef int catalog;  // catalog identifier type
    };

    template <class Ch>
    class std::messages : public locale::facet, public messages_base {
    public:
        typedef Ch char_type;
        typedef basic_string<Ch> string_type;

        explicit messages(size_t r = 0);
```

```
        catalog open(const basic_string<char>& fn, const locale&) const;
        string_type get(catalog c, int set, int msgid, const string_type& d) const;
        void close(catalog c) const;

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~messages();

        // virtual ''do_'' functions for public functions (see §D.4.1)
    };
```

A call *open*(*s*, *loc*) opens a ''catalog'' of messages called *s* for the locale *loc*. A catalog is a set of strings organized in an implementation-specific way and accessed through the *messages*::*get*() function. A negative value is returned if no catalog named *s* can be opened. A catalog must be opened before the first use of *get*().

A call *close*(*cat*) closes the catalog identified by *cat* and frees all resources associated with that catalog.

A call *get*(*cat*, *set*, *id*, "*foo*") looks for a message identified by (*set*, *id*) in the catalog *cat*. If a string is found, *get*() returns that string; otherwise, *get*() returns the default string (here, *string*("*foo*")).

Here is an example of a *messages* facet for an implementation in which a message catalog is a vector of sets of ''messages'' and a ''message'' is a string:

```
    struct Set {
        vector<string> msgs;
    };

    struct Cat {
        vector<Set> sets;
    };

    class My_messages : public messages<char> {
        vector<Cat>& catalogs;
    public:
        explicit My_messages(size_t = 0) :catalogs(*new vector<Cat>) { }

        catalog do_open(const string& s, const locale& loc) const;      // open catalog s
        string do_get(catalog c, int s, int m, const string&) const;    // get message (s,m) in c
        void do_close(catalog cat) const
        {
            if (catalogs.size()<=cat) catalogs.erase(catalogs.begin()+cat);
        }

        ~My_messages() { delete &catalogs; }
    };
```

All *messages*' member functions are *const*, so the catalog data structure (the *vector<Set>*) is stored outside the facet.

A message is selected by specifying a catalog, a set within that catalog, and a message string within that set. A string is supplied as an argument, to be used as a default result in case no message is found in the catalog:

```
string My_messages::do_get(catalog cat, int set, int msg, const string& def) const
{
    if (catalogs.size()<=cat) return def;
    Cat& c = catalogs[cat];
    if (c.sets.size()<=set) return def;
    Set& s = c.sets[set];
    if (s.msgs.size()<=msg) return def;
    return s.msgs[msg];
}
```

Opening a catalog involves reading a textual representation from disk into a ***Cat*** structure. Here, I chose a representation that is trivial to read. A set is delimited by <<< and >>>, and each message is a line of text:

```
messages<char>::catalog My_messages::do_open(const string& n, const locale& loc) const
{
    string nn = n + locale().name();
    ifstream f(nn.c_str());
    if (!f) return -1;

    catalogs.push_back(Cat());              // make in-core catalog
    Cat& c = catalogs.back();
    string s;
    while (f>>s && s=="<<<") {              // read Set
        c.sets.push_back(Set());
        Set& ss = c.sets.back();
        while (getline(f,s) && s != ">>>") ss.msgs.push_back(s);      // read message
    }
    return catalogs.size()-1;
}
```

Here is a trivial use:

```
int main()
{
    if (!has_facet< My_messages >(locale())) {
        cerr << "no messages facet found in " << locale().name() << '\n';
        exit(1);
    }

    const messages<char>& m = use_facet< My_messages >(locale());
    extern string message_directory;        // where I keep my messages
    int cat = m.open(message_directory,locale());
    if (cat<0) {
        cerr << "no catalog found\n";
        exit(1);
    }

    cout << m.get(cat,0,0,"Missed again!") << endl;
    cout << m.get(cat,1,2,"Missed again!") << endl;
```

```
        cout << m.get(cat,1,3,"Missed again!") << endl;
        cout << m.get(cat,3,0,"Missed again!") << endl;
}
```

If the catalog is

```
<<<
hello
goodbye
>>>
<<<
yes
no
maybe
>>>
```

this program prints

```
hello
maybe
Missed again!
Missed again!
```

D.4.7.1 Using Messages from Other Facets

In addition to being a repository for *locale*-dependent strings used to communicate with users, messages can be used to hold strings for other facets. For example, the *Season_io* facet (§D.3.2) could have been written like this:

```
class Season_io : public locale::facet {
        const messages<char>& m;        // message directory
        int cat;                        // message catalog
public:
        class Missing_messages { };

        Season_io(int i = 0)
                : locale::facet(i),
                  m(use_facet<Season_messages>(locale())),
                  cat(m.open(message_directory, locale()))
        { if (cat<0) throw Missing_messages(); }

        ~Season_io() { }      // to make it possible to destroy Season_io objects (§D.3)

        const string& to_str(Season x) const;          // string representation of x

        bool from_str(const string& s, Season& x) const;  // place Season corresponding to s in x

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
};

locale::id Season_io::id; // define the identifier object
```

```
const string& Season_io::to_str(Season x) const
{
    return m->get(cat,x,"no-such-season");
}

bool Season_io::from_str(const string& s, Season& x) const
{
    for (int i = Season::spring; i<=Season::winter; i++)
        if (m->get(cat,i,"no-such-season") == s) {
            x = Season(i);
            return true;
        }
    return false;
}
```

This ***messages***-based solution differs from the original solution (§D.3.2) in that the implementer of a set of ***Season*** strings for a new locale needs to be able to add them to a ***messages*** directory. This is easy for someone adding a new locale to an execution environment. However, since ***messages*** provides only a read-only interface, adding a new set of season names may be beyond the scope of an application programmer.

A ***_byname*** version (§D.4, §D.4.1) of ***messages*** is provided:

```
template <class Ch>
class std::messages_byname : public messages<Ch> { /* ... */ };
```

D.5 Advice

[1] Expect that every nontrivial program or system that interacts directly with people will be used in several different countries; §D.1.
[2] Don't assume that everyone uses the same character set as you do; §D.4.1.
[3] Prefer using ***locale***s to writing ad hoc code for culture-sensitive I/O; §D.1.
[4] Avoid embedding locale name strings in program text; §D.2.1.
[5] Minimize the use of global format information; §D.2.3, §D.4.4.7.
[6] Prefer locale-sensitive string comparisons and sorts; §D.2.4, §D.4.1.
[7] Make ***facet***s immutable; §D.2.2, §D.3.
[8] Keep changes of ***locale*** to a few places in a program; §D.2.3.
[9] Let ***locale*** handle the lifetime of ***facet***s; §D.3.
[10] When writing locale-sensitive I/O functions, remember to handle exceptions from user-supplied (overriding) functions; §D.4.2.2.
[11] Use a simple ***Money*** type to hold monetary values; §D.4.3.
[12] Use simple user-defined types to hold values that require locale-sensitive I/O (rather than casting to and from values of built-in types); §D.4.3.
[13] Don't believe timing figures until you have a good idea of all factors involved; §D.4.4.1.
[14] Be aware of the limitations of ***time_t***; §D.4.4.1, §D.4.4.5.
[15] Use a date-input routine that accepts a range of input formats; §D.4.4.5.
[16] Prefer the character classification functions in which the locale is explicit; §D.4.5, §D.4.5.1.

D.6 Exercises

1. (∗2.5) Define a ***Season_io*** (§D.3.2) for a language other than American English.
2. (∗2) Define a ***Season_io*** (§D.3.2) class that takes a set of name strings as a constructor argument so that ***Season*** names for different locales can be represented as objects of this class.
3. (∗3) Write a ***collate<char>::compare()*** that gives dictionary order. Preferably, do this for a language, such as German or French, that has more letters in its alphabet than English does.
4. (∗2) Write a program that reads and writes ***bool***s as numbers, as English words, and as words in another language of your choice.
5. (∗2.5) Define a ***Time*** type for representing time of day. Define a ***Date_and_time*** type by using ***Time*** and a ***Date*** type. Discuss the pros and cons of this approach compared to the ***Date*** from (§D.4.4). Implement ***locale***-sensitive I/O for ***Time*** and ***Date_and_time***.
6. (∗2.5) Design and implement a postal code (zip code) facet. Implement it for at least two countries with dissimilar conventions for writing addresses. For example: ***NJ 07932*** and ***CB2***1QA .
7. (∗2.5) Design and implement a phone number facet. Implement it for at least two countries with dissimilar conventions for writing phone numbers. For example, ***(973) 360-8000*** and ***1223 343000***.
8. (∗2.5) Experiment to find out what input and output formats your implementation uses for date information.
9. (∗2.5) Define a ***get_time()*** that ‘‘guesses’’ about the meaning of ambiguous dates, such as 12/5/1995, but still rejects all or almost all mistakes. Be precise about what ‘‘guesses’’ are accepted, and discuss the likelihood of a mistake.
10. (∗2) Define a ***get_time()*** that accepts a greater variety of input formats than the one in §D.4.4.5.
11. (∗2) Make a list of the locales supported on your system.
12. (∗2.5) Figure out where named locales are stored on your system. If you have access to the part of the system where locales are stored, make a new named locale. Be very careful not to break existing locales.
13. (∗2) Compare the two ***Season_io*** implementations (§D.3.2 and §D.4.7.1).
14. (∗2) Write and test a ***Date_out*** facet that writes ***Date***s using a format supplied as a constructor argument. Discuss the pros and cons of this approach compared to the global date format provided by ***date_fmt*** (§D.4.4.6).
15. (∗2.5) Implement I/O of Roman numerals (such as ***XI*** and ***MDCLII***).
16. (∗2.5) Implement and test ***Cvt_to_upper*** (§D.4.6).
17. (∗2.5) Use ***clock()*** to determine average cost of (1) a function call, (2) a virtual function call, (3) reading a ***char***, (4) reading a 1-digit ***int***, (5) reading a 5-digit ***int***, (6) reading a 5-digit ***double***, (7) a 1-character ***string***, (8) a 5-character ***string***, and (9) a 40-character ***string***.
18. (∗6.5) Learn another natural language.

Appendix E

Standard-Library Exception Safety

Everything will work just as you expect it to,
unless your expectations are incorrect.
– Hyman Rosen

Exception safety — exception-safe implementation techniques — representing resources — assignment — ***push_back***( ) — constructors and invariants — standard container guarantees — insertion and removal of elements — guarantees and tradeoffs — ***swap***( ) — initialization and iterators — references to elements — predicates — ***strings***, streams, algorithms, ***valarray***, and ***complex*** — the C standard library — implications for library users — advice — exercises.

E.1 Introduction

Standard-library functions often invoke operations that a user supplies as function or template arguments. Naturally, some of these user-supplied operations will occasionally throw exceptions. Other functions, such as allocator functions, can also throw exceptions. Consider:

```
void f(vector<X>& v, const X& g)
{
    v[2] = g;                       // X's assignment might throw an exception
    v.push_back(g);                 // vector<X>'s allocator might throw an exception
    sort(v.begin(),v.end());        // X's less-than operation might throw an exception
    vector<X> u = v;                // X's copy constructor might throw an exception
    // ...

    // u destroyed here: we must ensure that X's destructor can work correctly
}
```

What happens if the assignment throws an exception while trying to copy *g*? Will *v* be left with an invalid element? What happens if the constructor that *v.push_back()* uses to copy *g* throws *std::bad_alloc*? Has the number of elements changed? Has an invalid element been added to the container? What happens if *X*'s less-than operator throws an exception during the sort? Have the elements been partially sorted? Could an element have been removed from the container by the sorting algorithm and not put back?

Finding the complete list of possible exceptions in this example is left as an exercise (§E.8[1]). Explaining how this example is well behaved for every well-defined type *X* – even an *X* that throws exceptions – is part of the aim of this appendix. Naturally, a major part of this explanation involves giving meaning and effective terminology to the notions of ''well behaved'' and ''well defined'' in the context of exceptions.

The purpose of this appendix is to

[1] identify how a user can design types that meet the standard library's requirements,

[2] state the guarantees offered by the standard library,

[3] state the standard-library requirements on user-supplied code,

[4] demonstrate effective techniques for crafting exception-safe and efficient containers, and

[5] present a few general rules for exception-safe programming.

The discussion of exception safety necessarily focuses on worst-case behavior. That is, where could an exception cause the most problems? How does the standard library protect itself and its users from potential problems? And, how can users help prevent problems? Please don't let this discussion of exception-handling techniques distract from the central fact that throwing an exception is the best method for reporting an error (§14.1, §14.9). The discussion of concepts, techniques, and standard-library guarantees is organized like this:

§E.2 discusses the notion of exception safety.

§E.3 presents techniques for implementing efficient exception-safe containers and operations.

§E.4 outlines the guarantees offered for standard-library containers and their operations.

§E.5 summarizes exception-safety issues for the non-container parts of the standard library.

§E.6 reviews exception safety from the point of view of a standard-library user.

As ever, the standard library provides examples of the kinds of concerns that must be addressed in demanding applications. The techniques used to provide exception safety for the standard library can be applied to a wide range of problems.

E.2 Exception Safety

An operation on an object is said to be *exception safe* if that operation leaves the object in a valid state when the operation is terminated by throwing an exception. This valid state could be an error state requiring cleanup, but it must be well defined so that reasonable error-handling code can be written for the object. For example, an exception handler might destroy the object, repair the object, repeat a variant of the operation, just carry on, etc.

In other words, the object will have an invariant (§24.3.7.1), its constructors will establish that invariant, all further operations maintain that invariant even if an exception is thrown, and its destructor will do final cleanup. An operation should take care that the invariant is maintained before throwing an exception, so that the object is in a valid state. However, it is quite possible for

that valid state to be one that doesn't suit the application. For example, a string may have been left as the empty string or a container may have been left unsorted. Thus, ''repair'' means giving an object a value that is more appropriate/desirable for the application than the one it was left with after an operation failed. In the context of the standard library, the most interesting objects are containers.

Here, we consider under which conditions operations on standard-library containers can be considered exception safe. There can be only two conceptually really simple strategies:

[1] ''*No guarantees*:'' If an exception is thrown, any container being manipulated is possibly corrupted.

[2] ''*Strong guarantee*:'' If an exception is thrown, any container being manipulated remains in the state in which it was before the standard-library operation started.

Unfortunately, both answers are too simple for real use. Alternative [1] is unacceptable because it implies that after an exception is thrown from a container operation, the container cannot be accessed; it can't even be destroyed without fear of run-time errors. Alternative [2] is unacceptable because it imposes the cost of roll-back semantics on every individual standard-library operation.

To resolve this dilemma, the C++ standard library provides a set of exception-safety guarantees that share the burden of producing correct programs between implementers of the standard library and users of the standard library:

[3a] ''*Basic guarantee* for all operations:'' The basic invariants of the standard library are maintained, and no resources, such as memory, are leaked.

[3b] ''*Strong guarantee* for key operations:'' In addition to providing the basic guarantee, either the operation succeeds, or has no effects. This guarantee is provided for key library operations, such as ***push_back***( ), single-element ***insert***( ) on a ***list***, and ***uninitialized_copy***( ) (§E.3.1, §E.4.1).

[3c] ''*Nothrow guarantee* for some operations:'' In addition to providing the basic guarantee, some operations are guaranteed not to throw an exception This guarantee is provided for a few simple operations, such as ***swap***( ) and ***pop_back***( ) (§E.4.1).

Both the basic guarantee and the strong guarantee are provided on the condition that user-supplied operations (such as assignments and ***swap***( ) functions) do not leave container elements in invalid states, that user-supplied operations do not leak resources, and that destructors do not throw exceptions. For example, consider these ''handle-like'' (§25.7) classes:

```
template<class T> class Safe {
        T* p;       // p points to a T allocated using new
public:
        Safe() :p(new T) { }
        ~Safe() { delete p; }
        Safe& operator=(const Safe& a) { *p = *a.p; return *this; }
        // ...
};

template<class T> class Unsafe {       // sloppy and dangerous code
        T* p;       // p points to a T
public:
        Unsafe(T* pp) :p(pp) { }
        ~Unsafe() { if (!p->destructible()) throw E(); delete p; }
```

```
        Unsafe& operator=(const Unsafe& a)
        {
            p->~T();            // destroy old value (§10.4.11)
            new(p) T(a.p);      // construct copy of a.p in *p (§10.4.11)
            return *this;
        }

        // ...
    };

    void f(vector< Safe<Some_type> >&vg, vector< Unsafe<Some_type> >&vb)
    {
        vg.at(1) = Safe<Some_type>();
        vb.at(1) = Unsafe<Some_type>(new Some_type);
        // ...
    }
```

In this example, construction of a ***Safe*** succeeds only if a ***T*** is successfully constructed. The construction of a ***T*** can fail because allocation might fail (and throw ***std::bad_alloc***) and because ***T***'s constructor might throw an exception. However, in every successfully constructed ***Safe***, ***p*** will point to a successfully constructed ***T***; if a constructor fails, no ***T*** object (or ***Safe*** object) is created. Similarly, ***T***'s assignment operator may throw an exception, causing ***Safe***'s assignment operator to implicitly re-throw that exception. However, that is no problem as long as ***T***'s assignment operator always leaves its operands in a good state. Therefore, ***Safe*** is well behaved, and consequently every standard-library operation on a ***Safe*** will have a reasonable and well-defined result.

On the other hand, ***Unsafe()*** is carelessly written (or rather, it is carefully written to demonstrate undesirable behavior). The construction of an ***Unsafe*** will not fail. Instead, the operations on ***Unsafe***, such as assignment and destruction, are left to deal with a variety of potential problems. The assignment operator may fail by throwing an exception from ***T***'s copy constructor. This would leave a ***T*** in an undefined state because the old value of ***p*** was destroyed and no new value replaced it. In general, the results of that are unpredictable. ***Unsafe***'s destructor contains an ill-conceived attempt to protect against undesirable destruction. However, throwing an exception during exception handling will cause a call of ***terminate()*** (§14.7), and the standard library requires that a destructor return normally after destroying an object. The standard library does not – and cannot – make any guarantees when a user supplies objects this badly behaved.

From the point of view of exception handling, ***Safe*** and ***Unsafe*** differ in that ***Safe*** uses its constructor to establish an invariant (§24.3.7.1) that allows its operations to be implemented simply and safely. If that invariant cannot be established, an exception is thrown before an invalid object is constructed. ***Unsafe***, on the other hand, muddles along without a meaningful invariant, and the individual operations throw exceptions without an overall error-handling strategy. Naturally, this results in violations of the standard library's (reasonable) assumptions about the behavior of types. For example, ***Unsafe*** can leave invalid elements in a container after throwing an exception from ***T::operator=()*** and may throw an exception from its destructor.

Note that the standard-library guarantees relative to ill-behaved user-supplied operations are analogous to the language guarantees relative to violations of the basic type system. If a basic operation is not used according to its specification, the resulting behavior is undefined. For

example, if you throw an exception from a destructor for a ***vector*** element, you have no more reason to hope for a reasonable result than if you dereference a pointer initialized to a random number:

```
class Bomb {
public:
        // ...
        ~Bomb() { throw Trouble(); };
};

vector<Bomb> b(10);                          // leads to undefined behavior

void f()
{
        int* p = reinterpret_cast<int*>(rand());   // leads to undefined behavior
        *p = 7;
}
```

Stated positively: If you obey the basic rules of the language and the standard library, the library will behave well even when you throw exceptions.

In addition to achieving pure exception safety, we usually prefer to avoid resource leaks. That is, an operation that throws an exception should not only leave its operands in well-defined states but also ensure that every resource that it acquired is (eventually) released. For example, at the point where an exception is thrown, all memory allocated must be either deallocated or owned by some object, which in turn must ensure that the memory is properly deallocated.

The standard-library guarantees the absence of resource leaks provided that user-supplied operations called by the library also avoid resource leaks. Consider:

```
void leak(bool abort)
{
        vector<int> v(10);                          // no leak
        vector<int>* p = new vector<int>(10);       // potential memory leak
        auto_ptr< vector<int> > q(new vector<int>(10));   // no leak (§14.4.2)

        if (abort) throw Up();
        // ...
        delete p;
}
```

Upon throwing the exception, the ***vector*** called ***v*** and the ***vector*** held by ***q*** will be correctly destroyed so that their resources are released. The ***vector*** pointed to by ***p*** is not guarded against exceptions and will not be destroyed. To make this piece of code safe, we must either explicitly delete ***p*** before throwing the exception or make sure it is owned by an object – such as an ***auto_ptr*** (§14.4.2) – that will properly destroy it if an exception is thrown.

Note that the language rules for partial construction and destruction ensure that exceptions thrown while constructing sub-objects and members will be handled correctly without special attention from standard-library code (§14.4.1). This rule is an essential underpinning for all techniques dealing with exceptions.

Also, remember that memory isn't the only kind of resource that can leak. Opened files, locks, network connections, and threads are examples of system resources that a function may have to release or hand over to an object before throwing an exception.

E.3 Exception-Safe Implementation Techniques

As usual, the standard library provides examples of problems that occur in many other contexts and of solutions that apply widely. The basic tools available for writing exception-safe code are

[1] the *try-block* (§8.3.1), and

[2] the support for the ''resource acquisition is initialization'' technique (§14.4).

The general principles to follow are to

[3] never let go of a piece of information before we can store its replacement, and

[4] always leave objects in valid states when throwing or re-throwing an exception.

That way, we can always back out of an error situation. The practical difficulty in following these principles is that innocent-looking operations (such as `<`, `=`, and ***sort*( )**) might throw exceptions. Knowing what to look for in an application takes experience.

When you are writing a library, the ideal is to aim at the strong exception-safety guarantee (§E.2) and always to provide the basic guarantee. When writing a specific program, there may be less concern for exception safety. For example, if I write a simple data analysis program for my own use, I'm usually quite willing to have the program terminate in the unlikely event of virtual memory exhaustion. However, correctness and basic exception safety are closely related.

The techniques for providing basic exception safety, such as defining and checking invariants (§24.3.7.1), are similar to the techniques that are useful to get a program small and correct. It follows that the overhead of providing basic exception safety (the basic guarantee; §E.2) – or even the strong guarantee – can be minimal or even insignificant; see §E.8[17].

Here, I will consider an implementation of the standard container ***vector*** (§16.3) to see what it takes to achieve that ideal and where we might prefer to settle for more conditional safety.

E.3.1 A Simple Vector

A typical implementation of ***vector*** (§16.3) will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (§17.1.3) (or the equivalent information represented as a pointer plus offsets):

Here is a declaration of ***vector*** simplified to present only what is needed to discuss exception safety and avoidance of resource leaks:

```
template<class T, class A = allocator<T> >
class vector {
private:
        T* v;        // start of allocation
        T* space;    // end of element sequence, start of space allocated for possible expansion
        T* last;     // end of allocated space
        A alloc;     // allocator
```

```
public:
    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);                  // copy constructor
    vector& operator=(const vector& a);       // copy assignment

    ~vector();

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);

    // ...
};
```

Consider first a naive implementation of a constructor:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)  // warning: naive implementation
    :alloc(a)                                               // copy the allocator
{
    v = alloc.allocate(n);                                  // get memory for elements (§19.4.1)
    space = last = v+n;
    for (T* p = v; p!=last; ++p) a.construct(p,val);        // construct copy of val in *p (§19.4.1)
}
```

There are three sources of exceptions here:

[1] ***allocate*( )** throws an exception indicating that no memory is available;

[2] the allocator's copy constructor throws an exception;

[3] the copy constructor for the element type ***T*** throws an exception because it can't copy ***val***.

In all cases, no object is created, so ***vector***'s destructor is not called (§14.4.1).

When ***allocate*( )** fails, the ***throw*** will exit before any resources are acquired, so all is well.

When ***T***'s copy constructor fails, we have acquired some memory that must be freed to avoid memory leaks. A more difficult problem is that the copy constructor for ***T*** might throw an exception after correctly constructing a few elements but before constructing them all.

To handle this problem, we could keep track of which elements have been constructed and destroy those (and only those) in case of an error:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)    // elaborate implementation
    :alloc(a)                    // copy the allocator
{
    v = alloc.allocate(n);       // get memory for elements

    iterator p;

    try {
        iterator end = v+n;
        for (p=v; p!=end; ++p) alloc.construct(p,val);        // construct element (§19.4.1)
        last = space = p;
    }
```

```
        catch ( . . . ) {
            for (iterator q = v; q!=p; ++q) alloc.destroy(q);    // destroy constructed elements
            alloc.deallocate(v,n);              // free memory
            throw;                              // re-throw
        }
    }
```

The overhead here is the overhead of the *try-block*. In a good C++ implementation, this overhead is negligible compared to the cost of allocating memory and initializing elements. For implementations where entering a *try-block* incurs a cost, it may be worthwhile to add a test ***if***(***n***) before the ***try*** and handle the empty vector case separately.

The main part of this constructor is an exception-safe implementation of ***uninitialized_fill***( ):

```
    template<class For, class T>
    void uninitialized_fill(For beg, For end, const T& x)
    {
        For p;
        try {
            for (p=beg; p!=end; ++p)
                new(static_cast<void*>(&*p)) T(x);      // construct copy of x in *p (§10.4.11)
        }
        catch ( . . . ) {  // destroy constructed elements and rethrow:
            for (For q = beg; q!=p; ++q) (&*q)->~T();  // (§10.4.11)
            throw;
        }
    }
```

The curious construct ***&*p*** takes care of iterators that are not pointers. In that case, we need to take the address of the element obtained by dereference to get a pointer. The explicit cast to ***void**** ensures that the standard library placement function is used (§19.4.5), and not some user-defined ***operator new***( ) for ***T****s. This code is operating at a rather low level where writing truly general code can be difficult.

Fortunately, we don't have to reimplement ***uninitialized_fill***( ), because the standard library provides the desired strong guarantee for it (§E.2). It is often essential to have initialization operations that either complete successfully, having initialized every element, or fail leaving no constructed elements behind. Consequently, the standard-library algorithms ***uninitialized_fill***( ), ***uninitialized_fill_n***( ), and ***uninitialized_copy***( ) (§19.4.4) are guaranteed to have this strong exception-safety property (§E.4.4).

Note that the ***uninitialized_fill***( ) algorithm does not protect against exceptions thrown by element destructors or iterator operations (§E.4.4). Doing so would be prohibitively expensive (see §E.8[16-17]).

The ***uninitialized_fill***( ) algorithm can be applied to many kinds of sequences. Consequently, it takes a forward iterator (§19.2.1) and cannot guarantee to destroy elements in the reverse order of their construction.

Using ***uninitialized_fill***( ), we can write:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)    // messy implementation
    :alloc(a)                                   // copy the allocator
{
    v = alloc.allocate(n);                      // get memory for elements
    try {
        uninitialized_fill(v,v+n,val);          // copy elements
        space = last = v+n;
    }
    catch ( . . . ) {
        alloc.deallocate(v,n);                  // free memory
        throw;                                  // re-throw
    }
}
```

However, I wouldn't call that pretty code. The next section will demonstrate how it can be made much simpler.

Note that the constructor re-throws a caught exception. The intent is to make ***vector*** transparent to exceptions so that the user can determine the exact cause of a problem. All standard-library containers have this property. Exception transparency is often the best policy for templates and other ''thin'' layers of software. This is in contrast to major parts of a system (''modules'') that generally need to take responsibility for all exceptions thrown. That is, the implementer of such a module must be able to list every exception that the module can throw. Achieving this may involve grouping exceptions (§14.2), mapping exceptions from lower-level routines into the module's own exceptions (§14.6.3), or exception specification (§14.6).

E.3.2 Representing Memory Explicitly

Experience revealed that writing correct exception-safe code using explicit *try-block*s is more difficult than most people expect. In fact, it is unnecessarily difficult because there is an alternative: The ''resource acquisition is initialization'' technique (§14.4) can be used to reduce the amount of code needing to be written and to make the code more stylized. In this case, the key resource required by the ***vector*** is memory to hold its elements. By providing an auxiliary class to represent the notion of memory used by a ***vector***, we can simplify the code and decrease the chance of accidentally forgetting to release it:

```
template<class T, class A = allocator<T> >
struct vector_base {
    A alloc;   // allocator
    T* v;      // start of allocation
    T* space;  // end of element sequence, start of space allocated for possible expansion
    T* last;   // end of allocated space

    vector_base(const A& a, typename A::size_type n)
        : alloc(a), v(a.allocate(n)), space(v+n), last(v+n) { }
    ~vector_base() { alloc.deallocate(v,last-v); }
};
```

As long as ***v*** and ***last*** are correct, ***vector_base*** can be destroyed. Class ***vector_base*** deals with

memory for a type ***T***, not objects of type ***T***. Consequently, a user of ***vector_base*** must destroy all constructed objects in a ***vector_base*** before the ***vector_base*** itself is destroyed.

Naturally, ***vector_base*** itself is written so that if an exception is thrown (by the allocator's copy constructor or ***allocate*( )** function) no ***vector_base*** object is created and no memory is leaked.

We want to be able to ***swap*( )** ***vector_base***s. However, the default ***swap*( )** doesn't suit our needs because it copies and destroys a temporary. Because ***vector_base*** is a special-purpose class that wasn't given fool-proof copy semantics, that destructions would lead to undesirable sideeffects. Consequently, we provide a specialization:

```
template<class T> void swap(vector_base<T>& a, vector_base<T>& b)
{
    swap(a.a,b.a); swap(a.v,b.v); swap(a.space,b.space); swap(a.last,b.last);
}
```

Given ***vector_base***, ***vector*** can be defined like this:

```
template<class T, class A = allocator<T> >
class vector : private vector_base<T,A> {
    void destroy_elements( ) { for (T* p = v; p!=space; ++p) p->~T( ); }  // §10.4.11
public:
    explicit vector(size_type n, const T& val = T( ), const A& = A( ));

    vector(const vector& a);                  // copy constructor
    vector& operator=(const vector& a);       // copy assignment

    ~vector( ) { destroy_elements( ); }

    size_type size( ) const { return space-v; }
    size_type capacity( ) const { return last-v; }

    void push_back(const T&);

    // ...
};
```

The ***vector*** destructor explicitly invokes the ***T*** destructor for every element. This implies that if an element destructor throws an exception, the ***vector*** destruction fails. This can be a disaster if it happens during stack unwinding caused by an exception and ***terminate*( )** is called (§14.7). In the case of normal destruction, throwing an exception from a destructor typically leads to resource leaks and unpredictable behavior of code relying on reasonable behavior of objects. There is no really good way to protect against exceptions thrown from destructors, so the library makes no guarantees if an element destructor throws (§E.4).

Now the constructor can be simply defined:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :vector_base<T,A>(a,n)        // allocate space for n elements
{
    uninitialized_fill(v,v+n,val);    // copy elements
}
```

The copy constructor differs by using ***uninitialized_copy*( )** instead of ***uninitialized_fill*( )**:

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
    :vector_base<T,A>(a.alloc,a.size())
{
    uninitialized_copy(a.begin(),a.end(),v);
}
```

Note that this style of constructor relies on the fundamental language rule that when an exception is thrown from a constructor, sub-objects (such as bases) that have already been completely constructed will be properly destroyed (§14.4.1). The ***uninitialized_fill*( )** algorithm and its cousins (§E.4.4) provide the equivalent guarantee for partially constructed sequences.

E.3.3 Assignment

As usual, assignment differs from construction in that an old value must be taken care of. Consider a straightforward implementation:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)   // offers the strong guarantee (§E.2)
{
    vector_base<T,A> b(alloc,a.size());              // get memory
    uninitialized_copy(a.begin(),a.end(),b.v);       // copy elements
    destroy_elements();
    alloc.deallocate(v,last-v);                      // free old memory
    vector_base::operator=(b);                       // install new representation
    b.v = 0;                                         // prevent deallocation
    return *this;
}
```

This assignment is safe, but it repeats a lot of code from constructors and destructors. To avoid this, we could write:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)   // offers the strong guarantee (§E.2)
{
    vector temp(a);                                  // copy a
    swap< vector_base<T,A> >(*this,temp);            // swap representations
    return *this;
}
```

The old elements are destroyed by ***temp***'s destructor, and the memory used to hold them is deallocated by ***temp***'s ***vector_base***'s destructor.

The performance of the two versions ought to be equivalent. Essentially, they are just two different ways of specifying the same set of operations. However, the second implementation is shorter and doesn't replicate code from related ***vector*** functions, so writing the assignment that way ought to be less error prone and lead to simpler maintenance.

Note the absence of the traditional test for self-assignment (§10.4.4). These assignment implementations work by first constructing a copy and then swapping representations. This obviously handles self-assignment correctly. I decided that the efficiency gained from the test in the rare case

of self-assignment was more than offset by its cost in the common case where a different *vector* is assigned.

In either case, two potentially significant optimizations are missing:

[1] If the capacity of the vector assigned to is large enough to hold the assigned vector, we don't need to allocate new memory.

[2] An element assignment may be more efficient than an element destruction followed by an element construction.

Implementing these optimizations, we get:

```
template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a)   // optimized, basic guarantee (§E.2)
{
    if (capacity() < a.size()) {      // allocate new vector representation:
        vector temp(a);                                 // copy a
        swap< vector_base<T,A> >(*this,temp);           // swap representations
        return *this;
    }
    if (this == &a) return *this;     // protect against self assignment (§10.4.4)
                                      // assign to old elements:
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();                          // copy the allocator
    if (asz<=sz) {
        copy(a.begin(),a.begin()+asz,v);
        for (T* p = v+asz; p!=space; ++p) p->~T();     // destroy surplus elements (§10.4.11)
    }
    else {
        copy(a.begin(),a.begin()+sz,v);
        uninitialized_copy(a.begin()+sz,a.end(),space);    // construct extra elements
    }
    space = v+asz;
    return *this;
}
```

These optimizations are not free. The *copy*( ) algorithm (§18.6.1) does *not* offer the strong exception-safety guarantee. It does not guarantee that it will leave its target unchanged if an exception is thrown during copying. Thus, if *T*::*operator*=( ) throws an exception during *copy*( ), the *vector* being assigned to need not be a copy of the vector being assigned, and it need not be unchanged. For example, the first five elements might be copies of elements of the assigned vector and the rest unchanged. It is also plausible that an element – the element that was being copied when *T*::*operator*=( ) threw an exception – ends up with a value that is neither the old value nor a copy of the corresponding element in the vector being assigned. However, if *T*::*operator*=( ) leaves its operands in a valid state if it throws an exception, the *vector* is still in a valid state – even if it wasn't the state we would have preferred.

Here, I have copied the allocator using an assignment. It is actually not required that every allocator support assignment (§19.4.3); see also §E.8[9].

The standard-library ***vector*** assignment offers the weaker exception-safety property of this last implementation – and its potential performance advantages. That is, ***vector*** assignment provides the basic guarantee, so it meets most people's idea of exception safety. However, it does not provide the strong guarantee (§E.2). If you need an assignment that leaves the ***vector*** unchanged if an exception is thrown, you must either use a library implementation that provides the strong guarantee or provide your own assignment operation. For example:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // "obvious" a = b
{
    vector<T,A> temp(a.get_allocator());
    temp.reserve(b.size());
    for (typename vector<T,A>::iterator p = b.begin(); p!=b.end(); ++p)
        temp.push_back(*p);
    swap(a,temp);
}
```

If there is insufficient memory for ***temp*** to be created with room for ***b.size()*** elements, ***std::bad_alloc*** is thrown before any changes are made to ***a***. Similarly, if ***push_back()*** fails for any reason, ***a*** will remain untouched because we apply ***push_back()*** to ***temp*** rather than to ***a***. In that case, any elements of ***temp*** created by ***push_back()*** will be destroyed before the exception that caused the failure is re-thrown.

Swap does not copy ***vector*** elements. It simply swaps the data members of a ***vector***; that is, it swaps ***vector_base***s (§E.3.2). Consequently, it does not throw exceptions even if operations on the elements might (§E.4.3). Consequently, ***safe_assign()*** does not do spurious copies of elements and is reasonably efficient.

As is often the case, there are alternatives to the obvious implementation. We can let the library perform the copy into the temporary for us:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // simple a = b
{
    vector<T,A> temp(b);      // copy the elements of b into a temporary
    swap(a,temp);
}
```

Indeed, we could simply use call-by-value (§7.2):

```
template<class T, class A>
void safe_assign(vector<T,A>& a, vector<T,A> b)    // simple a = b (note: b is passed by value)
{
    swap(a,b);
}
```

E.3.4 *push_back()*

From an exception-safety point of view, ***push_back()*** is similar to the assignment in that we must take care that the ***vector*** remains unchanged if we fail to add a new element:

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
    if (space == last) {   // no more free space; relocate:
        vector_base b(alloc,size()?2*size():2); // double the allocation
        uninitialized_copy(v,space,b.v);
        new(b.space) T(x);                      // place a copy of x in *b.space (§10.4.11)
        ++b.space;
        destroy_elements();
        swap<vector_base<T,A> >(b,*this);       // swap representations
        return;
    }
    new(space) T(x);                            // place a copy of x in *space (§10.4.11)
    ++space;
}
```

Naturally, the copy constructor used to initialize ***space** might throw an exception. If that happens, the value of the ***vector*** remains unchanged, with ***space*** left unincremented. In that case, the ***vector*** elements are not reallocated so that iterators referring to them are not invalidated. Thus, this implementation implements the strong guarantee that an exception thrown by an allocator or even a user-supplied copy constructor leaves the ***vector*** unchanged. The standard library offers that guarantee for ***push_back*()** (§E.4.1).

Note the absence of a *try-block* (except for the one hidden in ***uninitialized_copy*()**). The update was done by carefully ordering the operations so that if an exception is thrown, the ***vector*** remains unchanged.

The approach of gaining exception safety through ordering and the ''resource acquisition is initialization'' technique (§14.4) tends to be more elegant and more efficient than explicitly handling errors using *try-block*s. More problems with exception safety arise from a programmer ordering code in unfortunate ways than from lack of specific exception-handling code. The basic rule of ordering is not to destroy information before its replacement has been constructed and can be assigned without the possibility of an exception.

Exceptions introduce possibilities for surprises in the form of unexpected control flows. For a piece of code with a simple local control flow, such as the ***operator*=()**, ***safe_assign*()**, and ***push_back*()** examples, the opportunities for surprises are limited. It is relatively simple to look at such code and ask oneself ''can this line of code throw an exception, and what happens if it does?'' For large functions with complicated control structures, such as complicated conditional statements and nested loops, this can be hard. Adding *try-block*s increases this local control structure complexity and can therefore be a source of confusion and errors (§14.4). I conjecture that the effectiveness of the ordering approach and the ''resource acquisition is initialization'' approach compared to more extensive use of *try-block*s stems from the simplification of the local control flow. Simple, stylized code is easier to understand and easier to get right.

Note that the ***vector*** implementation is presented as an example of the problems that exceptions can pose and of techniques for addressing those problems. The standard does not require an implementation to be exactly like the one presented here. What the standard does guarantee is the subject of §E.4.

E.3.5 Constructors and Invariants

From the point of view of exception safety, other ***vector*** operations are either equivalent to the ones already examined (because they acquire and release resources in similar ways) or trivial (because they don't perform operations that require cleverness to maintain valid states). However, for most classes, such ''trivial'' functions constitute the majority of code. The difficulty of writing such functions depends critically on the environment that a constructor established for them to operate in. Said differently, the complexity of ''ordinary member functions'' depends critically on choosing a good class invariant (§24.3.7.1). By examining the ''trivial'' ***vector*** functions, it is possible to gain insight into the interesting question of what makes a good invariant for a class and how constructors should be written to establish such invariants.

Operations such as ***vector*** subscripting (§16.3.3) are easy to write because they can rely on the invariant established by the constructors and maintained by all functions that acquire or release resources. In particular, a subscript operator can rely on ***v*** referring to an array of elements:

```
template< class T, class A>
T& vector<T,A>::operator[](size_type i)
{
    return v[i];
}
```

It is important and fundamental to have constructors acquire resources and establish a simple invariant. To see why, consider an alternative definition of ***vector_base***:

```
template<class T, class A = allocator<T> >      // clumsy use of constructor
class vector_base {
public:
    A alloc;   // allocator
    T* v;      // start of allocation
    T* space;  // end of element sequence, start of space allocated for possible expansion
    T* last;   // end of allocated space

    vector_base(const A& a, typename A::size_type n) : alloc(a), v(0), space(0), last(0)
    {
        v = alloc.allocate(n);
        space = last = v+n;
    }

    ~vector_base() { if (v) alloc.deallocate(v, last-v); }
};
```

Here, I construct a ***vector_base*** in two stages: First, I establish a ''safe state'' where ***v***, ***space***, and ***last*** are set to ***0***. Only after that has been done do I try to allocate memory. This is done out of misplaced fear that if an exception happens during element allocation, a partially constructed object could be left behind. This fear is misplaced because a partially constructed object cannot be ''left behind'' and later accessed. The rules for static objects, automatic objects, member objects, and elements of the standard-library containers prevent that. However, it could/can happen in pre-standard libraries that used/use placement new (§10.4.11) to construct objects in containers designed without concern for exception safety. Old habits can be hard to break.

Note that this attempt to write safer code complicates the invariant for the class: It is no longer guaranteed that *v* points to allocated memory. Now *v* might be *0*. This has one immediate cost. The standard-library requirements for allocators do not guarantee that we can safely deallocate a pointer with the value *0* (§19.4.1). In this, allocators differ from *delete* (§6.2.6). Consequently, I had to add a test in the destructor. Also, each element is first initialized and then assigned. The cost of doing that extra work can be significant for element types for which assignment is nontrivial, such as *string* and *list*.

This two-stage construct is not an uncommon style. Sometimes, it is even made explicit by having the constructor do only some ''simple and safe'' initialization to put the object into a destructible state. The real construction is left to an *init*( ) function that the user must explicitly call. For example:

```
template<class T>      // archaic (pre-standard, pre-exception) style
class vector_base {
public:
        T* v;        // start of allocation
        T* space;    // end of element sequence, start of space allocated for possible expansion
        T* last;     // end of allocated space

        vector_base( ) : v(0), space(0), last(0) { }
        ~vector_base( ) { free(v); }

        bool init(size_t n)    // return true if initialization succeeded
        {
                if (v = (T*)malloc(sizeof(T)*n)) {
                        uninitialized_fill(v, v+n, T( ));
                        space = last = v+n;
                        return true;
                }
                return false;
        }
};
```

The perceived value of this style is

[1] The constructor can't throw an exception, and the success of an initialization using *init*( ) can be tested by ''usual'' (that is, non-exception) means.

[2] There exists a trivial valid state. In case of a serious problem, an operation can give an object that state.

[3] The acquisition of resources is delayed until a fully initialized object is actually needed.

The following subsections examine these points and shows why this two-stage construction technique doesn't deliver its expected benefits. It can also be a source of problems.

E.3.5.1 Using *init()* Functions

The first point (using an *init*( ) function in preference to a constructor) is bogus. Using constructors and exception handling is a more general and systematic way of dealing with resource acquisition and initialization errors (§14.1, §14.4). This style is a relic of pre-exception C++.

Carefully written code using the two styles are roughly equivalent. Consider:

```
int f1(int n)
{
        vector<X> v;
        // ...
        if (v.init(n)) {
                // use v as vector of n elements
        }
        else {
                // handle_problem
        }
}
```

and

```
int f2(int n)
try {
        vector v<X> v(n);
        // ...
        // use v as vector of n elements
}
catch ( . . . ) {
        // handle problem
}
```

However, having a separate ***init*( )** function is an opportunity to

[1] forget to call ***init*( )** (§10.2.3),
[2] forget to test on the success of ***init*( )**,
[3] call ***init*( )** more than once,
[4] forget that ***init*( )** might throw an exception, and
[5] use the object before calling ***init*( )**.

The definition of ***vector<T>::init*( )** illustrates [4].

In a good C++ implementation, ***f2*( )** will be marginally faster than ***f1*( )** because it avoids the test in the common case.

E.3.5.2 Relying on a Default Valid State

The second point (having an easy-to-construct ‘‘default’’ valid state) is correct in general, but in the case of ***vector***, it is achieved at an unnecessary cost. It is now possible to have a ***vector_base*** with ***v==0***, so the *vector* implementation must protect against that possibility throughout. For example:

```
template< class T>
T& vector<T>::operator[](size_t i)
{
        if (v) return v[i];
        // handle error
}
```

Leaving the possibility of ***v==0*** open makes the cost of non-range-checked subscripting equivalent to range-checked access:

```
template< class T>
T& vector<T>::at(size_t i)
{
    if (i<v.size()) return v[i];
    throw out_of_range("vector index");
}
```

What fundamentally happened here was that I complicated the basic invariant for ***vector_base*** by introducing the possibility of ***v==0***. In consequence, the basic invariant for ***vector*** was similarly complicated. The end result of this is that all code in ***vector*** and ***vector_base*** must be more complicated to cope. This is a source of potential errors, maintenance problems, and run-time overhead. Note that conditional statements can be surprisingly costly on modern machine architectures. Where efficiency matters, it can be crucial to implement a key operation, such as vector subscripting, without conditional statements.

Interestingly, the original definition of ***vector_base*** already did have an easy-to-construct valid state. No ***vector_base*** object could exist unless the initial allocation succeeded. Consequently, the implementer of ***vector*** could write an ''emergency exit'' function like this:

```
template< class T, class A>
void vector<T,A>::emergency_exit()
{
    space = v;                  // set the size of *this to 0
    throw Total_failure();
}
```

This is a bit drastic because it fails to call element destructors and to deallocate the space for elements held by the ***vector_base***. That is, it fails to provide the basic guarantee (§E.2). If we are willing to trust the values of ***v*** and ***space*** and the element destructors, we can avoid potential resource leaks:

```
template< class T, class A>
void vector<T,A>::emergency_exit()
{
    destroy_elements();         // clean up
    throw Total_failure();
}
```

Please note that the standard ***vector*** is such a clean design that it minimizes the problems caused by two-phase construction. The ***init()*** function is roughly equivalent to ***resize()***, and in most places the possibility of ***v==0*** is already covered by ***size()==0*** tests. The negative effects described for two-phase construction become more marked when we consider application classes that acquire significant resources, such as network connections and files. Such classes are rarely part of a framework that guides their use and their implementation in the way the standard-library requirements guide the definition and use of ***vector***. The problems also tend to increase as the mapping between the application concepts and the resources required to implement them becomes more complex. Few classes map as directly onto system resources as does ***vector***.

The idea of having a ''safe state'' is in principle a good one. If we can't put an object into a valid state without fear of throwing an exception before completing that operation, we do indeed

have a problem. However, this ''safe state'' should be one that is a natural part of the semantics of the class rather than an implementation artifact that complicates the class invariant.

E.3.5.3 Delaying resource acquisition

Like the second point (§E.3.5.2), the third (to delay acquisition until a resource is needed) misapplies a good idea in a way that imposes cost without yielding benefits. In many cases, notably in containers such as ***vector***, the best way of delaying resource acquisition is for the programmer to delay the creation of objects until they are needed. Consider a naive use of ***vector***:

```
void f(int n)
{
    vector<X> v(n);        // make n default objects of type X
    // ...

    v[3] = X(99);          // real ''initialization'' of v[3]
    // ...
}
```

Constructing an ***X*** only to assign a new value to it later is wasteful – especially if an ***X*** assignment is expensive. Therefore, two-phase construction of ***X*** can seem attractive. For example, the type ***X*** may itself be a ***vector***, so we might consider two-phase construction of ***vector*** to optimize creation of empty ***vector***s. However, creating default (empty) vectors is already efficient, so complicating the implementation with a special case for the empty vector seems futile. More generally, the best solution to spurious initialization is rarely to remove complicated initialization from the element constructors. Instead, a user can create elements only when needed. For example:

```
void f2(int n)
{
    vector<X> v;                 // make empty vector
    // ...

    v.push_back(X(99));          // construct element when needed
    // ...
}
```

To sum up: the two-phase construction approach leads to more complicated invariants and typically to less elegant, more error-prone, and harder-to-maintain code. Consequently, the language-supported ''constructor approach'' should be preferred to the ''***init***( )-function approach'' whenever feasible. That is, resources should be acquired in constructors whenever delayed resource acquisition isn't mandated by the inherent semantics of a class.

E.4 Standard Container Guarantees

If a library operation itself throws an exception, it can – and does – make sure that the objects on which it operates are left in a well-defined state. For example, ***at***( ) throwing ***out_of_range*** for a ***vector*** (§16.3.3) is not a problem with exception safety for the ***vector***. The writer of ***at***( ) has no problem making sure that a ***vector*** is in a well-defined state before throwing. The problems – for

library implementers, for library users, and for people trying to understand code – come when a user-supplied function throws an exception.

The standard-library containers offer the basic guarantee (§E.2): The basic invariants of the library are maintained, and no resources are leaked as long as user code behaves as required. That is, user-supplied operations should not leave container elements in invalid states or throw exceptions from destructors. By ''operations,'' I mean operations used by the standard-library implementation, such as constructors, assignments, destructors, and operations on iterators (§E.4.4).

It is relatively easy for the programmer to ensure that such operations meet the library's expectations. In fact, much naively written code conforms to the library's requirements. The following types clearly meet the standard library's requirements for container element types:

[1] built-in types – including pointers,
[2] types without user-defined operations,
[3] classes with operations that neither throw exceptions nor leave operands in invalid states,
[4] classes with destructors that don't throw exceptions and for which it is simple to verify that operations used by the standard library (such as constructors, assignments, <, ==, and *swap*( )) don't leave operands in invalid states.

In each case, we must also make sure that no resource is leaked. For example:

```
void f(Circle* pc, Triangle* pt, vector<Shape*>& v2)
{
    vector<Shape*> v(10);          // either create vector or throw bad_alloc
    v[3] = pc;                     // no exception thrown
    v.insert(v.begin()+4,pt);      // either insert pt or no effect on v
    v2.erase(v2.begin()+3);        // either erase v2[3] or no effect on v2
    v2 = v;                        // copy v or no effect on v2
    // ...
}
```

When *f*( ) exits, *v* will be properly destroyed, and *v2* will be in a valid state. This fragment does not indicate who is responsible for deleting *pc* and *pt*. If *f*( ) is responsible, it can either catch exceptions and do the required deletion, or assign the pointers to local *auto_ptr*s.

The more interesting question is: When do the library operations offer the strong guarantee that an operation either succeeds or has no effect on its operands? For example:

```
void f(vector<X>& vx)
{
    vx.insert(vx.begin()+4,X(7));      // add element
}
```

In general, *X*'s operations and *vector<X>*'s allocator can throw an exception. What can we say about the elements of *vx* when *f*( ) exits because of an exception? The basic guarantee ensures that no resources have been leaked and that *vx* has a set of valid elements. However, exactly what elements? Is *vx* unchanged? Could a default *X* have been added? Could an element have been removed because that was the only way for *insert*( ) to recover while maintaining the basic guarantee? Sometimes, it is not enough to know that a container is in a good state; we also want to know exactly what state that is. After catching an exception, we typically want to know that the elements are exactly those we intended, or we will have to start error recovery.

E.4.1 Insertion and Removal of Elements

Inserting an element into a container and removing one are obvious examples of operations that might leave a container in an unpredictable state if an exception is thrown. The reason is that insertions and deletions invoke many operations that may throw exceptions:

[1] A new value is copied into a container.
[2] An element deleted from (erased from) a container must be destroyed.
[3] Sometimes, memory must be allocated to hold a new element.
[4] Sometimes, ***vector*** and ***deque*** elements must be copied to new locations.
[5] Associative containers call comparison functions for elements.
[6] Many insertions and deletions involve iterator operations.

Each of these cases can cause an exception to be thrown.

If a destructor throws an exception, no guarantees are made (§E.2). Making guarantees in this case would be prohibitively expensive. However, the library can and does protect itself – and its users – from exceptions thrown by other user-supplied operations.

When manipulating a linked data structure, such as a ***list*** or a ***map***, elements can be added and removed without affecting other elements in the container. This is not the case for a container implemented using contiguous allocation of elements, such as a ***vector*** or a ***deque***. There, elements sometimes need to be moved to new locations.

In addition to the basic guarantee, the standard library offers the strong guarantee for a few operations that insert or remove elements. Because containers implemented as linked data structures behave differently from containers with contiguous allocation of elements, the standard provides slightly different guarantees for different kinds of containers:

[1] Guarantees for ***vector*** (§16.3) and ***deque*** (§17.2.3):
- If an exception is thrown by a ***push_back***( ) or a ***push_front***( ), that function has no effect.
- Unless thrown by the copy constructor or the assignment operator of the element type, if an exception is thrown by an ***insert***( ), that function has no effect.
- Unless thrown by the copy constructor or the assignment operator of the element type, no ***erase***( ) throws an exception.
- No ***pop_back***( ) or ***pop_front***( ) throws an exception.

[2] Guarantees for ***list*** (§17.2.2):
- If an exception is thrown by a ***push_back***( ) or a ***push_front***( ), that function has no effect.
- If an exception is thrown by an ***insert***( ), that function has no effect.
- No ***erase***( ), ***pop_back***( ), ***pop_front***( ), ***splice***( ), or ***reverse***( ) throws an exception.
- Unless thrown by a predicate or a comparison function, the ***list*** member functions ***remove***( ), ***remove_if***( ), ***unique***( ), ***sort***( ), and ***merge***( ) do not throw exceptions.

[3] Guarantees for associative containers (§17.4):
- If an exception is thrown by an ***insert***( ) while inserting a single element, that function has no effect.
- No ***erase***( ) throws an exception.

Note that where the strong guarantee is provided for an operation on a container, all iterators, pointers to elements, and references to elements remain valid if an exception is thrown.

These rules can be summarized in a table:

Container-Operation Guarantees				
	vector	**deque**	**list**	**map**
clear()	nothrow (*copy*)	nothrow (*copy*)	nothrow	nothrow
erase()	nothrow (*copy*)	nothrow (*copy*)	nothrow	nothrow
1-element insert()	strong (*copy*)	strong (*copy*)	strong	strong
N-element insert()	strong (*copy*)	strong (*copy*)	strong	basic
merge()	—	—	nothrow (*comparison*)	—
push_back()	strong	strong	strong	—
push_front()	—	strong	strong	—
pop_back()	nothrow	nothrow	nothrow	—
pop_front()	—	nothrow	nothrow	—
remove()	—	—	nothrow (*comparison*)	—
remove_if()	—	—	nothrow (*predicate*)	—
reverse()	—	—	nothrow	—
splice()	—	—	nothrow	—
swap()	nothrow	nothrow	nothrow	nothrow (*copy-of-comparison*)
unique()	—	—	nothrow (*comparison*)	—

In this table:

basic means that the operation provides only the basic guarantee (§E.2)
strong means that the operation provides the strong guarantee (§E.2)
nothrow means that the operation does not throw an exception (§E.2)
— means that the operation is not provided as a member of this container

Where a guarantee requires that some user-supplied operations not throw exceptions, those operations are indicated in parentheses under the guarantee. These requirements are precisely stated in the text preceding the table.

The *swap*( ) functions differ from the other functions mentioned by not being members. The guarantee for *clear*( ) is derived from that offered by *erase*( ) (§16.3.6). This table lists guarantees offered in addition to the basic guarantee. Consequently this table does not list operations, such as *reverse*( ) and *unique*( ) for *vector*, that are provided only as algorithms for all sequences without additional guarantees.

The ''almost container'' *basic_string* (§17.5, §20.3) offers the basic guarantee for all operations (§E.5.1). The standard also guarantees that *basic_string*'s *erase*( ) and *swap*( ) don't throw, and offers the strong guarantee for *basic_string*'s *insert*( ) and *push_back*( ).

In addition to ensuring that a container is unchanged, an operation providing the strong guarantee also leaves all iterators, pointers, and references valid. For example:

```
void update(map<string,X>& m, map<string,X>::iterator current)
{
    X x;
    string s;
    while (cin>>s>>x)
    try {
        current = m.insert(current, make_pair(s,x));
    }
    catch(...) {
        // here current still denotes the current element
    }
}
```

E.4.2 Guarantees and Tradeoffs

The patchwork of additional guarantees reflects implementation realities. Programmers prefer the strong guarantee with as few conditions as possible, but they also tend to insist that each individual standard-library operation be optimally efficient. Both concerns are reasonable, but for many operations, it is not possible to satisfy both simultaneously. To give a better idea of the tradeoffs involved, I'll examine ways of adding of single and multiple elements to *list*s, *vector*s, and *map*s.

Consider adding a single element to a *list* or a *vector*. As ever, *push_back*( ) provides the simplest way of doing that:

```
void f(list<X>& lst, vector<X>& vec, const X& x)
{
    try {
        lst.push_back(x);          // add to list
    }
    catch (...) {
        // lst is unchanged
        return;
    }
```

```
        try {
            vec.push_back(x);          // add to vector
        }
        catch (...) {
            // vec is unchanged
            return;
        }

        // lst and vec each have a new element with the value x
    }
```

Providing the strong guarantee in these cases is simple and cheap. It is also very useful because it provides a completely exception-safe way of adding elements. However, ***push_back*( )** isn't defined for associative containers – a ***map*** has no ***back*( )**. After all, the last element of an associative container is defined by the order relation rather than by position.

The guarantees for ***insert*( )** are a bit more complicated. The reason is that sometimes ***insert*( )** has to place an element in ''the middle'' of a container. This is no problem for a linked data structure, such as ***list*** or ***map***. However, if there is free reserved space in a ***vector***, the obvious implementation of ***vector<X>::insert*( )** copies the elements after the insertion point to make room. This is optimally efficient, but there is no simple way of restoring a ***vector*** if ***X***'s copy assignment or copy constructor throws an exception (see §E.8[10-11]). Consequently, ***vector*** provides a guarantee that is conditional upon element copy operations not throwing exceptions. However, ***list*** and ***map*** don't need such a condition; they can simply link in new elements after doing any necessary copying.

As an example, assume that ***X***'s copy assignment and copy constructor throw ***X::cannot_copy*** if they cannot successfully create a copy:

```
void f(list<X>& lst, vector<X>& vec, map<string,X>& m, const X& x, const string& s)
{
    try {
        lst.insert(lst.begin(),x);      // add to list
    }
    catch (...) {
        // lst is unchanged
        return;
    }

    try {
        vec.insert(vec.begin(),x);      // add to vector
    }
    catch (X::cannot_copy) {
        // oops: vec may or may not have a new element
        return;
    }
    catch (...) {
        // vec is unchanged
        return;
    }
```

```
        try {
            m.insert(make_pair(s,x));      // add to map
        }
        catch (...) {
            // m is unchanged
            return;
        }

        // lst and vec each have a new element with the value x
        // m has an element with the value (s,x)
    }
```

If *X::cannot_copy* is caught, a new element may or may not have been inserted into *vec*. If a new element was inserted, it will be an object in a valid state, but it is unspecified exactly what the value is. It is possible that after *X::cannot_copy*, some element will have been ''mysteriously'' duplicated (see §E.8[11]). Alternatively, *insert*( ) may be implemented so that it deletes some ''trailing'' elements to be certain that no invalid elements are left in a container.

Unfortunately, providing the strong guarantee for *vector*'s *insert*( ) without the caveat about exceptions thrown by copy operations is not feasible. The cost of completely protecting against an exception while moving elements in a *vector* would be significant compared to simply providing the basic guarantee in that case.

Element types with copy operations that can throw exceptions are not uncommon. Examples from the standard library itself are *vector<string>*, *vector< vector<double> >*, and *map<string, int>*.

The *list* and *vector* containers provide the same guarantees for *insert*( ) of single and multiple elements. The reason is simply that for *vector* and *list*, the same implementation strategies apply to both single-element and multiple-element *insert*( ). However, *map* provides the strong guarantee for single-element *insert*( ), but only the basic guarantee for multiple-element *insert*( ). A single-element *insert*( ) for *map* that provides the strong guarantee is easily implemented. However, the obvious strategy for implementing multiple-element *insert*( ) for a *map* is to insert the new elements one after another, and it is not easy to provide the strong guarantee for that. The problem with this is that there is no simple way of backing out of previous successful insertions if the insertion of an element fails.

If we want an insertion function that provides the strong guarantee that either every element was successfully added or the operation had no effect, we can build it by constructing a new container and then *swap*( ):

```
template<class C, class Iter>
void safe_insert(C& c, typename C::const_iterator i, Iter begin, Iter end)
{
    C tmp(c.begin(),i);                          // copy leading elements to temporary
    copy(begin,end,inserter(tmp,tmp.end()));     // copy new elements
    copy(i,c.end(),inserter(tmp,tmp.end()));     // copy trailing elements
    swap(c,tmp);
}
```

As ever, this code may misbehave if the element destructor throws an exception. However, if an element copy operation throws an exception, the argument container is unchanged.

E.4.3 Swap

Like copy constructors and assignments, *swap*( ) operations are essential to many standard algorithms and are often supplied by users. For example, *sort*( ) and *stable_sort*( ) typically reorder elements, using *swap*( ). Thus, if a *swap*( ) function throws an exception while exchanging values from a container, the container could be left with unchanged elements or a duplicate element rather than a pair of swapped elements.

Consider the obvious definition of the standard-library *swap*( ) function (§18.6.8):

```
template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Clearly, *swap*( ) doesn't throw an exception unless the element type's copy constructor or copy assignment does.

With one minor exception for associative containers, standard container *swap*( ) functions are guaranteed not to throw exceptions. Basically, containers are swapped by exchanging the data structures that act as handles for the elements (§13.5, §17.1.3). Since the elements themselves are not moved, element constructors and assignments are not invoked, so they don't get an opportunity to throw an exception. In addition, the standard guarantees that no standard-library *swap*( ) function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. This leaves only one potential source of exceptions: The comparison object in an associative container is copied as part of the handle. The only possible exception from a *swap*( ) of standard containers is the copy constructor and assignment of the container's comparison object (§17.1.4.1). Fortunately, comparison objects usually have trivial copy operations that do not have opportunities to throw exceptions.

A user-supplied *swap*( ) should be written to provide the same guarantees. This is relatively simple to do as long as one remembers to swap types represented as handles by swapping their handles, rather than slowly and elaborately copying the information referred to by the handles (§13.5, §16.3.9, §17.1.3).

E.4.4 Initialization and Iterators

Allocation of memory for elements and the initialization of such memory are fundamental parts of every container implementation (§E.3). Consequently, the standard algorithms for constructing objects in uninitialized memory – *uninitialized_fill*( ), *uninitialized_fill_n*( ), and *uninitialized_copy*( ) (§19.4.4) – are guaranteed to leave no constructed objects behind if they throw an exception. They provide the strong guarantee (§E.2). This sometimes involves destroying elements, so the requirement that destructors not throw exceptions is essential to these algorithms; see §E.8[14]. In addition, the iterators supplied as arguments to these algorithms are required to be well behaved. That is, they must be valid iterators, refer to valid sequences, and iterator operations (such as ++ and != and *) on a valid iterator are not allowed to throw exceptions.

Iterators are examples of objects that are copied freely by standard algorithms and operations on standard containers. Thus, copy constructors and copy assignments of iterators should not throw exceptions. In particular, the standard guarantees that no copy constructor or assignment operator of an iterator returned from a standard container throws an exception. For example, an iterator returned by *vector<T>::begin()* can be copied without fear of exceptions.

Note that ++ and -- on an iterator can throw exceptions. For example, an *istreambuf_iterator* (§19.2.6) could reasonably throw an exception to indicate an input error, and a range-checked iterator could throw an exception to indicate an attempt to move outside its valid range (§19.3). However, they cannot throw exceptions when moving an iterator from one element of a sequence to another, without violating the definition of ++ and -- on an iterator. Thus, *uninitialized_fill()*, *uninitialized_fill_n()*, and *uninitialized_copy()* assume that ++ and -- on their iterator arguments will not throw; if they do throw, either those ''iterators'' weren't iterators according to the standard, or the ''sequence'' specified by them wasn't a sequence. Again, the standard containers do not protect the user from the user's own undefined behavior (§E.2).

E.4.5 References to Elements

When a reference, a pointer, or an iterator to an element of a container is handed to some code, that code can corrupt the container by corrupting the element. For example:

```
void f(const X& x)
{
    list<X> lst;
    lst.push_back(x);
    list<X>::iterator i = lst.begin();
    *i = x;     // copy x into list
    // ...
}
```

If *x* is corrupted, *list*'s destructor may not be able to properly destroy *lst*. For example:

```
struct X {
    int* p;

    X() { p = new int; }
    ~X() { delete p; }
    // ...
};

void malicious()
{
    X x;
    x.p = reinterpret_cast<int*>(7);     // corrupt x
    f(x);                                // time bomb
}
```

When the execution reaches the end on *f()*, the *list<X>* destructor is called, and that will in turn invoke *X*'s destructor for the corrupted value. The effect of executing *delete p* when *p* isn't *0* and doesn't point to an *X* is undefined and could be an immediate crash. Alternatively, it

might leave the free store corrupted in a way that causes difficult-to-track problems much later on in an apparently unrelated part of a program.

This possibility of corruption should not stop people from manipulating container elements through references and iterators; it is often the simplest and most efficient way of doing things. However, it is wise to take extra care with such references into containers. When the integrity of a container is crucial, it might be worthwhile to offer safer alternatives to less experienced users. For example, we might provide an operation that checks the validity of a new element before copying it into an important container. Naturally, such checking can only be done with knowledge of the application types.

In general, if an element of a container is corrupted, subsequent operations on the container can fail in nasty ways. This is not particular to containers. Any object left in a bad state can cause subsequent failure.

E.4.6 Predicates

Many standard algorithms and many operations on standard containers rely on predicates that can be supplied by users. In particular, all associative containers depend on predicates for both lookup and insertion.

A predicate used by a standard container operation may throw an exception. In that case, every standard-library operation provides the basic guarantee, and some operations, such as *insert*( ) of a single element, provide the strong guarantee (§E.4.1). If a predicate throws an exception from an operation on a container, the resulting set of elements in the container may not be exactly what the user wanted, but it will be a set of valid elements. For example, if == throws an exception when invoked from *list*::*unique*( ) (§17.2.2.3), the user cannot assume that no duplicates are in the list. All the user can safely assume is that every element on the list is valid (see §E.5.3).

Fortunately, predicates rarely do anything that might throw an exception. However, user-defined <, ==, and != predicates must be taken into account when considering exception safety.

The comparison object of an associative container is copied as part of a *swap*( ) (§E.4.3). Consequently, it is a good idea to ensure that the copy operations of predicates that might be used as comparison objects do not throw exceptions.

E.5 The Rest of the Standard Library

The crucial issue in exception safety is to maintain the consistency of objects; that is, we must maintain the basic invariants for individual objects and the consistency of collections of objects. In the context of the standard library, the objects for which it is the most difficult to provide exception safety are the containers. From the point of view of exception safety, the rest of the standard library is less interesting. However, note that from the perspective of exception safety, a built-in array is a container that might be corrupted by an unsafe operation.

In general, standard-library functions throw only the exceptions that they are specified to throw, plus any thrown by user-supplied operations that they may call. In addition, any function that (directly or indirectly) allocates memory can throw an exception to indicate memory exhaustion (typically, *std*::*bad_alloc*).

E.5.1 Strings

The operations on ***string***s can throw a variety of exceptions. However, ***basic_string*** manipulates its characters through the functions provided by ***char_traits*** (§20.2), and these functions are not allowed to throw exceptions. That is, the ***char_traits*** supplied by the standard library do not throw exceptions, and no guarantees are made if an operation of a user-defined ***char_traits*** throws an exception. In particular, note that a type used as the element (character) type for a ***basic_string*** is not allowed to have a user-defined copy constructor or a user-defined copy assignment. This removes a significant potential source of exception throws.

A ***basic_string*** is very much like a standard container (§17.5, §20.3). In fact, its elements constitute a sequence that can be accessed using ***basic_string<Ch,Tr,A>::iterator***s and ***basic_string<Ch,Tr,A>::const_iterator***s. Consequently, a string implementation offers the basic guarantee (§E.2), and the guarantees for ***erase***( ), ***insert***( ), ***push_back***( ) and ***swap***( ) (§E.4.1) apply to ***basic_string***s. For example, ***basic_string<Ch,Tr,A>::push_back***( ) offers the strong guarantee.

E.5.2 Streams

If required to do so, iostream functions throw exceptions to signal state changes (§21.3.6). The semantics of this are well defined and pose no exception-safety problems. If a user-defined ***operator<<***( ) or ***operator>>***( ) throws an exception, it may appear to the user as if the iostream library threw an exception. However, such an exception will not affect the stream state (§21.3.3). Further operations on the stream may not find the expected data – because the previous operation threw an exception instead of completing normally – but the stream itself is uncorrupted. As ever after an I/O problem, a ***clear***( ) may be needed before doing further reads/writes (§21.3.3, §21.3.5).

Like ***basic_string***, the *iostream*s rely on ***char_traits*** to manipulate characters (§20.2.1, §E.5.1). Thus, an implementation can assume that operations on characters do not throw exceptions, and no guarantees are made if the user violates that assumption.

To allow for crucial optimizations, ***locale***s (§D.2) and ***facet***s (§D.3) are assumed not to throw exceptions. If they do, a stream using them could be corrupted. However, the most likely exception, a ***std::bad_cast*** from a ***use_facet*** (§D.3.1), can occur only in user-supplied code outside the standard stream implementation. At worst, this will produce incomplete output or cause a read to fail rather than corrupt the ***ostream*** (or ***istream***) itself.

E.5.3 Algorithms

Aside from ***uninitialized_copy***( ), ***uninitialized_fill***( ), and ***uninitialized_fill_n***( ) (§E.4.4), the standard offers only the basic guarantee (§E.2) for algorithms. That is, provided that user-supplied objects are well behaved, the algorithms will maintain all standard-library invariants and leak no resources. To avoid undefined behavior, user-supplied operations should always leave their operands in valid states, and destructors should not throw exceptions.

The algorithms themselves do not throw exceptions. Instead, they report errors and failures through their return values. For example, search algorithms generally return the end of a sequence to indicate ‘‘not found’’ (§18.2). Thus, exceptions thrown from a standard algorithm

must originate from a user-supplied operation. That is, the exception must come from an operation on an element – such as a predicate (§18.4), an assignment, or a ***swap*( )** – or from an allocator (§19.4).

If such an operation throws an exception, the algorithm terminates immediately, and it is up to the functions that invoked the algorithm to handle the exception. For some algorithms, it is possible for an exception to occur at a point where the container is not in a state that the user would consider good. For example, some sorting algorithms temporarily copy elements into a buffer and later put them back into the container. Such a ***sort*( )** might copy elements out of the container (planning to write them back in proper order later), overwrite them, and then throw an exception. From a user's point of view, the container was corrupted. However, all elements are in a valid state, so recovery should be reasonably straightforward.

Note that the standard algorithms access sequences through iterators. That is, the standard algorithms never operate on containers directly, only on elements in a container. The fact that a standard algorithm never directly adds or removes elements from a container simplifies the analysis of the impact of exceptions. Similarly, if a data structure is accessed only through iterators, pointers, and references to ***const*** (for example, through a ***const Rec****), it is usually trivial to verify that an exception has no undesired effects.

E.5.4 Valarray and Complex

The numeric functions do not explicitly throw exceptions (Chapter 22). However, ***valarray*** needs to allocate memory and thus might throw ***std*::*bad_alloc***. Furthermore, ***valarray*** or ***complex*** may be given an element type (scalar type) that throws exceptions. As ever, the standard library provides the basic guarantee (§E.2), but no specific guarantees are made about the effects of a computation terminated by an exception.

Like ***basic_string*** (§E.5.1), ***valarray*** and ***complex*** are allowed to assume that their template argument type does not have user-defined copy operations so that they can be bitwise copied. Typically, these standard-library numeric types are optimized for speed, assuming that their element type (scalar type) does not throw exceptions.

E.5.5 The C Standard Library

A standard-library operation without an exception specification may throw exceptions in an implementation-defined manner. However, functions from the standard C library do not throw exceptions unless they take a function argument that does. After all, these functions are shared with C, and C doesn't have exceptions. An implementation may declare a standard C function with an empty *exception-specification*, ***throw*( )**, to help the compiler generate better code.

Functions such as ***qsort*( )** and ***bsearch*( )** (§18.11) take a pointer to function as argument. They can therefore throw an exception if their arguments can. The basic guarantee (§E.2) covers these functions.

E.6 Implications for Library Users

One way to look at exception safety in the context of the standard library is that we have no problems unless we create them for ourselves: The library will function correctly as long as user-supplied operations meet the standard library's basic requirements (§E.2). In particular, no exception thrown by a standard container operation will cause memory leaks from containers or leave a container in an invalid state. Thus, the problem for the library user becomes: How can I define my types so that they don't cause undefined behavior or leak resources?

The basic rules are:

[1] When updating an object, don't destroy its old representation before a new representation is completely constructed and can replace the old one without risk of exceptions. For example, see the implementations of *vector*::*operator*=( ), *safe_assign*( ), and *vector*::*push_back*( ) in §E.3.

[2] Before throwing an exception, release every resource acquired that is not owned by some (other) object.

[2a] The ''resource acquisition is initialization'' technique (§14.4) and the language rule that partially constructed objects are destroyed to the extent that they were constructed (§14.4.1) can be most helpful here. For example, see *leak*( ) in §E.2.

[2b] The *uninitialized_copy*( ) algorithm and its cousins provide automatic release of resources in case of failure to complete construction of a set of objects (§E.4.4).

[3] Before throwing an exception, make sure that every operand is in a valid state. That is, leave each object in a state that allows it to be accessed and destroyed without causing undefined behavior or an exception to be thrown from a destructor. For example, see *vector*'s assignment in §E.3.2.

[3a] Note that constructors are special in that when an exception is thrown from a constructor, no object is left behind to be destroyed later. This implies that we don't have to establish an invariant and that we must be sure to release all resources acquired during a failed construction before throwing an exception.

[3b] Note that destructors are special in that an exception thrown from a destructor almost certainly leads to violation of invariants and/or calls to *terminate*( ).

In practice, it can be surprisingly difficult to follow these rules. The primary reason is that exceptions can be thrown from places where people don't expect them. A good example is *std*::*bad_alloc*. Every function that directly or indirectly uses *new* or an *allocator* to acquire memory can throw *bad_alloc*. In some programs, we can solve this particular problem by not running out of memory. However, for programs that are meant to run for a long time or to accept arbitrary amounts of input, we must expect to handle various failures to acquire resources. Thus, we must assume every function capable of throwing an exception until we have proved otherwise.

One simple way to try to avoid surprises is to use containers of elements that do not throw exceptions (such as containers of pointers and containers of simple concrete types) or linked containers (such as *list*) that provide the strong guarantee (§E.4). Another, complementary, approach is to rely primarily on operations, such as *push_back*( ), that offer the strong guarantee that an operation either succeeds or has no effect (§E.2). However, these approaches are by themselves insufficient to avoid resource leaks and can lead to an ad hoc, overly restrictive, and

pessimistic approach to error handling and recovery. For example, a ***vector<T*>*** is trivially exception safe if operations on ***T*** don't throw exceptions. However, unless the objects pointed to are deleted somewhere, an exception from the ***vector*** will lead to a resource leak. Thus, introducing a ***Handle*** class to deal with deallocation (§25.7) and using ***vector***<Handle<T> > rather than the plain ***vector<T*>*** will probably improve the resilience of the code.

When writing new code, it is possible to take a more systematic approach and make sure that every resource is represented by a class with an invariant that provides the basic guarantee (§E.2). Given that, it becomes feasible to identify the critical objects in an application and provide roll-back semantics (that is, the strong guarantee – possibly under some specific conditions) for operations on such objects.

Most applications contain data structures and code that are not written with exception safety in mind. Where necessary, such code can be fitted into an exception-safe framework by either verifying that it doesn't throw exceptions (as was the case for the C standard library; §E.5.5) or through the use of interface classes for which the exception behavior and resource management can be precisely specified.

When designing types intended for use in an exception-safe environment, we must pay special attention to the operations used by the standard library: constructors, destructors, assignments, comparisons, swap functions, functions used as predicates, and operations on iterators. This is best done by defining a class invariant that can be simply established by all constructors. Sometimes, we must design our class invariants so that we can put an object into a state where it can be destroyed even when an operation suffers a failure at an ''inconvenient'' point. Ideally, that state isn't an artifact defined simply to aid exception handling, but a state that follows naturally from the semantics of the type (§E.3.5).

When considering exception safety, the emphasis should be on defining valid states for objects (invariants) and on proper release of resources. It is therefore important to represent resources directly as classes. The ***vector_base*** (§E.3.2) is a simple example of this. The constructors for such resource classes acquire lower-level resources (such as the raw memory for ***vector_base***) and establish invariants (such as the proper initialization of the pointers of a ***vector_base***). The destructors of such classes implicitly free lower-level resources. The rules for partial construction (§14.4.1) and the ''resource acquisition is initialization'' technique (§14.4) support this way of handling resources.

A well-written constructor establishes the class invariant for an object (§24.3.7.1). That is, the constructor gives the object a value that allows subsequent operations to be written simply and to complete successfully. This implies that a constructor often needs to acquire resources. If that cannot be done, the constructor can throw an exception so that we can deal with that problem before an object is created. This approach is directly supported by the language and the standard library (§E.3.5).

The requirement to release resources and to place operands in valid states before throwing an exception means that the burden of exception handling is shared among the function throwing, the functions on the call chain to the handler, and the handler. Throwing an exception does not make handling an error ''somebody else's problem.'' It is the obligation of functions throwing or passing along an exception to release resources that they own and to put operands in consistent states. Unless they do that, an exception handler can do little more than try to terminate gracefully.

E.7 Advice

[1] Be clear about what degree of exception safety you want; §E.2.
[2] Exception safety should be part of an overall strategy for fault tolerance; §E.2.
[3] Provide the basic guarantee for all classes. That is, maintain an invariant, and don't leak resources; §E.2, §E.3.2, §E.4.
[4] Where possible and affordable, provide the strong guarantee that an operation either succeeds or leaves all operands unchanged; §E.2, §E.3.
[5] Don't throw an exception from a destructor; §E.2, §E.3.2, §E.4.
[6] Don't throw an exception from an iterator navigating a valid sequence; §E.4.1, §E.4.4.
[7] Exception safety involves careful examination of individual operations; §E.3.
[8] Design templates to be transparent to exceptions; §E.3.1.
[9] Prefer the constructor approach to resource requisition to using ***init***( ) functions; §E.3.5.
[10] Define an invariant for a class to make it clear what is a valid state; §E.2, §E.6.
[11] Make sure that an object can always be put into a valid state without fear of an exception being thrown; §E.3.2, §E.6.
[12] Keep invariants simple; §E.3.5.
[13] Leave all operands in valid states before throwing an exception; §E.2, §E.6.
[14] Avoid resource leaks; §E.2, §E.3.1, §E.6.
[15] Represent resources directly; §E.3.2, §E.6.
[16] Remember that ***swap***( ) can sometimes be an alternative to copying elements; §E.3.3.
[17] Where possible, rely on ordering of operations rather than on explicit use of *try-block*s; §E.3.4.
[18] Don't destroy ''old'' information until its replacement has been safely produced; §E.3.3, §E.6.
[19] Rely on the ''resource acquisition is initialization'' technique; §E.3, §E.3.2, §E.6.
[20] Make sure that comparison operations for associative containers can be copied; §E.3.3.
[21] Identify critical data structures and provide them with operations that provide the strong guarantee; §E.6

E.8 Exercises

1. (∗1) List all exceptions that could possibly be thrown from ***f***( ) in §E.1.
2. (∗1) Answer the questions after the example in §E.1.
3. (∗1) Define a class ***Tester*** that occasionally throws exceptions from basic operations, such as copy constructors. Use ***Tester*** to test your standard-library containers.
4. (∗1) Find the error in the ''messy'' version of ***vector***'s constructor (§E.3.1), and write a program to get it to crash. Hint: First implement ***vector***'s destructor.
5. (∗2) Implement a simple list providing the basic guarantee. Be very specific about what the list requires of its users to provide the guarantee.
6. (∗3) Implement a simple list providing the strong guarantee. Carefully test this list. Give an argument why people should believe it to be safe.
7. (∗2.5) Reimplement ***String*** from §11.12 to be as safe as a standard container.
8. (∗2) Compare the run time of the various versions of ***vector***'s assignment and

***safe_assign*( )** (§E.3.3).

9. (∗1.5) Copy an allocator without using an assignment operator (as needed to improve ***operator*=( )** in §E.3.3).
10. (∗2) Add single-element and multiple-element ***erase*( )** and ***insert*( )** that provide the basic guarantee to ***vector*** (§E.3.2).
11. (∗2) Add single-element and multiple-element ***erase*( )** and ***insert*( )** that provide the strong guarantee to ***vector*** (§E.3.2). Compare the cost and complexity of these solutions to the solutions to exercise 10.
12. (∗2) Write a ***safe_insert*( )** (§E.4.2) that inserts elements into the existing ***vector*** (rather than copying to a temporary). What constraints do you have to impose on operations?
13. (∗2) Write a ***safe_insert*( )** (§E.4.2) that inserts elements into the existing ***map*** (rather than copying to a temporary). What constraints do you have to impose on operations?
14. (∗2.5) Compare the size, complexity, and performance of the ***safe_insert*( )** functions from exercises 12 and 13 to the ***safe_insert*( )** from §E.4.2.
15. (∗2.5) Write a better (simpler and faster) ***safe_insert*( )** for associative containers only. Use traits to write a ***safe_insert*( )** that automatically selects the optimal ***safe_insert*( )** for a container. Hint: §19.2.3.
16. (∗2.5) Try to rewrite ***uninitialized_fill*( )** (§19.4.4, §E.3.1) to handle destructors that throw exceptions. Is that possible? If so, at what cost? If not, why not?
17. (∗2.5) Try to rewrite ***uninitialized_fill*( )** (§19.4.4, §E.3.1) to handle iterators that throw exceptions for `++` and `--`. Is that possible? If so, at what cost? If not, why not?
18. (∗3) Take a container from a library different from the standard library. Examine its documentation to see what exception-safety guarantees it provides. Do some tests to see how resilient it is against exceptions thrown by memory allocation and user-supplied code. Compare it to a corresponding standard-library container.
19. (∗3) Try to optimize the ***vector*** from §E.3 by disregarding the possibility of exceptions. For example, remove all *try-block*s. Compare the performance against the version from §E.3 and against a standard-library ***vector*** implementation. Also, compare the size and the complexity of the source code of these different ***vector***s.
20. (∗1) Define invariants for ***vector*** (§E.3) with and without the possibility of ***v==0*** (§E.3.5) .
21. (∗2.5) Read the source of an implementation of ***vector***. What guarantees are implemented for assignment, multi-element ***insert*( )**, and ***resize*( )**?
22. (∗3) Write a version of ***hash_map*** (§17.6) that is as safe as a standard container.

I

Index

Is there another word for synonym?
– anon

162
** 263
-1 831
->* 853
.* 853
16-bit character 580
7-bit char 580
8-bit char 580
  bitset 494
'
 and [] 838
 operator 123
 predefined 264
 prohibiting 264
!
 for basic_ios 616
 logical_not 516
 valarray 664
!=
 bitset 494
 complex 680
 generated 468
 iterator 551
 not_equal_to 516
 string 591
 valarray 667
#, preprocessing directive 813
$ character 81
%
 modulus 517
 valarray 667
%: digraph 829
%:%: digraph 829
%=, valarray 664
%> digraph 829
&
 bitset 495
 bitwise and operator 124
 predefined 264
 prohibiting 264
 valarray 667
&&
 logical and operator 123
 logical_and 516
 valarray 667
&=
 of bitset 494
 valarray 664
', character literal 73
*
 and [], -> and 290
 complex 680
 iterator 551
 multiplies 517
 valarray 667
*=
 complex 679
 valarray 664
+
 complex 680

iterator 551
plus 517
string 593
user-defined 265
user-defined operator 281
valarray 667
++
increment operator 125
iterator 551
user-defined operator 264, 291
+=
advance() and 551
complex 679
iterator 551
operator 109
string 592
user-defined operator 264, 268, 281
valarray 664
-
complex 680
distance() and 551, 554
iterator 551
minus 517
negate 517
valarray 664, 667
--
decrement operator 125
iterator 551
user-defined operator 291
-=
complex 679
iterator 551
operator 109
valarray 664
->
and * and [] 290
iterator 551
member access operator 102
user-defined operator 289
->*, pointer to member 418
.
floating-point 74
member access operator 101
.*, pointer to member 418
..., ellipsis 154
/
complex 680
divides 517
valarray 667
/* comment 161
//
comment 10
difference from C 816
/=
complex 679
valarray 664
::
and virtual function, operator 312
explicit qualification 847
namespace and 169
operator 305
scope resolution operator 82, 228
::*, pointer to member 418
:> digraph 829
<
comparison 467
iterator 551
less 516
string 591
template syntax 811
valarray 667
vector 457
<% digraph 829
<: digraph 829
<<
bitset 494
bitset 495
complex 680
for output why 607
inserter 608
of char 611
of complex 612
of pointer to function 631
of streambuf 642
ostream 609
output cout 46
output operator 607
precedence 608
put to 607
string 598
valarray 667
virtual 612
<<=
of bitset 494
valarray 664
<=
generated 468
iterator 551
less_equal 516
string 591
valarray 667
=
map 484
predefined 264
prohibiting 264
string 587
user-defined operator 281
valarray 663
vector 447
==
bitset 494
complex 680

equal_to 516
equality without 468
iterator 551
string 591
user-defined 534
valarray 667
vector 457
>
and >> 812
generated 468
greater 516
iterator 551
string 591
valarray 667
>=
generated 468
greater_equal 516
iterator 551
string 591
valarray 667
>>
> and 812
bitset 494
bitset 495
char 615
complex 680
extractor 608
get from 607
input cin 50, 112
istream 614
of complex 621
of pointer to function 632
string 598
valarray 667
>>=
of bitset 494
valarray 664
?:, arithmetic-if 134
[]
, and 838
-> and * and 290
and insert() 488
bitset 494
design of 295
iterator 551
map 482
of vector 445
on string 584
valarray 663
\
backslash 830
escape character 73, 830
\', single quote 830
^
bitset 495
bitwise exclusive or operator 124
valarray 667
^=
of bitset 494
valarray 664
_ character 81
|
bitset 495
bitwise or operator 124
valarray 667
|=
of bitset 494
valarray 664
||
logical or operator 123
logical_or 516
valarray 667
~, valarray 664
0
constant-expression 835
false and 71
null integer value 830
string and 587
zero null 88
-1 and size_t 448
1, true and 71

A

Aarhus 536
abort() 218, 380
abs() 660–661, 680
valarray 667
abstract
and concrete type 771
class 708
class 313
class and design 318
class, class hierarchy and 324
iterator 435
node class 774
type 34, 767, 769
abstraction
classes and 733
data 30
abstraction, late 437
abstraction, levels of 733
access 278
checked 445
control 225, 402
control and base class 405
control and multiple-inheritance 406
control, cast and 414
control, run-time 785
control, *using-declaration* and 407
element 445
operator, design of 295

to base 850
to member 849
to member class 851
unchecked 445
accumulate() 682
acos(), valarray 667
acquisition, resource 364
action 776
Ada 10, 725
adapter
member function 520
pointer to function 521
adapters, container 469
add element to sequence 529
adding
to container 555
to sequence 555
to standard library 434
address of element 454
addressing, united of 88
adjacent_difference() 684
adjacent_find() 525
adjustfield 626, 630
adoption of C++, gradual 718
advance() and += 551
aims
and means 694
design 700
Algol68 10
algorithm 56
C-style function and 522
and member function 520
and polymorphic object 63
and polymorphism 520
and sequence 508
and string 584
container and 507
conventions 508
design 510
exception container 566
generalized numeric 682
generic 41
modifying sequence 529
nonmodifying sequence 523
on array 544
return value 508
summary 509
<algorithm> 432, 509
algorithms, standard library 64
alias, namespace 178
alignment 102
all, catch 362
allocate array 128
allocate() 567
allocation
C-style 577
and de-allocation 127
unit of 88
allocator 567
Pool_alloc 572
general 573
nothrow 823
use of 568
user-defined 570
allocator 567
allocator_type 443, 480
alternative
design 710
error handling 192, 355
implementation 320
interface 173
return 357
to macro 161
ambiguity
dynamic_cast and 412
resolution, multiple-inheritance 391
ambiguous type conversion 276
ambition 693
analogy
bridge 724
car factory 698
plug 728
proof by 692
units 728
analysis
design and 696
error 711
experimentation and 710
method, choosing an 696
stage 697
and C-style string, string 579
and
keyword 829
operator &, bitwise 124
operator &&, logical 123
and_eq keyword 829
Annemarie 92
anomaly, constructor and destructor 245
anonymous union 841
ANSI
C 13
C++ 11
any() 494
app append to file 639
append to file, app 639
append(), string 592
application framework 731, 786
apply() to valarray 664
architecture 696
arg() 680
argc, main() argv 117
argument

array 147
command line 117
deducing template 335, 855
default 153
depend on template 861
explicit template 335
function template 335
passing 283
passing, function 145
reference 98
template 331
type check, function 145
type conversion, function 145
type, difference from C 817
types, virtual function 310
undeclared 154
value, example of default 227
argv argc, main() 117
arithmetic
conversions, usual 122
conversions, usual 836
function object 517
mixed-mode 268
pointer 88, 93, 125
type 70
vector 65, 662
arithmetic-if ?: 134
array 26, 88
algorithm on 544
allocate 128
argument 147
array of 837
as container 496
assignment 92
associative 286, 480
by string, initialization of 89
deallocate 128
delete 250
element, constructor for 250
element object 244
initializer 89
initializer, difference from C 818
layout 669
multidimensional 668, 677, 836
new and 423
of array 837
of objects 250
passing multidimensional 839
pointer and 91, 147
string and 589
valarray and 663
valarray and vector and 662
arrays, numeric 662
ASCII 580, 829
character set 73, 601
asin() 660
valarray 667
asm assembler 806
assembler 8, 10
asm 806
Assert() 750
assert() 750
<assert.h> 432
assertion checking 750
assign()
char_traits 581
string 588
vector 447
assignment
and initialization 283
array 92
copy 246, 283
function call and 99
map 484
of class object 245
operator 110, 268
string 587
to self 246
valarray 663
Assoc example 286
associative
array 286, 480
array – see map
container 480
container, sequence and 461
associativity of operator 121
asynchronous event 357
at() 53, 445
on string 585
out_of_range and 385
atan() 660
valarray 667
atan2() 660
valarray 667
ate 639
atexit()
and destructors 218
and exception 382
atof() 600
atoi() 589, 600
atol() 600
AT&T Bell Laboratories 11
auto 843
automatic
garbage collection 247, 844
memory 843
memory management 844
object 244
auto_ptr 367

B

\b, backspace 830
back() 445
 of queue 476
back_inserter() 57, 555
back_insert_iterator 555
backslash \ 830
backspace \b 830
bad() 616
bad_alloc 129
 and new 384
 exception 576
 missing 823
badbit 617
bad_cast 410
 and dynamic_cast 384
bad_exception 378, 384
bad_typeid and typeid() 384
balance 695
base
 access to 850
 and derived class 39, 737
 class 303
 class, access control and 405
 class, initialization of 306
 class, overriding from virtual 401
 class, private 743
 class, private member of 305
 class, protected 743
 class, replicated 394
 class, universal 438
 class, virtual 396
 member or 740
 override private 738
 private 405, 742
 protected 319, 405
basefield 626–627
Basic 725
basic_filebuf, class 648
basic_ios 608, 616, 622, 629
 ! for 616
 format state 606
 stream state 606
basic_iostream 637
 formatting 606
basic_istream 613
basic_ofstream 638
basic_ostream 608–609
basic_streambuf 645
 buffering 606
basicstring
 begin() 584
 end() 584
 rbegin() 584
 rend() 584
basic_string 582
 const_iterator 583
 const_pointer 583
 const_reference 583
 const_reverse_iterator 583
 difference_type 583
 iterator 583
 member type 582
 pointer 583
 reference 583
 reverse_iterator 583
 size_type 583
 traits_type 583
 value_type 583
basic_stringstream 640
BCPL 10
before() 415
beg, seekdir and
begin() 54, 481
 basicstring 584
 iterator 444
behavior, undefined 828
Bell Laboratories, AT&T 11
Bi 511
bibliography, design 719
bidirectional iterator 550
bidirectional_iterator_tag 553
big-O notation 464
binary
 mode, binary 639
 operator, user-defined 263
 search 540, 546
binary binary mode 639
binary_function 515
binary_negate 518
 not2() and 522
binary_search() 540
bind1st() 518
 and binder1st 520
bind2nd() 518
binder1st 518
 bind1st() and 520
binder2nd 518–519
binding
 name 860
 strength, operator 121, 607
BinOp 511
BinPred 511
bit
 field 125, 840
 field, bitset and 492
 pattern 73
 position 492
 reference to 492
 vector 124
bitand keyword 829
bitor keyword 829

bits
 in char 658
 in float 658
 in int 658
<bitset> 431
bitset 492
 494
 != 494
 & 495
 &= of 494
 << 495
 << 494
 <<= of 494
 == 494
 >> 495
 >> 494
 >>= of 494
 [] 494
 ^ 495
 ^= of 494
 and bit field 492
 and enum 492
 and set 492
 and vector<bool> 492
 constructor 493
 flip() 494
 input 495
 operation 494
 output 495
 reset() 494
 set() 494
 | 495
 |= of 494
bitset(), invalid_argument and 385
bitwise
 and operator & 124
 exclusive or operator ^ 124
 logical operators 124
 or operator | 124
blackboard as design tool 703
BLAS 668
bool 71
 conversion to 835
 input of 615
 output of 610
 vector of 458
boolalpha 610, 625
boolalpha() 633
break 109, 116
 case and 134
 statement 116
bridge analogy 724
bsearch() 546
buffer
 memory 575
 ostream and 642
 position in 642
Buffer 331, 335
 example 738
buffering 642
 I/O 645
 basic_streambuf 606
built-in
 feature vs technique 43
 type 70
 type, constructor for 131
 type, input of 614
 type, output of 609
 type, user-defined operator and 265
by
 reference, catch 360
 value, catch 359
byte 76

C

C
 //, difference from 816
 ANSI 13
 C++ and 13–14, 199
 and C++ 7
 and C++ compatibility 816
 and C++, mixing 719
 and exception 383
 and, learning 7
 argument type, difference from 817
 array initializer, difference from 818
 declaration and definition, difference from 818
 difference from 816
 enum, difference from 817
 function call, difference from 816
 function definition, difference from 817
 initialization and goto, difference from 818
 input and output 651
 int implicit, difference from 817
 jump past initialization, difference from 818
 linkage to 205
 macros, difference from 818
 programmer 14
 scope, difference from 816
 sizeof, difference from 816
 standard library 599
 struct name, difference from 818
 struct scope, difference from 818
 void* assignment, difference from 818
 with Classes 10
 with classes 686
C++ 21
.c file 202
cache example 232
calculator example 107, 165, 190, 208
call

by reference 98, 146
by value 146
function 145
of destructor, explicit 256
callback 371
stream 650
call-by reference 282
callC() example 384
call_from_C() example 384
calloc() 577
capacity() 455
car factory analogy 698
Car example 772
card, CRC 703
c_array 496
carriage return \r 830
CASE 711, 725, 730
case and break 134
case-sensitive compare 591
<cassert> 432
cast
C-style 131
and access control 414
cross 408
deprecated C-style 819
down 408
up 408
casting away const 414
catch all 362
catch 186, 361
all 362
by reference 360
by value 359
every exception 54
catch(...) 54
category, iterator 553
<cctype> 432, 601
ceil() 660
cerr 609
and clog 624
initialization 637
<cerrno> 432
<cfloat> 433, 660
C-function and exception 382
C++ 10
ANSI 11
C and 7
ISO 11
and C 13–14, 199
compatibility, C and 816
design of 7, 10
feature summary 819
functional decomposition and 726
gradual adoption of 718
gradual approach to learning 7
introducing 718
large programs and 9
learning 6, 718, 820
library, first 686
misuse of 725
mixing C and 719
procedural programming and 725
programmer 14
properties of 724
standardization 11
style subscript 674
teaching and 12
use of 12
change 700
incremental 684
response to 698
size of sequence 529
changing interface 774
char 73, 76
7-bit 580
8-bit 580
<< of 611
>> 615
bits in 658
character type 71
get() 620
input 618
input of 615
output of 610
signed 831
unsigned 831
char*, specialization and 344
character 580
$ 81
16-bit 580
\, escape 73, 830
_ 81
buffer, streambuf and 642
classification, wide 601
literal ' 73
name, universal 831
national 829
set 829
set, ASCII 73, 601
set, large 831
set, restricted 829
special 830
string 432
traits 580
type 580
type char 71
value of 580
characters in name 81
CHAR_BIT 660
char_traits 580
assign() 581
char_type 580

compare() 581
copy() 581
eof() 581
eq() 581
eq_int_type() 581
find() 581
get_state() 581
int_type() 581
length() 581
lt() 581
move() 581
not_eof() 581
off_type 581
pos_type 581
state_type 581
to_char_type() 581
to_int_type() 581
char_traits<char> 580
char_traits<wchart> 581
char_type 608
char_traits 580
check, range 445, 561
checked
access 445
iterator 561
pointer 291
Checked example 565
Checked_iter example 561
checking
assertion 750
for, wild pointer 722
invariant 749
checking, late, missing 823
checking
of *exception-specification* 376
range 275, 781
choosing
a design method 696
an analysis method 696
cin 614
>>, input 50, 112
cout and 624
initialization 637
value of 276
circle and ellipse 703
class
abstract 313
and design, abstract 318
and type 724
base 303
basic_filebuf 648
concept and 301
constructor for derived 306
conversion of pointer to 304
derived 15, 303
destructor for derived 306
forward reference to 278
friend 279
function 776
handle 782
hierarchy 15, 307, 734
hierarchy and abstract class 324
hierarchy and template 345
hierarchy design 314
hierarchy, reorganization of 707
initialization of base 306
interface 778
member, constructor for 247
member of derived 304
member, private 225
member, public 225
node 772
object, assignment of 245
operations, set of 237
overriding from virtual base 401
pointer to 304
private base 743
private member of base 305
protected base 743
storage 244
use of 725
class 16, 32
abstract 708
abstract node 774
access to member 851
and concept 223
base and derived 39, 737
concrete 236, 241, 766
concrete node 774
declaration 225
definition 225
free-standing 732
function-like 514
helper 293
hierarchy 38, 389
hierarchy navigation 411
kind of 765
lattice 389
leaf 774
member 293
namespace and 849
nested 293
not a 705
random number 685
string 292
struct and 234
template and 348
union and 843
universal base 438
user-defined type 224
classes
and abstraction 733

and concepts 732
and real-world 734
design and 732
find the 702
finding the 734
stream 637
use of 733
classic() locale 649
classification 703
cleanup, initialization and 364
clear goal 698
clear() , 487, 616
failure and 385
<climits> 433, 660
<clocale> 433, 650
Clock example 398
clog 609
cerr and 624
initialization 637
clone 424
clone() 426
close() 639
closing
of file 638
of stream 639
closure 676
cloud example 700
Clu 10
Club_eq 516
<cmath> 434, 660
Cmp 339, 511
Cobol 725
code
bloat, curbing 342
uniformity of 767
coders and designers 695
coercion 267
collaboration, design 708
collating sequence 338
collector,
conservative 846
copying 846
comma and subscripting 838
command line argument 117
comment 138
/* 161
// 10
common
code and constructor 624
code and destructor 624
commonality 301
communication 694–695, 717
compare, case-sensitive 591
compare()
char_traits 581
string 590
comparison
< 467
default 467
equality and 457
in map 484
requirement 467
string 590
user-supplied 467
compatibility 13
C and C++ 816
compilation
separate 27, 198
template separate 351
unit of 197
compile time, header and 211
compile-time polymorphism 347
compl keyword 829
complete encapsulation 283
complex 64, 267
!= 680
* 680
*= 679
+ 680
+= 679
- 680
-= 679
/ 680
/= 679
<< 680
<< of 612
== 680
>> 680
>> of 621
and complex<> 680
conversion 681
cos() 680
cosh() 680
expr() 680
input 680
log() 680
log10() 680
mathematical functions 680
operations 679
output 680
pow() 680
sin() 680
sinh() 680
sqrt() 680
tanh() 680
complex<>, complex and 680
complexity divide and conquer 693
component 701, 755
standard 698, 714
composite operator 268
composition of namespace 181
compositor 677

computation, numerical 64
concatenation, `string` 592–593
concept 15
  and class 301
  `class` and 223
  independent 327
concepts, classes and 732
concrete
  `class` 236, 241, 766
  class, derive from 780
  node `class` 774
  type 33, 236, 766–767
  type, abstract and 771
  type and derivation 768
  type, problems with 37
  type, reuse of 241
  type, reuse of 768
condition 753
  declaration in 135
conditional expression 134
`conj()` 680
connection between input and output 623
`const` 94
  C-style string and 90
  and linkage 199
  and overloading 600
  casting away 414
  function, inspector 706
  iterator 443
  member 249
  member function 229
  physical and logical 231
  pointer 96
  pointer to 96
constant
  expression 833
  in-class definition of 249
  member 249
  time 464
constant-expression `0` 835
`const_cast` 131, 232
`const_iterator` 54, 443, 480
  `basic_string` 583
`const_mem_fun1_ref_t` 518, 521
`const_mem_fun1_t` 518, 521
`const_mem_fun_ref_t` 518, 521
`const_mem_fun_t` 518, 521
`const_pointer` 443
  `basic_string` 583
`const_reference` 443, 480
  `basic_string` 583
`const_reverse_iterator` 443, 480
  `basic_string` 583
`construct()` 567
construction
  and destruction 244
  and destruction, order or 414
  order of 248, 252
  `valarray` 662
constructor 32–33, 226, 706
  and C-style initialization 270
  and conversion 272
  and destructor 242, 246–247
  and destructor anomaly 245
  and initializer list 270
  and `template`, copy 348
  and type conversion 269, 275
  and `union` 257
  and `virtual` base 397
  `bitset` 493
  common code and 624
  copy 246, 283
  default 243
  default copy 271
  exception in 367
  exceptions in 371
  `explicit` 284
  for array element 250
  for built-in type 131
  for class member 247
  for derived class 306
  for free store object 246
  for global variable 252
  for local variable 245
  `map` 484
  pointer to 424
  `string` 585
  `vector` 447
  `virtual` 323, 424
constructors, exceptions and 366
`cont` iterator 508
container 40, 52
  STL 441
  Simula-style 438
  Smalltalk-style 438
  adapters 469
  adding to 555
  algorithm, exception 566
  and algorithm 507
  and iterator 435, 444
  and polymorphism 520
  array as 496
  associative 480
container, based 438
container
  design 434, 441
  implementation of 465
  input into 451
  intrusive 438
  iterator 464
  kind of 461
  memory management 455, 567

operation on 464
optimal 435
representation of 465
sequence and 512
standard library 56
standard-library 442
string as 491
summary 464
user-defined 497
valarray as 492
containers 431
containment 738
and inheritance 740
context
of template definition 860
of template instantiation 860
continue 116
statement 116
contravariance 420
control, format 625
controlled statement 136
convenience
and orthogonality 431
vs. safety 847
conventions
algorithm 508
lexical 794
national 649
conversion 706
ambiguous type 276
complex 681
constructor and 272
constructor and type 269, 275
explicit 284
floating-point 834
implicit 275, 281, 284
implicit type 76, 276, 833
integer 834
of pointer to class 304
of string, implicit 590
operator, type 275
pointer 834
signed unsigned integer 834
string 589
to bool 835
to floating-point 835
to integer type 835
to integral 835
undefined enum 77
user-defined 347
user-defined pointer 349
user-defined type 267, 281
conversions 747
usual arithmetic 122
conversions, usual, arithmetic 836
cookbook method 692
copy 229, 245, 271
assignment 246, 283
constructor 246, 283
constructor and template 348
constructor, default 271
delayed 295
memberwise 283
of exception 362
requirement 466
copy() 42, 529, 589
char_traits 581
_copy suffix 533
copy_backward() 529
copyfmt() 627
copyfmt_event 651
copyfmt_event, copyfmt() 651
copy_if() not standard 530
copying, elimination of 675
copy-on-write 295
cos() 660
complex 680
valarray 667
cosh() 660
complex 680
valarray 667
cost of exception 381
count() 57, 494, 526
in map 485
count_if() 62, 526
counting, reference 783
coupling, efficiency and 768
cout 609
<<, output 46
and cin 624
initialization 637
Cowboy example 778
__cplusplus 206
CRC card 703
create dependency 745
creation
localization of object 322
object 242
criteria
sorting 534
standard library 430
cross cast 408
<csetjmp> 433
cshift() 664
<csignal> 433
<cstdarg> 433
<cstdio> 202, 432
<cstdlib> 219, 432, 434, 546, 577, 600, 661
c_str() 589
<cstring> 432, 577, 599
C-style
allocation 577

cast 131
cast, deprecated 819
error handling 661
function and algorithm 522
initialization, constructor and 270
string and `const` 90
string, `string` and 579
string, `string` and 589
`<ctime>` 431, 433
`<ctype.h>` 432, 601
`cur`, `seekdir` and
curbing code bloat 342
Currying 520
`<cwchar>` 432
`<cwtype>` 432, 601
`<cwtype.h>` 601
cycle, development 698

D

data
abstraction 30
abstraction vs inheritance 727
member, pointer to 853
per-object 573
per-type 573
`data()` 589
date, format of 649
`Date` example 236
`DBL_MINEXP` 660
deallocate array 128
`deallocate()` 567
de-allocation, allocation and 127
debugging 226
`dec` 626–627, 634
decimal 73
output 626
decision, delaying 706
declaration 23, 78–79
and definition, difference from C 818
and definition, `namespace` member 167
`class` 225
`friend` 279
function 143
in condition 135
in `for` statement 137
of member class, forward 293
point of 82
declaration 803
declarations, keeping consistent 201
declarator operator 80
declarator 807
decomposition, functional 725
decrement
increment and 291
operator `--` 125
deducing `template` argument 335, 855
default
argument 153
argument value, example of 227
comparison 467
constructor 243
copy constructor 271
`template` argument 340, 824
value 239
value, supplying 500
`default` 109
`#define` 160
definition 78
`class` 225
context of `template` 860
difference from C declaration and 818
function 144
in-class 235
`namespace` member declaration and 167
of constant, in-class 249
of `virtual` function 310
point of 861
using-directive and 180
delayed copy 295
delaying decision 706
delegation 290
delete
element from sequence 529, 534
from `hash_map` 501
`delete`
and garbage collection 845
array 250
`delete[]` and 250
operator 127
size and 421
`delete()`, `operator` 129, 576
`delete[]` 128
and `delete` 250
`delete[]()`, `operator` 423, 576
`delete_ptr()` example 531
`denorm_min()` 659
depend on `template` argument 861
dependencies 724
dependency 15
create 745
inheritance 737
minimize 173
use 745
deprecated
C-style cast 819
`static` 818
`<deque>` 431
`deque`, double-ended queue 474
derivation, concrete type and 768
derive
from concrete class 780

without virtual 780
derived
and friend 852
class 15, 303
class, base and 39, 737
class, constructor for 306
class, destructor for 306
class, member of 304
exceptions 359
design 696
I/O 605
abstract class and 318
aims 700
algorithm 510
alternative 710
and analysis 696
and classes 732
and language 724
and language, gap between 725
and programming 692
bibliography 719
class hierarchy 314
collaboration 708
container 434, 441
error 711
for testing 712
how to start a 708
hybrid 718
inheritance and 707
integrity of 716
language and programming language 730
method 694
method, choosing a 696
object-oriented 692, 726
of C++ 7, 10
of [] 295
of access operator 295
reuse 709
stability of 708
stage 697
standard library 429–430
steps 701
string 579
template in 757
tool, blackboard as 703
tool, presentation as 704
tool, tutorial as 708
tools 711
unit of 755
designers, coders and 695
destroy() 567
destruction
construction and 244
order or construction and 414
destructor 33, 283
and garbage collection 846
and union 257
anomaly, constructor and 245
common code and 624
constructor and 242, 246–247
exception in 373
explicit call of 256
for derived class 306
virtual 319
destructors
atexit() and 218
exceptions and 366
development
cycle 698
process 696
software 692
stage 697
diagnostics 432
diamond-shaped inheritance 399
dictionary 480
– see map
difference
from C 816
from C // 816
from C argument type 817
from C array initializer 818
from C declaration and definition 818
from C enum 817
from C function call 816
from C function definition 817
from C initialization and goto 818
from C int implicit 817
from C jump past initialization 818
from C macros 818
from C scope 816
from C sizeof 816
from C struct name 818
from C struct scope 818
from C void* assignment 818
difference_type 443, 480, 552
basic_string 583
digits 658
digits10 659
digraph
%: 829
%:%: 829
%> 829
:> 829
<% 829
<: 829
direct manipulation 730
directed acyclic graph 308
direction
of seek, seekdir
of seekg()
of seekp()
directive

#, preprocessing 813
template instantiation 866
discrimination of exception 188
disguised pointer 844
dispatch, double 326
distance() and - 551, 554
distribution
exponential 685
uniform 685
div() 661
divide and conquer, complexity 693
divides / 517
div_t 661
do statement 114, 137
documentation 714–715
do_it() example 777
domain error 661
dominance 401
Donald Knuth 713
dot product 684
double
dispatch 326
quote 830
double 74
output 626
double-ended queue deque 474
doubly-linked list 470
down cast 408
draw_all() example 520
Duff's device 141
dynamic
memory 127, 576, 843
store 34
type checking 727
type checking, mis-use of 439
dynamic_cast 407–408
and ambiguity 412
and polymorphism 409
and static_cast 413
bad_cast and 384
implementation of 409
to reference 410
use of 774

E

eatwhite() 620
eback() 645
EDOM 661
efficiency 8, 713
and coupling 768
and generality 431
of operation 464
egptr() 645
element
access 445
access, list 472
access, map 482
address of 454
constructor for array 250
first 445
from sequence, delete 529, 534
last 445
object, array 244
requirements for 466
to sequence, add 529
eliminate_duplicates() example 534
eliminating programmers 730
elimination
of copying 675
of temporary 675
ellipse, circle and 703
ellipsis ... 154
else 134
emphasis, examples and 5
Employee example 302
empty string 585
empty() 455, 489
string 598
encapsulation 754
complete 283
end, seekdir and
end() 54, 481
basicstring 584
iterator 444
#endif 162
endl 634
ends 634
engineering, viewgraph 704
enum 76
and integer 77
bitset and 492
conversion, undefined 77
difference from C 817
member 249
sizeof 78
user-defined operator and 265
enumeration 76
switch on 77
enumerator 76
EOF 620, 653
eof() 616
char_traits 581
eofbit 617
epptr() 645
epsilon() 659
eq(), char_traits 581
eq_int_type(), char_traits 581
equal() 527
equality
and comparison 457
hash_map 497

without == 468
equal_range() 540
in map 485
equal_to == 516
equivalence, type 104
Erand 685
ERANGE 601, 661
erase()
from map 487
from vector 452
in string 595
errno 383, 601, 661
<errno.h> 432
error
analysis 711
design 711
domain 661
exception and 622
handling 115, 186, 383, 566
handling, C-style 661
handling alternative 192, 355
handling, multilevel 383
linkage 199
loop and 523
range 661
recovery 566
reporting 186
run-time 29, 355
sequence 512
string 586
errors, exceptions and 355, 374
escape character \ 73, 830
essential operators 283
evaluation
lazy 707
order of 122
short-circuit 123, 134
event
asynchronous 357
driven simulation 326
event 651
event_callback 651
example
(bad), Shape 417
Assoc 286
Buffer 738
Car 772
Checked 565
Checked_iter 561
Clock 398
Cowboy 778
Date 236
Employee 302
Expr 424
Extract_officers 524
Filter 786
Form 635
Hello, world! 46
Io 776
Io_circle 775
Io_obj 774
Ival_box 315, 407
Lock_ptr 366
Math_container 346
Matrix 282
Object 417
Plane 729
Pool 570
Range 781
Rational 747
Saab 728
Set 769
Set_controller 785
Shape 774
Slice_iter 670
Stack 27
Storable 396
String 328
Substring 596
Table 243
Vector 341, 780
Vehicle 734
Window 398
cache 232
calculator 107, 165, 190, 208
callC() 384
call_from_C() 384
cloud 700
delete_ptr() 531
do_it() 777
draw_all() 520
eliminate_duplicates() 534
identity() 531
iocopy() 617
iosbase::Init 639
iseq() 513
of default argument value 227
of input 114
of operator overloading 292
of reference 292
of user-defined memory management 292
of virtual function 646
oseq() 556
scrollbar 743
sort() 158, 334
example:, member template 349
examples and emphasis 5
exception 29, 186, 355
C and 383
C-function and 382
I/O 622
and error 622

and function 375
and interface 375
and main() 54
and member initialization 373
and multiple inheritance 360
and new 367, 369
and recursive function 374
atexit() and 382
bad_alloc 576
catch every 54
container algorithm 566
copy of 362
cost of 381
discrimination of 188
handler 812
in constructor 367
in destructor 373
mapping 378
new and 576
qsort() and 382
standard 384
type of 379
<exception> 379–380, 384–385, 433
exception hierarchy 385
exceptions 357
and constructors 366
and destructors 366
and errors 355, 374
derived 359
grouping of 358
in constructor 371
uncaught 380
unexpected 377
exceptions() 622
exception-specification 375
checking of 376
exclusive or operator ^, bitwise 124
exhaustion
free store 129
resource 369
exit() 116, 218
exp(), valarray 667
experimentation and analysis 710
explicit
call of destructor 256
conversion 284
qualification :: 847
template argument 335
template instantiation 866
type conversion 130
explicit constructor 284
exponent, size of 659
exponential distribution 685
exponentiation, vector 667
export 205
Expr example 424
expr() 660
complex 680
expression
conditional 134
constant 833
expression, full 254
expression 798
extended type information 416
extensibility 700
extensible I/O 605
extern 205
extern 198
external linkage 199
Extract_officers example 524
extractor, >> 608

F

\f, formfeed 830
fabs() 660
facilities, standard library 66, 429
factory 323
fail() 616
failbit 617
failure 709, 716
failure and clear() 385
false and 0 71
fat interface 439, 761
fault tolerance 383
feature
summary, C++ 819
vs technique, built-in 43
features, portability and 815
feedback 695, 698
field
bit 125, 840
output 629–630
type of 75
fields, order of 75
file
.c 202
.h 201
and stream 637
closing of 638
header 27, 201
input from 637
mode of 639
opening of 638
output to 637
position in 642
source 197
filebuf 649
fill() 537, 629
fill_n() 537
Filter example 786
finally 362

find the classes 702
find() 57, 525
 char_traits 581
 in map 485
 in string 594
find_end() 528
find_first_not_of() in string 594
find_first_of() 525
 in string 594
find_if() 62, 525
finding the classes 734
find_last() 444
find_last_of() in string 594
firewall 383
first
 C++ library 686
 element 445
first-time switch 253, 640
fixed 626, 628
fixed() 634
flag manipulation 626
flags() 626
flexibility 700
flip() bitset 494
float 74
 bits in 658
 output 626
float_denorm_style 659
floatfield 626, 628
<float.h> 433
floating
 point output 626, 628
 point type 74
floating-point
 . 74
 conversion 834
 conversion to 835
 literal 74
 promotion 833
float_round_style 659
floor() 660
FLT_RADIX 660
flush 634
flush() 631, 642
flushing of output 626
fmod() 660
For 511
for
 statement 26, 136
 statement, declaration in 137
for(;;) 109
for_each() 62, 523
Form example 635
formal
 methods 711
 model 730
format
 control 625
 information, locale 606
 object 635
 of date 649
 of integer 649
 state 625
 state, basic_ios 606
 state, ios_base 606
 string 652
formatted output 625
formatting
 basic_iostream 606
 in core 641
formfeed \f 830
for-statement initializer 825
Fortran
 style subscript 674
 vector 668
forward
 and output iterator 554
 declaration of member class 293
 iterator 550
 reference to class 278
forwarding function 778, 780
forward_iterator_tag 553
foundation operator 706
fragmentation, memory 846
framework, application 731, 786
free
 store 34, 127, 421, 576, 843
 store exhaustion 129
 store object 244
 store object, constructor for 246
free() 577
free-standing
 class 732
 function 732
frexp() 660
friend 16, 278, 852
 and member 265, 280
 class 279
 declaration 279
 derived and 852
 function 279
 of friend 852
 template and 854
front operation 472
front() 445, 472
 of queue 476
front_inserter() 555
front_insert_iterator 555
<fstream> 432, 638
fstream 638
function
 adapter, pointer to 521

and algorithm, C-style 522
argument passing 145
argument type check 145
argument type conversion 145
argument types, `virtual` 310
call 145
call and assignment 99
call, difference from C 816
class 776
`const` member 229
declaration 143
definition 144
definition, difference from C 817
definition of `virtual` 310
definition, old-style 817
example of `virtual` 646
exception and 375
forwarding 778, 780
free-standing 732
`friend` 279
`get()` 759
helper 273
higher-order 518
implementation of `virtual` 36
`inline` 144
`inline` member 235
inspector `const` 706
member 224, 238
name, overloaded 149
nested 806
object 287, 514, 776
object, arithmetic 517
only, instantiate used 866
operator `::` and `virtual` 312
pointer to 156
pointer to member 418
pure `virtual` 313
`set()` 759
specialization 344
`static` member 278
`template` 334
`template` argument 335
`template` overloading 336
type of overriding 424
value `return` 148
`virtual` 310, 390, 706
virtual 15
`virtual` output 612
functional
decomposition 725
decomposition and C++ 726
`<functional>` 431, 516–519, 521
function-like `class` 514
functions, list of `operator` 262
functor 514
fundamental
sequence 469
type 23, 70

G

game 685
gap between design and language 725
garbage
collection, automatic 247, 844
collection, `delete` and 845
collection, destructor and 846
collector 128, 130
gargantuanism 713
`gbump()` 645
`gcount()` 618
general allocator 573
generality
efficiency and 431
of sequence 512
of solution 701
generalized
numeric algorithm 682
slice 677
general-purpose programming-language 21
`generate()` 537
generated
`!=` 468
`<=` 468
`>` 468
`>=` 468
specialization 859
`generate_n()` 537
generator
random number 537
type 348
generic
algorithm 41
programming 40, 757–758
programming, `template` and 327
get
area 645
from, `>>` 607
position, `tellp()` 642
`get()` 618, 643
`char` 620
function 759
`get_allocator()` 457
from `string` 598
`getchar()` 653
`getline()` 51, 618
into `string` 598
`getloc()` 646, 650
`get_state()`, `char_traits` 581
`get_temporary_buffer()` 575
global 16
initialization of 217

namespace 847
object 244, 252
objects 640
scope 82, 847
variable 200, 228
variable, constructor for 252
variable, use of 111
global() locale 649
goal, clear 698
good() 616
goodbit 617
goto
difference from C initialization and 818
nonlocal 357
statement 137
gptr() 645
gradual
adoption of C++ 718
approach to learning C++ 7
grammar 793
graph, directed acyclic 308
greater > 516
greater_equal >= 516
grouping of exceptions 358
growing system 711
gslice 677
gslice_array 677
guarantees, standard 827

H

.h
file 201
header 821
hack, struct 809
half-open sequence 512
handle
class 782
intrusive 783
handler, exception 812
hardware 75
has-a 741
has_denorm 659
has_denorm_loss 659
hash
function 502
function, hash_map 497
table 497
hashing 502
hash_map 497
delete from 501
equality 497
hash function 497
lookup 500
representation 498
resize() 502
has_infinity 659
has_quiet_NaN 659
has_signaling_NaN 659
header 117, 201
.h 821
and compile time 211
file 27, 201
standard 431
standard library 202
heap 34, 543, 576
and priority_queue 543
memory 843
store 127
heap, priority_queue and 479
Hello, world! example 46
helper
class 293
function 273
function and namespace 240
hex 626–627, 634
hexadecimal 73
output 626
hiding
information 27
name 82
hierarchies, interface 708
hierarchy 732
class 38, 389
class 15, 307, 734
design, class 314
exception 385
navigation, class 411
object 739, 748
reorganization of class 707
stream 637
traditional 315
higher-order function 518
high-level language 7
Histogram 455
horizontal tab \t 830
how to start a design 708
human activity, programming as a 693
hybrid design 718

I

ideas, real-world as source of 734
identifier 81
identity() example 531
IEC-559, is_iec559 659
if
statement 133
switch and 134
_if suffix 525
#ifdef 162
#ifndef 216

ifstream 638
ignore() 618
imag() 679–680
imbue() 645, 647, 650
 imbue_event 651
imbue_event, imbue() 651
implementation
 alternative 320
 and interface 317
 dependency type of integer literal 832
 inheritance 400, 743
 interface and 224, 314, 399, 758, 771
 iterator 59
 of I/O 606
 of RTTI 409
 of container 465
 of dynamic_cast 409
 of virtual function 36
 pre-standard 820
 priority_queue 478
 stack 475–476
 stage 697
implementation-defined 827
implicit
 conversion 275, 281, 284
 conversion of string 590
 type conversion 76, 276, 833
implicit_cast 335
in core formatting 641
In 511
in open for reading 639
in_avail() 644, 646
in-class
 definition 235
 definition of constant 249
#include guard 216
include directory, standard 201
#include 27, 117, 183, 201
includes() 542
inclusion, template 350
increment
 and decrement 291
 operator ++ 125
incremental change 684
indentation 138
independent concept 327
index 454
indirect_array 679
indirection 290
individual 716
inertia, organizational 713
infinity() 659
information hiding 27
inheritance 39, 303, 307, 703
 and design 707
 and template 349
 containment and 740
 data abstraction vs 727
 dependency 737
 diamond-shaped 399
 implementation 400, 743
 interface 400, 743
 multiple 308, 390, 735
 template and 347
 using multiple 399
 using-declaration and 392
 using-directive and 392
initialization 79, 226, 244
 and cleanup 364
 and goto, difference from C 818
 assignment and 283
 cerr 637
 cin 637
 clog 637
 constructor and C-style 270
 cout 637
 difference from C jump past 818
 library 640
 main() and 217
 member 248
 of array by string 89
 of base class 306
 of global 217
 of reference 98
 of structure 102
 order of member 247
 reference member 244, 250
 run-time 217
initializer
 array 89
 for-statement 825
 list, constructor and 270
 member 247
initiative 695
inline
 and linkage 199
 function 144
 member function 235
inner product 684
inner_product() 683
innovation 717
inplace_merge() 541
input
 and output 432, 605
 and output, C 651
 and output, connection between 623
 bitset 495
 char 618
 cin >> 50, 112
 complex 680
 example of 114
 from file 637

into container 451
into vector 451
iterator 550
manipulator 632
of bool 615
of built-in type 614
of char 615
of pointer 615
of user-defined type 621
sequence 513
string 598
unbuffered 642
unformatted 618
valarray 668
width() of 616
input_iterator_tag 553
insert() 55
[] and 488
into map 487
into vector 452
string 592
inserter, << 608
inserter() 555
insertion, overwriting vs 555
insert_iterator 555
inspector const function 706
inspiration 734
instantiate used function only 866
instantiation
context of template 860
directive, template 866
explicit template 866
multiple 867
point of 863
template 859
int 73, 76
bits in 658
implicit, difference from C 817
largest 658
output bits of 495
smallest 658
integer
conversion 834
conversion, signed unsigned 834
enum and 77
format of 649
literal 73, 76
literal, implementation dependency type of 832
literal, type of 832
output 627
type 70, 73
type, conversion to 835
value 0, null 830
integral
conversion to 835
promotion 833
type 70
integration 728
integrity of design 716
interface
alternative 173
and implementation 224, 314, 399, 758, 771
changing 774
class 778
exception and 375
fat 439, 761
hierarchies 708
implementation and 317
inheritance 400, 743
module and 165
multiple 172
public and protected 645
specifying 707
internal
linkage 199
structure 694
internal 625, 630
internal() 634
INT_MAX 660
introducing C++ 718
intrusive
container 438
handle 783
int_type 608
int_type(), char_traits 581
invalid_argument and bitset() 385
invariant 748
checking 749
I/O 47, 50
buffering 645
design 605
exception 622
extensible 605
implementation of 606
iterator and 60
object 774
sentry 624
system, organization of 606
type safe 607
unbuffered 647
wide character 608
Io example 776
Io_circle example 775
iocopy() example 617
<iomanip> 432, 633
Io_obj example 774
<ios> 432, 608
ios 625, 822
ios_base 626, 628–629, 650
format state 606
iosbase::Init example 639
<iosfwd> 432, 607

iostate 617, 822
io_state 822
<iostream> 46, 432, 609, 614
 <istream> and 613
 <ostream> and 608
iostream 637
 sentry 624
is-a 741
isalnum() 601
isalpha() 113, 601
is_bounded 659
iscntrl() 601
isdigit() 601
Iseq 513
iseq() example 513
is_exact 659
isgraph() 601
is_iec559 IEC-559 659
is_integer 658
islower() 601
is_modulo 659
ISO
 646 829
 C++ 11
isprint() 601
is_signed 658
isspace() 601, 615
 whitespace 114
is_specialized 658
<istream> 432
 and <iostream> 613
istream 614, 643
 >> 614
 and iterator 559
istreambuf iterator 559
istreambuf_iterator 559
istream_iterator 60, 559
istringstream 641
istrstream 656
isupper() 601
isxdigit() 601
iterator 57, 549
 != 551
 * 551
 + 551
 ++ 551
 += 551
 - 551
 -- 551
 -= 551
 -> 551
 < 551
 <= 551
 == 551
 > 551
 >= 551
 STL 441
 Sink 532
 [] 551
 abstract 435
 and I/O 60
 and sequence 550
 begin() 444
 bidirectional 550
 category 553
 checked 561
 const 443
 cont 508
 container 464
 container and 435, 444
 end() 444
 forward 550
 forward and output 554
 implementation 59
 input 550
 istream and 559
 istreambuf 559
 map 481
 naming convention 511
 operation 551
 ostream and 558
 ostreambuf 560
 output 550
 random-access 550
 rbegin() 444
 read through 551
 rend() 444
 reverse 443, 557
 stream 558
 string 584
 user-defined 561
 valarray 670
 valid 550
 write through 551
<iterator> 432
iterator 54, 443, 480
 basic_string 583
iterator_category 552
iterator_traits 552
iter_swap() 538
itoa() 155
Itor 435
Ival_box example 315, 407
Ival_slider 399
iword() 650

J

jump past initialization, difference from C 818

K

keeping consistent declarations 201
Kernighan and Ritchie 654
key 55
 and value 480
key,
 duplicate 480, 490
 unique 480
`key_comp()` 485
`key_compare` 480, 485
`key_type` 480
keyword 793–794
 `and` 829
 `and_eq` 829
 `bitand` 829
 `bitor` 829
 `compl` 829
 `not` 829
 `not_eq` 829
 `or` 829
 `or_eq` 829
 `xor` 829
 `xor_eq` 829
kind
 of `class` 765
 of container 461
kinds of object 244
Knuth, Donald 713

L

`L'`, wide-character literal 73
`labs()` 661
lack of modularity 309
language
 and library 45
 design and 724
 gap between design and 725
 high-level 7
 low-level 8
 people and machines 9
 programming 15
 programming styles technique 6
 support 433–434
large
 character set 831
 program 211–212
 programs and C++ 9
largest `int` 658
last element 445
last-time switch 640
Latin-1 580
lattice, `class` 389
layout, array 669
lazy evaluation 707
`ldexp()` 660
`ldiv()` 661
`ldiv_t` 661
leaf `class` 774
learning
 C and 7
 C++ 6, 718, 820
 C++, gradual approach to 7
`left` 625, 630
`left()` 634
legacy 708
length of `valarray` 664, 679
`length()`
 `char_traits` 581
 of `string` 598
 `string` 586
`less` 515
 `<` 516
`less_equal` `<=` 516
`less_than` 519
levels of abstraction 733
lexical conventions 794
`lexicographical_compare()` of sequence 544
libraries, standard 700
library 15, 701, 714, 755
 C standard 599
 algorithms, standard 64
 container, standard 56
 facilities, standard 66, 429
 first C++ 686
 header, standard 202
 initialization 640
 language and 45
 non-standard 45
 standard 45, 182
 standard – see standard library
lifetime
 of object 84
 of temporary 254
limits, numeric 658
`<limits>` 433, 658
`<limits.h>` 433, 660
line, read 618
linear time 464
`Link` 394
linkage
 and `namespace` 207
 and pointer to function 207
 `const` and 199
 error 199
 external 199
 `inline` and 199
 internal 199
 to C 205
 type-safe 198
linker 198

Liskov substitution 743
Lisp 725
list
 of operator functions 262
 operation 452
<list> 431
List 435
list 54
 doubly-linked 470
 element access 472
 merge() algorithm and 541
 merge() stable 470
 remove() 472
 remove_if() 472
 reverse() 472
 sort() stable 470
 unique() 472
literal
 ', character 73
 L', wide-character 73
 floating-point 74
 implementation dependency type of integer 832
 integer 73, 76
 of user-defined type 273
 string 294
 string 46, 90
 type of integer 832
loader 198
local
 fix 697
 scope 82
 static 145
 static store 251
 variable, constructor for 245
<locale> 433, 649
locale 649
 POSIX 649
 classic() 649
 format information 606
 global() 649
locale() 649
<locale.h> 433, 650
locality 212
localization of object creation 322
locking 366, 785
Lock_ptr example 366
log() 660
 complex 680
 valarray 667
log10() 660
 complex 680
 valarray 667
logarithmic time 464
logical
 and operator && 123
 const, physical and 231
 operators, bitwise 124
 or operator || 123
 structure of program 198
logical_and && 516
logical_not 515
 ! 516
logical_or || 516
long namespace name 178
long 73
long double 74
longer term 699
lookup, hash_map 500
loop
 and error 523
 merging 675
 statement 116
lower_bound() 540
 in map 485
low-level language 8
lt(), char_traits 581
lvalue 84, 264, 281
lying 705

M

machines, language people and 9
macro 160
 alternative to 161
macros, difference from C 818
main() 46, 116, 218
 and initialization 217
 argv argc 117
 exception and 54
maintenance 212
 software 712
make_heap() 543
make_pair() 482
malloc() 577
management 713
 memory 843
manipulator
 input 632
 output 631
 standard 633
 user-defined 635
 with argument 633
mantissa, size of 659
manual overload resolution 151
map 480
<map> 431
map 55, 480
 = 484
 [] 482
 assignment 484
 comparison in 484
 constructor 484

count() in 485
element access 482
equal_range() in 485
erase() from 487
find() in 485
insert() into 487
iterator 481
lower_bound() in 485
member type 480
modify a 487
subscripting 482
upper_bound() in 485
use of 774
mapped type, value 55
mapped_type 480
mapping exception 378
Marian 79
masks_array 678
Math_container example 346
mathematical
functions, complex 680
functions, standard 660
functions, valarray 667
functions, vector 667
model 711
<math.h> 434, 660
Matrix 672
example 282
max() 544, 658
valarray 664
max_element() of sequence 544
max_exponent 659
max_exponent10 659
max_size() 455, 489
of string 598
meaning for operator, predefined 264
means, aims and 694
measurement, productivity 716
member
->*, pointer to 418
.*, pointer to 418
::*, pointer to 418
access operator -> 102
access operator . 101
access to 849
and nonmember operators 267
class 293
class, access to 851
class, forward declaration of 293
const 249
constant 249
constructor for class 247
enum 249
friend and 265, 280
function 224, 238
function adapter 520
function, algorithm and 520
function, const 229
function, inline 235
function, pointer to 418
function, static 278
initialization 248
initialization, exception and 373
initialization, order of 247
initialization, reference 244, 250
initializer 247
object 244
object, union 244
of base class, private 305
of derived class 304
of template, static 854
or base 740
or pointer 738
pointer to data 853
private class 225
protected 404–405
public class 225
reference 740
static 228, 421
template 330
template example: 349
template, missing 823
type, basic_string 582
type, map 480
type, vector 442
union 257, 843
member-declaration 808
memberwise copy 283
memchr() 577
memcmp() 577
memcpy() 577
mem_fun() 63, 518, 521
mem_fun1_ref_t 518, 521
mem_fun1_t 518, 521
mem_fun_ref() 518, 521
mem_fun_ref_t 518, 521
mem_fun_t 518, 520–521
memmove() 577
memory
automatic 843
buffer 575
dynamic 127, 576, 843
fragmentation 846
heap 843
management 843
management, automatic 844
management, container 455, 567
management, example of user-defined 292
stack 843
static 843
uninitialized 574
<memory> 431, 574

memset() 577
merge() 541
  algorithm and list 541
  stable, list 470
message queue 477
method 310
  choosing a design 696
  choosing an analysis 696
  cookbook 692
  design 694
methods, formal 711
min() 544, 658
  valarray 664
min_element() of sequence 544
min_exponent 659
min_exponent10 659
minimalism 706
minimize dependency 173
minus - 517
mismatch() 516, 527
missing
  bad_alloc 823
  checking, late 823
  member template 823
  namespace 822
  specialization partial 823
  standard library 822
mis-use
  of RTTI 439
  of dynamic type checking 439
misuse
  of C++ 725
  of RTTI 417
mixed-mode arithmetic 268
mixin 402
mixing C and C++ 719
ML 10
mode of file 639
model
  formal 730
  mathematical 711
  waterfall 697
models 708
modf() 660
modifier 706
modify a map 487
modifying sequence algorithm 529
modular programming 26
modularity 312
  lack of 309
module
  and interface 165
  and type 30
modulus % 517
moron 713, 717
move(), char_traits 581
multidimensional
  array 668, 677, 836
  array, passing 839
multilevel error handling 383
multimap 490
multi-method 326
multiple
  inheritance 308, 390, 735
  inheritance, exception and 360
  inheritance, use of 776
  inheritance, using 399
  instantiation 867
  interface 172
multiple-inheritance
  access control and 406
  ambiguity resolution 391
multiplies * 517
multiset 491
mutable 232
mutual reference 278

N

\n, newline 830
name 81
  binding 860
  binding, template 859
  characters in 81
  clash 176
  hiding 82
  long namespace 178
  namespace qualified 169
  short namespace 178
names, reserved 81
namespace
  nested 848
  transition to 182
namespace 27, 167, 847
  alias 178
  and :: 169
  and class 849
  and overloading 183
  composition 179
  composition of 181
  global 847
  helper function and 240
  is open 184
  linkage and 207
  member declaration and definition 167
  missing 822
  name, long 178
  name, short 178
  operators and 265
  purpose of 180
  qualified name 169
  relops 468

selection from 180
std 46
unnamed 177, 200
using 183
naming convention, iterator 511
narrow() 645
n-ary operators 675
national
character 829
conventions 649
natural operation 767
NDEBUG 750
negate - 517
nested
class 293
function 806
namespace 848
nesting 756
<new> 384, 433, 576
new
and array 423
and exception 576
bad_alloc and 384
exception and 367, 369
operator 127
placement 255
size and 421
new()
operator 129, 576
placement 576
new[]() , operator 423, 576
new_handler 129, 576
_new_handler 370
newline \n 830
next_permutation() 545
Nicholas 49
noboolalpha() 633
Nocase 467
node
class 772
class, abstract 774
class, concrete 774
non-C++ program 217
none() 494
nonlocal goto 357
nonmember operators, member and 267
nonmodifying sequence algorithm 523
non-standard library 45
non-type template parameter 331
norm() 680
noshowbase() 634
noshowpoint() 634
noshowpos() 634
noskipws() 634
not a class 705
not keyword 829
not1() 518
and unary_negate 522
not2() 518
and binary_negate 522
notation, value of 261
not_eof(), char_traits 581
not_eq keyword 829
not_equal_to != 516
nothrow 576
allocator 823
nouppercase() 634
npos 586
nth_element() 540
null
0 zero 88
integer value 0 830
NULL 88, 433
number, size of 75
numeric
algorithm, generalized 682
arrays 662
limits 658
<numeric> 434, 682
numerical computation 64
numeric_limits 658

O

O notation 464
object 32, 84
I/O 774
array element 244
automatic 244
constructor for free store 246
creation 242
creation, localization of 322
format 635
free store 244
function 287, 514, 776
global 244, 252
hierarchy 739, 748
kinds of 244
lifetime of 84
member 244
placement of 255
real-world 732
state of 748
static 244
temporary 244, 254
union member 244
variably-sized 243
Object 438
example 417
object-oriented
design 692, 726
programming 37–38, 301

pure 732
objects
array of 250
global 640
`oct` 626–627
`oct()` 634
octal 73
output 626
ODR the one-definition-rule 203
offset, pointer to member and 419
`off_type` 608, 643
`char_traits` 581
`ofstream` 638
old-style function definition 817
one right way 693
one-beyond-last 512
one-definition-rule, ODR the 203
`Op` 511
open
for reading, `in` 639
for writing, `out` 639
`namespace` is 184
opening of file 638
`openmode` 639
operation
`bitset` 494
efficiency of 464
front 472
iterator 551
list 452
natural 767
on container 464
operations
`complex` 679
on references 97
on structure 102
selecting 705
set of class 237
`valarray` 664, 667
vector 664, 667
operator
`,` 123
&, bitwise `and` 124
&&, logical `and` 123
+, user-defined 281
++, increment 125
++, user-defined 264, 291
`+=` 109
+=, user-defined 264, 268, 281
`--`, decrement 125
`--`, user-defined 291
`-=` 109
`->`, member access 102
`->`, user-defined 289
`.`, member access 101
`::` 305
`::` and `virtual` function 312
`::`, scope resolution 82, 228
<<, output 607
=, user-defined 281
^, bitwise exclusive or 124
and built-in type, user-defined 265
and `enum`, user-defined 265
assignment 110, 268
associativity of 121
binding strength 121, 607
composite 268
declarator 80
`delete` 127
design of access 295
foundation 706
`new` 127
overloaded 241
overloading, example of 292
precedence 121
predefined meaning for 264
stack 450
summary 119
ternary 636
type conversion 275
user-defined 263
user-defined binary 263
user-defined unary 263
`|`, bitwise `or` 124
`||`, logical `or` 123
`operator`
`delete()` 129, 576
`delete[]()` 423, 576
functions, list of 262
`new()` 129, 576
`new[]()` 423, 576
`void*()` 616
`operator()` 287
`operator[]` 286
operator 810
operators
and `namespace` 265
bitwise logical 124
essential 283
member and nonmember 267
n-ary 675
optimal container 435
optimization 675
`or`
keyword 829
operator `|`, bitwise 124
operator `||`, logical 123
order 467
of construction 248, 252
of evaluation 122
of fields 75
of member initialization 247

of specialization 343
or construction and destruction 414
or_eq keyword 829
organization
of I/O system 606
standard library 431
organizational inertia 713
orthogonality, convenience and 431
oseq() example 556
<ostream> 432
and <iostream> 608
ostream 608, 642
<< 609
and buffer 642
and iterator 558
and streambuf 642
put() 609
template and 608
write() 609
ostreambuf iterator 560
ostreambuf_iterator 560
ostream_iterator 60, 558
ostringstream 641
ostrstream 656
Out 511
out open for writing 639
out_of_range 53, 446
and at() 385
string 586
output 47
C input and 651
bits of int 495
bitset 495
complex 680
connection between input and 623
cout << 46
decimal 626
double 626
field 629–630
float 626
floating point 626, 628
flushing of 626
formatted 625
function, virtual 612
hexadecimal 626
input and 432, 605
integer 627
iterator 550
manipulator 631
octal 626
of bool 610
of built-in type 609
of char 610
of pointer 611
of user-defined type 612
operator << 607
padding 625
sequence 556
string 598
to file 637
unbuffered 642
valarray 668
why, << for 607
output_iterator_tag 553
overflow, stack 476
overflow() 647
overflow_error and to_ulong() 385
overhead 8
overlapping sequences 529
overload
resolution 149
resolution, manual 151
return type and 151
scope and 151
overloaded
function name 149
operator 241
overloading
const and 600
example of operator 292
function template 336
namespace and 183
override 313
private base 738
overriding 395
from virtual base class 401
function, type of 424
overwriting vs insertion 555

P

padding 630
output 625
pair 482
paradigm, programming 22
parameter
non-type template 331
template 331
parameterization
policy 757
template 707
parametric polymorphism 347
parentheses, uses of 123
parser, recursive decent 108
partial
sort 539
specialization 342
partial_sort() 539
partial_sort_copy() 539
partial_sum() 684
partition 542
partition() 542

partitioning of program 208, 211
passing multidimensional array 839
pattern 709
  specialization 342
pbackfail() 647
pbase() 645
pbump() 645
peek() 643
people and machines, language 9
perfection 43
permutation 545
per-object data 573
per-type data 573
phone_book example 52
physical
  and logical const 231
  structure of program 198
placement
  new 255
  new() 576
  of object 255
Plane example 729
plug analogy 728
plus + 517
point
  of declaration 82
  of definition 861
  of instantiation 863
pointer 26, 87
  and array 91, 147
  arithmetic 88, 93, 125
  checked 291
  checking for, wild 722
  const 96
  conversion 834
  conversion, user-defined 349
  disguised 844
  input of 615
  member or 738
  output of 611
  semantics 294
  size of 75
  smart 289, 291
  to class 304
  to class, conversion of 304
  to const 96
  to constructor 424
  to data member 853
  to function 156
  to function, << of 631
  to function, >> of 632
  to function adapter 521
  to function, linkage and 207
  to member ->* 418
  to member .* 418
  to member ::* 418
  to member and offset 419
  to member function 418
  to void 100
  type 569
pointer 443, 552, 567
  basic_string 583
pointers and union 845
pointer_to_binary_function 521
pointer_to_unary_function 518, 521
polar() 680
policy parameterization 757
polymorphic 35
  object, algorithm and 63
polymorphism 158, 312
  algorithm and 520
  compile-time 347
  container and 520
  dynamic_cast and 409
  parametric 347
  run-time 347
  see virtual function
Pool example 570
Pool_alloc allocator 572
pop()
  of priority_queue 478
  of queue 476
  of stack 475
pop_back() 450
pop_front() 472
pop_heap() 543
portability 9, 700, 828
  and features 815
position
  bit 492
  in buffer 642
  in file 642
POSIX locale 649
postcondition 753
pos_type 608, 643
  char_traits 581
pow() 660
  complex 680
  valarray 667
pptr() 645
precedence
  << 608
  operator 121
precision() 628
precondition 753
Pred 511
predefined
  , 264
  & 264
  = 264
  meaning for operator 264
predicate 61, 63, 515

standard library 516
user-defined 516
prefix code 624
preprocessing directive # 813
presentation as design tool 704
pre-standard implementation 820
`prev_permutation()` 545
`printf()` 651
priority queue 478
`priority_queue`
and `heap` 479
heap and 543
implementation 478
`pop()` of 478
`push()` of 478
`top()` of 478
private
class member 225
member of base class 305
`private` 402
base 405, 742
base class 743
base, override 738
`public protected` 849–850
`private:` 234
problems
of scale 715
with concrete type 37
procedural
programming 23
programming and C++ 725
process, development 696
product
dot 684
inner 684
productivity measurement 716
program 46, 798
large 211–212
logical structure of 198
non-C++ 217
partitioning of 208, 211
physical structure of 198
size of 8
start 217
structure of 8
termination 218
programmed-in relationship 746
programmer
C 14
C++ 14
programmers, eliminating 730
programming 16
and C++, procedural 725
as a human activity 693
design and 692
generic 40, 757–758
language 15
language, design language and 730
modular 26
object-oriented 37–38, 301
paradigm 22
procedural 23
purpose of 694
style 22
styles technique language 6
`template` and generic 327
programming-language, general-purpose 21
prohibiting
`,` 264
`&` 264
`=` 264
promotion
floating-point 833
integral 833
standard 833
proof by analogy 692
properties of C++ 724
`protected` 402
base 319, 405
base class 743
interface, `public` and 645
member 404–405
`private`, `public` 849–850
protection 226
unit of 754
prototypes 710
proxy 785
`Ptr` 349
`ptrdiff_t` 122, 433
`ptrfun()` 518
`ptr_fun()` 521
`pubimbue()` 646
public class member 225
`public` 402
and `protected` interface 645
`protected private` 849–850
`public:` 225, 234
`pubseekoff()`
`pubseekpos()`
`pubsetbuf()` 646
`pubsync()` 646
pure
object-oriented 732
`virtual` function 313
purpose
of `namespace` 180
of programming 694
`push()`
of `priority_queue` 478
of `queue` 476
of `stack` 475
`push_back()` 55, 450

and realloc() 451
push_front() 55, 472
push_heap() 543
put
area 645
to, << 607
put(), ostream 609
putback() 643
pword() 650

Q

qsort() 158, 546
and exception 382
quadratic time 464
qualification ::, explicit 847
qualified name, namespace 169
qualifier, template as 858
quality 717
queue
deque, double-ended 474
priority 478
<queue> 431
queue
back() of 476
front() of 476
message 477
pop() of 476
push() of 476
quiet_NaN() 659
quote
\', single 830
double 830
quotient 661

R

\r, carriage return 830
Ran 511
rand(), random number 685
Randint 685
RAND_MAX 685
random
number 538
number class 685
number generator 537
number rand() 685
random-access iterator 550
random_access_iterator_tag 553
random_shuffle() 537
range
check 445, 561
check of string 584
check, valarray 664
checking 275, 781
checking Vec 53
error 661
sequence and 512
Range example 781
Rational example 747
raw storage 574
raw_storage_iterator 575
rbegin() 481
basicstring 584
iterator 444
rdbuf() 644
rdstate() 616
read
line 618
through iterator 551
read() 618
readsome() 643
real() 679–680
realloc() 577
push_back() and 451
real-world
as source of ideas 734
classes and 734
object 732
rebind 567
use of 569
recursion 148
recursive
decent parser 108
function, exception and 374
reduce 683
reduction 683
redundancy 712
reference 97
argument 98
call by 98, 146
call-by 282
count 292
counting 783
dynamic_cast to 410
example of 292
initialization of 98
member 740
member initialization 244, 250
mutual 278
return by 148
return by 283
to class, forward 278
reference 443, 480, 552, 567
basic_string 583
to bit 492
references, operations on 97
register 806
register_callback() 651
reinterpret_cast 130, 256
relationship, programmed-in 746

relationships between templates 348
relaxation of return type 424
release, resource 364
reliability 383
`relops`, `namespace` 468
remainder 661
`remove()` 536
 `list` 472
`remove_copy_if()` 536
`remove_if()` 536
 `list` 472
renaming virtual function 778
`rend()` 481
 `basicstring` 584
 iterator 444
reorganization of class hierarchy 707
`replace()` 535
 in `string` 595
`replace_copy()` 535
`replace_copy_if()` 535
`replace_if()` 535
replicated base class 394
representation
 `hash_map` 498
 of container 465
requirement
 comparison 467
 copy 466
requirements for element 466
`reserve()` 455
reserved names 81
`reset()` `bitset` 494
`resetiosflags()` 634
`resize()` 52, 455
 `hash_map` 502
 of `string` 598
 of `valarray` 664
 `valarray` 666
resource
 acquisition 364
 exhaustion 369
 release 364
response to change 698
responsibility 700, 706
restricted character set 829
restriction 9
result
 of `sizeof` 122
 type 122
resumption 370
re-throw 362, 379
return
 `\r`, carriage 830
 by reference 283
 type and overload 151
 type of `virtual` 424
 type, relaxation of 424
 value 283
 value, algorithm 508
 value type check 148
 value type conversion 148
`return`
 alternative 357
 by reference 148
 by value 148
 function value 148
 of `void` expression 148
`return;` 148
`return_temporary_buffer()` 575
reuse 714
 design 709
 of concrete type 241
 of concrete type 768
reverse iterator 443, 557
`reverse()` 537
 `list` 472
`reverse_copy()` 537
`reverse_iterator` 443, 480, 557
 `basic_string` 583
reward 713
`rfind()` in `string` 594
`right` 625, 630
`right()` 634
Ritchie, Kernighan and 654
`rotate()` 537
`rotate_copy()` 537
`round_error()` 659
RTTI 407
 implementation of 409
 mis-use of 439
 misuse of 417
 use of 417
rule of two 741
run time support 8
run-time
 access control 785
 error 29, 355
 initialization 217
 polymorphism 347
 type identification 407
 type information 407, 774

S

`Saab` example 728
safety, convenience vs. 847
`Satellite` 390
saving space 840
`sbumpc()` 646
scale 212, 692
 problems of 715
scaling 665

scientific 626, 628
scientific() 634
scope 278
 and overload 151
 difference from C 816
 global 82, 847
 local 82
 resolution operator :: 82, 228
scrollbar example 743
search, binary 540, 546
search() 528
search_n() 528
seekdir
 and beg 643
 and cur 643
 and end 643
 direction of seek
seekg()
 direction of 643
seekoff()
seekp()
 direction of 643
 set position 642
seekpos()
selecting operations 705
selection from namespace 180
self, assignment to 246
self-reference this 230
semantics
 pointer 294
 value 294
sentry
 I/O 624
 iostream 624
separate
 compilation 27, 198
 compilation, template 351
separation of concerns 694
sequence 41, 469
 add element to 529
 adding to 555
 algorithm and 508
 algorithm, modifying 529
 algorithm, nonmodifying 523
 and associative container 461
 and container 512
 and range 512
 change size of 529
 delete element from 529, 534
 error 512
 fundamental 469
 generality of 512
 half-open 512
 input 513
 iterator and 550
 lexicographical_compare() of 544
 max_element() of 544
 min_element() of 544
 output 556
 set operation on 542
 sorted 539
 string 579
sequences, overlapping 529
set 124
 of class operations 237
 operation on sequence 542
 position, seekp()
<set> 431
Set example 769
set 491
 bitset and 492
 of Shape* 348
set()
 bitset 494
 function 759
setbase() 634
setbuf() 647
Set_controller example 785
set_difference() 543
setf() 626, 630
setfill() 634
setg() 645
set_intersection() 542
setiosflags() 634
<setjmp.h> 433
set_new_handler() 129, 576
setp() 645
setprecision() 633–634
setstate() 616
set_symmetric_difference() 543
set_terminate() 380
set_unexpected() 379
set_union() 542
setw() 634
sgetc() 646
sgetn() 646
Shakespeare 709
Shape
 example 774
 example 37
 example (bad) 417
Shape*, set of 348
shift() 664
short namespace name 178
short 73
short-circuit evaluation 123, 134
showbase 626, 628
showbase() 634
showmanyc() 647
showpoint 626
showpoint() 634
showpos 626

showpos() 634
shuffle 538
sign extension 831
signal 357
<signal.h> 157, 433
signaling_NaN() 659
signed
 char 831
 type 73
 unsigned integer conversion 834
Simula 10, 38
Simula-style container 438
simulation 685, 711
 event driven 326
sin() 660
 complex 680
 valarray 667
single quote \' 830
sinh() 660
 complex 680
 valarray 667
Sink iterator 532
size
 and delete 421
 and new 421
 of exponent 659
 of mantissa 659
 of number 75
 of pointer 75
 of program 8
 of sequence, change 529
 of string 147
 of structure 102
size() 455, 489, 494
 of string 598
 of valarray 664
 string 586
sizeof 75
 difference from C 816
 enum 78
 result of 122
size_t 122, 433
 -1 and 448
size_type 443, 480
 basic_string 583
skipws 625
skipws() 634
slice, generalized 677
slice 664, 668
slice_array 671
Slice_iter example 670
slicing 307
smallest int 658
Smalltalk 725
 style 417
Smalltalk-style container 438
smanip 633
smart pointer 289, 291
snextc() 646
software
 development 692
 maintenance 712
solution, generality of 701
sort 546
 partial 539
 stable 539
sort() 56, 539
 example 158, 334
 stable, list 470
sorted sequence 539
sort_heap() 543
sorting 338
 criteria 534
source
 code, template 350
 file 197
 of ideas, real-world as 734
space, saving 840
special character 830
specialization 859
 and char* 344
 and void* 341
 function 344
 generated 859
 order of 343
 partial 342
 partial, missing 823
 pattern 342
 template 341
 use of 865
 user 859
specialized, more 343
specifying interface 707
splice() 470
sputbackc() 646
sputc() 646
sputn() 646
sqrt() 660
 complex 680
 valarray 667
srand() 685
<sstream> 119, 432, 640
stability of design 708
stable
 list merge() 470
 list sort() 470
 sort 539
stable_partition() 542
stable_sort() 539
stack
 memory 843
 operator 450

<stack> 431
Stack example 27
stack
 implementation 475–476
 overflow 476
 pop() of 475
 push() of 475
 top() of 475
 underflow 476
stage
 analysis 697
 design 697
 development 697
 implementation 697
standard
 component 698, 714
 exception 384
 guarantees 827
 header 431
 include directory 201
 libraries 700
 library 45, 182
 library, C 599
 library, adding to 434
 library algorithms 64
 library container 56
 library criteria 430
 library design 429–430
 library facilities 66, 429
 library header 202
 library, missing 822
 library organization 431
 library predicate 516
 manipulator 633
 mathematical functions 660
 promotion 833
standardization, C++ 11
standard-library container 442
start, program 217
starting from scratch 708
state
 format 625
 machine 730
 of object 748
 stream 616
statement
 break 116
 continue 116
 controlled 136
 do 114, 137
 for 26, 136
 goto 137
 if 133
 loop 116
 summary 132
 switch 25, 133
 while 136
statement 802
state_type, char_traits 581
static
 memory 843
 type checking 727
static
 anachronism 200
 deprecated 818
 local 145
 member 228, 421
 member function 278
 member of template 854
 object 244
 store, local 251
static_cast 130, 159
 dynamic_cast and 413
std, namespace 46
std:: 46
<stdarg.h> 155, 433
<stddef> 433
<stddef.h> 433
<stdexcept> 385, 432
<stdio.h> 182, 202, 432
<stdlib.h> 432, 434, 546, 577, 600, 661
steps, design 701
STL 66
 container 441
 iterator 441
Storable example 396
storage
 class 244
 raw 574
store
 dynamic 34
 free 34, 127, 421, 576, 843
 heap 127
 local static 251
strcat() 599
strchr() 599
strcmp() 599
strcpy() 599
strcspn() 599
stream 432
 callback 650
 classes 637
 closing of 639
 file and 637
 hierarchy 637
 iterator 558
 state 616
 state, basic_ios 606
 string 640–641
<streambuf> 432
streambuf 646–647, 649
 << of 642

and character buffer 642
ostream and 642
streamoff 609
streamsize 609
stride() 668
string
and const, C-style 90
character 432
format 652
initialization of array by 89
literal 46, 90
size of 147
<string> 48, 432, 580
String example 328
string 48, 582
!= 591
+ 593
+= 592
< 591
<< 598
<= 591
= 587
== 591
> 591
>= 591
>> 598
[] on 584
algorithm and 584
and 0 587
and C-style string 579
and C-style string 589
and array 589
append() 592
as container 491
assign() 588
assignment 587
at() on 585
class 292
compare() 590
comparison 590
concatenation 592–593
constructor 585
conversion 589
design 579
empty 585
empty() 598
erase() in 595
error 586
find() in 594
find_first_not_of() in 594
find_first_of() in 594
find_last_of() in 594
get_allocator() from 598
getline() into 598
implicit conversion of 590
input 598
insert() 592
iterator 584
length() 586
length() of 598
literal 294
max_size() of 598
of user-defined type 583
out_of_range 586
output 598
range check of 584
replace() in 595
resize() of 598
rfind() in 594
sequence 579
size() 586
size() of 598
stream 640–641
subscripting of 584
substr() of 596
swap() 599
unsigned 583
stringbuf 649
<string.h> 432, 577, 599
stringstream 641
strlen() 599
strncat() 599
strncmp() 599
strncpy() 599
strpbrk() 599
strrchr() 599
strstr() 599
<strstream.h> 656
struct 101
and class 234
hack 809
name, difference from C 818
scope, difference from C 818
structure 101
initialization of 102
internal 694
of program 8
operations on 102
size of 102
style, programming 22
subarray 663, 671, 677–679
subarrays 668
subclass 303
superclass and 39
subrange 781
subscript
C++ style 674
Fortran style 674
subscripting 445, 454
comma and 838
map 482
of string 584

user-defined 286
valarray 663
substitution, Liskov 743
substr() of string 596
substring 596
Substring example 596
subtype 743
subtyping 730, 742
successful large system 709
suffix
_copy 533
_if 525
code 624
Sum 514
sum() of valarray 664
summary
algorithm 509
container 464
syntax 793
sungetc() 646
superclass 303
and subclass 39
supplying default value 500
support 714
run time 8
swap() 344, 457–458, 489, 538
string 599
swap_ranges() 538
switch
first-time 253, 640
last-time 640
on type 417
switch 109
and if 134
on enumeration 77
statement 25, 133
sync() 643, 647
sync_with_stdio() 651
synonym, see typedef
syntax
<, template 811
summary 793
system
growing 711
successful large 709
working 709

T

\t, horizontal tab 830
tab
\t, horizontal 830
\v, vertical 830
Table example 243
tan(), valarray 667
tanh() 660
complex 680
valarray 667
Task 394
taxonomy 703
teaching and C++ 12
technique
built-in feature vs 43
language, programming styles 6
tellg() 643
tellp() get position 642
template, use of 776
template 16, 40, 328, 854
and class 348
and friend 854
and generic programming 327
and inheritance 347
and ostream 608
argument 331
argument, deducing 335, 855
argument, default 340, 824
argument, depend on 861
argument, explicit 335
argument, function 335
as qualifier 858
as template parameter 855
class hierarchy and 345
copy constructor and 348
definition, context of 860
example:, member 349
function 334
in design 757
inclusion 350
inheritance and 349
instantiation 859
instantiation, context of 860
instantiation directive 866
instantiation, explicit 866
member 330
missing member 823
name binding 859
overloading, function 336
parameter 331
parameter, non-type 331
parameter, template as 855
parameterization 707
separate compilation 351
source code 350
specialization 341
static member of 854
syntax < 811
template-declaration 811
templates, relationships between 348
temporary 98
elimination of 675
lifetime of 254
object 244, 254

variable 244, 254
term, longer 699
`terminate()` 380
`terminate_handler` 380
termination 370
program 218
ternary operator 636
`test()` 494
testing 712
design for 712
`this` 278
self-reference 230
`throw` 186, 362, 379
`tie()` 623
time
constant 464
linear 464
logarithmic 464
quadratic 464
`<time.h>` 431, 433
`Tiny` 275
`tinyness_before` 659
`to_char_type()`, `char_traits` 581
`to_int_type()`, `char_traits` 581
tools, design 711
`top()`
of `priority_queue` 478
of `stack` 475
`to_ulong()` 494
`overflow_error` and 385
`toupper()` 591
traditional hierarchy 315
traits, character 580
`traits_type` 608
`basic_string` 583
`transform()` 530
transition 717–718
and *using-directive* 183
to namespace 182
translation unit 197
`traps` 659
traversal 61
tree 307
trigraphs 829
`true` and `1` 71
`trunc` truncate file 639
truncate file, `trunc` 639
truncation 835
`try` 187
try-block 187, 812
tutorial as design tool 708
two, rule of 741
type 23, 69
abstract 34, 767, 769
abstract and concrete 771
arithmetic 70
built-in 70
`char`, character 71
character 580
check, function argument 145
check, return value 148
checking, dynamic 727
checking, mis-use of dynamic 439
checking, static 727
class and 724
`class` user-defined 224
concrete 33, 236, 766–767
constructor for built-in 131
conversion, ambiguous 276
conversion, constructor and 269, 275
conversion, explicit 130
conversion, function argument 145
conversion, implicit 76, 276, 833
conversion operator 275
conversion, return value 148
conversion, unions and 842
conversion, user-defined 267, 281
equivalence 104
floating point 74
fundamental 23, 70
generator 348
identification, run-time 407
information, extended 416
information, run-time 407, 774
input of built-in 614
input of user-defined 621
integer 70, 73
integral 70
literal of user-defined 273
module and 30
of exception 379
of field 75
of integer literal 832
of integer literal, implementation dependency 832
of overriding function 424
of `virtual`, return 424
output of built-in 609
output of user-defined 612
pointer 569
problems with concrete 37
relaxation of return 424
result 122
reuse of concrete 241
safe I/O 607
`signed` 73
`string` of user-defined 583
switch on 417
`unsigned` 73
user-defined 32, 70
user-defined operator and built-in 265
`typedef` 84
type-field 308

typeid() 414
 bad_typeid and 384
<typeinfo> 384, 415, 433
type_info 414
typename 443, 856
type-safe linkage 198

U

uflow() 647
unary operator, user-defined 263
unary_function 515
unary_negate 518
 not1() and 522
unbuffered
 I/O 647
 input 642
 output 642
uncaught exceptions 380
uncaught_exception() 373
unchecked access 445
undeclared argument 154
#undef 162
undefined
 behavior 828
 enum conversion 77
underflow, stack 476
underflow() 647
unexpected exceptions 377
unexpected() 375
unexpected_handler 379
unformatted input 618
unget() 643
Unicode 580
uniform distribution 685
uniformity of code 767
uninitialized memory 574
uninitialized_copy() 574
uninitialized_fill() 574
uninitialized_fill_n() 574
union 841
 and class 843
 anonymous 841
 constructor and 257
 destructor and 257
 member 257, 843
 member object 244
 pointers and 845
 unnamed 841
unions and type conversion 842
unique() 532
 list 472
unique_copy() 56, 532
unit
 of allocation 88
 of compilation 197
 of design 755
 of protection 754
 translation 197
unitbuf 626
united of addressing 88
units analogy 728
universal
 base class 438
 character name 831
UNIX 8, 13
unnamed
 namespace 177, 200
 union 841
unsetf() 626
unsigned
 char 831
 integer conversion, signed 834
 string 583
 type 73
up cast 408
upper_bound() 540
 in map 485
uppercase 626
uppercase() 634
Urand 685
use
 case 704
 count 292
 dependency 745
 of C++ 12
 of RTTI 417
 of allocator 568
 of class 725
 of classes 733
 of dynamic_cast 774
 of global variable 111
 of map 774
 of multiple inheritance 776
 of rebind 569
 of specialization 865
 of template 776
used function only, instantiate 866
user specialization 859
user-defined
 + 265
 == 534
 allocator 570
 binary operator 263
 container 497
 conversion 347
 iterator 561
 manipulator 635
 memory management, example of 292
 operator 263
 operator + 281
 operator ++ 264, 291

operator += 264, 268, 281
operator -- 291
operator -> 289
operator = 281
operator and built-in type 265
operator and enum 265
pointer conversion 349
predicate 516
subscripting 286
type 32, 70
type, class 224
type conversion 267, 281
type, input of 621
type, literal of 273
type, output of 612
type, string of 583
unary operator 263
user-supplied comparison 467
uses of parentheses 123
using multiple inheritance 399
using
namespace 183
namespace, using vs. 847
vs. using namespace 847
using-declaration 169, 180
and access control 407
and inheritance 392
vs. *using-directive* 847
using-directive 171
and definition 180
and inheritance 392
transition and 183
using-declaration vs. 847
usual arithmetic conversions 122
utilities 431
<utility> 431, 468

V

\v, vertical tab 830
va_arg() 155
<valarray> 434, 662
valarray 65, 662
! 664
!= 667
% 667
%= 664
& 667
&& 667
&= 664
* 667
*= 664
+ 667
+= 664
- 664, 667
-= 664
/ 667
/= 664
< 667
<< 667
<<= 664
<= 667
= 663
== 667
> 667
>= 667
>> 667
>>= 664
[] 663
^ 667
^= 664
abs() 667
acos() 667
and array 663
and vector and array 662
apply() to 664
as container 492
asin() 667
assignment 663
atan() 667
atan2() 667
construction 662
cos() 667
cosh() 667
exp() 667
input 668
iterator 670
length of 664, 679
log() 667
log10() 667
mathematical functions 667
max() 664
min() 664
operations 664, 667
output 668
pow() 667
range check 664
resize() 666
resize() of 664
sin() 667
sinh() 667
size() of 664
sqrt() 667
subscripting 663
sum() of 664
tan() 667
tanh() 667
| 667
|= 664
|| 667
~ 664
valid iterator 550

value
call by 146
default 239
key and 480
mapped type 55
of character 580
of cin 276
of notation 261
return 283
return by 148
return, function 148
semantics 294
value_comp() 485
value_compare 485
value_type 443, 480, 552
basic_string 583
variable
constructor for global 252
constructor for local 245
global 200, 228
temporary 244, 254
variably-sized object 243
Vec, range checking 53
vector
Fortran 668
arithmetic 65, 662
bit 124
exponentiation 667
mathematical functions 667
operations 664, 667
<vector> 431
Vector 435
example 341, 780
vector 52, 442, 469
< 457
= 447
== 457
[] of 445
and array, valarray and 662
assign() 447
constructor 447
erase() from 452
input into 451
insert() into 452
member type 442
of bool 458
of vector 836
vector of 836
vector<bool> 458
bitset and 492
Vehicle example 734
vertical tab \v 830
viewgraph engineering 704
virtual
function 15
function, renaming 778
virtual 34
<< 612
base class 396
base class, overriding from 401
base, constructor and 397
constructor 323, 424
derive without 780
destructor 319
function 310, 390, 706
function argument types 310
function, definition of 310
function, example of 646
function, implementation of 36
function, operator :: and 312
function, pure 313
output function 612
return type of 424
vision 698
void 76
expression, return of 148
pointer to 100
void*
assignment, difference from C 818
specialization and 341
void*(), operator 616
volatile 808

W

waterfall model 697
wcerr 609
<wchar.h> 432
wchar_t 72–73
wcin 614
wcout and 624
wclog 609
wcout 609
and wcin 624
wfilebuf 649
wfstream 638
while statement 136
whitespace 614–615
isspace() 114
wide
character I/O 608
character classification 601
wide-character literal L' 73
widen() 645
width() 629
of input 616
wifstream 638
wild pointer checking for 722
Window example 398
wiostream 637
wistream 614
wistringstream 641

`wofstream` 638
word 76
working system 709
`wostream` 608
`wostringstream` 641
wrapper 781
write through iterator 551
`write()`, `ostream` 609
`ws` 634
`wstreambuf` 649
`wstring` 582
`wstringbuf` 649
`wstringstream` 641
`<wtype.h>` 432

X

X3J16 11
`xalloc()` 650
`xgetn()` 647
`xor` keyword 829
`xor_eq` keyword 829
`xputn()` 647

Y

`Year` 285

Z

zero null, `0` 88

AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

Data Abstraction in C

By B. STROUSTRUP*

(Manuscript received August 5, 1983)

C++ is a superset of the C programming language; it is fully implemented and has been used for nontrivial projects. There are now more than one hundred C++ installations. This paper describes the facilities for data abstraction provided in C++. These include Simula-like classes providing (optional) data hiding, (optional) guaranteed initialization of data structures, (optional) implicit type conversion for user-defined types, and (optional) dynamic typing; mechanisms for overloading function names and operators; and mechanisms for user-controlled memory management. It is shown how a new data type, like complex numbers, can be implemented, and how an "object-based" graphics package can be structured. A program using these data abstraction facilities is at least as efficient as an equivalent program not using them, and the compiler is faster than older C compilers.

I. INTRODUCTION

The aim of this paper is to show how to write C++ programs using "data abstraction", as described below†. This paper presents some general discussion of each new language feature to help the reader

* AT&T Bell Laboratories.

† Note on the name C++: ++ is the C increment operator; when this operator is applied to a variable (typically a vector index or a pointer), it increments the variable so that it denotes the succeeding element. The name C++ was coined by Rich Mascitti. Consider ++ a surname, to be used only on formal occasions or to avoid ambiguity. Among friends C++ is referred to as C, and the C language described in the C book[1] is "old C". The slightly shorter name C+ is a syntax error; it has also been used as the

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

understand where that feature fits in the overall design of the language, which programming techniques it is intended to support, and what kinds of errors and costs it is intended to help the programmer avoid. However, this paper is not a reference manual, so it does not give complete details of the language primitives; these can be found in Ref. 3.

C++ evolved from C[1] through some intermediate stages, collectively known as "C with classes".[4,5] The primary influence on the design of the abstraction facilities was the Simula67 class concept.[6,7] The intent was to create data abstraction facilities that are both expressive enough to be of significant help in structuring large systems, and at the same time useful in areas where C's terseness and ability to express low-level detail are great assets. Consequently, while C classes provide general and flexible structuring mechanisms, great care has been taken to ensure that their use does not cause run time or storage overhead that could have been avoided in old C.

Except for details like the introduction of new key words, C++ is a superset of C; see Section XXII, "Implementation and Compatibility" below. The language is fully implemented and in use. Tens of thousands of lines of code have been written and tested by dozens of programmers.

The paper falls into three main sections:

1. A brief presentation of the idea of data abstraction.

2. A full description of the facilities provided for the support of that idea through the presentation of small examples. This in itself falls into three sections:

a. Basic techniques for data hiding, access to data, allocation, and initialization. Classes, class member functions, constructors, and function name overloading are presented (starts with Section III, "Restriction of Access to Data").

b. Mechanisms and techniques for creating new types with associated operators. Operator overloading, user-defined type conversion, references, and free store operators are presented (starts with Section VIII, "Operator Overloading and Type Conversion").

c. Mechanisms for creating abstraction hierarchies, for dynamic typing of objects, and for creating polymorphic classes and functions. Derived classes and virtual functions are presented (starts with Section XIV, "Derived Classes").

Items b and c do not depend directly on each other.

---

name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C, but the latter is not an acceptable name. The language is not called D, since it is an extension of C and does not attempt to remedy problems inherent in the basic structure of C. The name C++ signifies the evolutionary nature of the changes from old C. For yet another interpretation of the name C++ see the Appendix of Ref. 2.

3. Finally some general observations on programming techniques, on language implementation, on efficiency, on compatibility with old C, and on other languages are offered (starts with Section XVIII, "Input and Output").

A few sections are marked as "digressions"; they contain information that, while important to a programmer, and hopefully of interest to the general reader, does not directly relate to data abstraction.

II. DATA ABSTRACTION

"Data abstraction" is a popular, but generally ill-defined, technique for programming. The fundamental idea is to separate the incidental details of the implementation of a subprogram from the properties essential to the correct use of it. Such a separation can be expressed by channeling all use of the subprogram through a specific "interface". Typically the interface is the set of functions that may access the data structures that provide the representation of the "abstraction". One reason for the lack of a generally accepted definition is that any language facility supporting it will emphasize some aspects of the fundamental idea at the expense of others. For example:

1. Data hiding—Facilities for specifying interfaces that prevent corruption of data and relieve a user from the need to know about implementation details.

2. Interface tailoring—Facilities for specifying interfaces that support and enforce particular conventions for the use of abstractions. Examples include operator overloading and dynamic typing.

3. Instantiation—Facilities for creating and initializing of one or more "instances" (variables, objects, copies, versions) of an abstraction.

4. Locality—Facilities for simplifying the implementation of an abstraction by taking advantage of the fact that all access is channeled through its interface. Examples include simplified scope rules and calling conventions within an implementation.

5. Programming environment—Facilities for supporting the construction of programs using abstractions. Examples include loaders that understand abstractions, libraries of abstractions, and debuggers that allow the programmer to work in terms of abstractions.

6. Efficiency—A language facility must be "efficient enough" to be useful. The intended range of applications is a major factor in determining which facilities can be provided in a language. Conversely, the efficiency of the facilities determines how freely they can be used in a given program. Efficiency must be considered in three separate contexts: compile time, link time, and run time.

The emphasis in the design of the C data abstraction facility was on 2, 3, and 6, that is, on facilities enabling a programmer to provide

elegant and efficient interfaces to abstractions. In C, data abstraction is supported by enabling the programmer to define new types, called "classes". The members of a class cannot be accessed, except in an explicitly declared set of functions. Simple data hiding can be achieved like this:

```
class data_type {
    /* data declarations */
    /* list of functions that may use
       the data declarations ("friends") */
};
```

where only the "friends" can access the representation of variables of class `data_type` as defined by the data declarations. Alternatively, and often more elegantly, one can define a data type where the set of functions that may access the representation is an integral part of the type itself:

```
class object_type {
    /* declarations used to implement object_type */
public:
    /* declarations specifying
       the interface to object_type */
};
```

One obvious, but nontrivial, aim of many modern language designs is to enable programmers to define "abstract data types" with properties similar to the properties of the fundamental data types of the languages. Below we show how to add a data type `complex` to the C language, so that the usual arithmetic operators can be applied to complex variables. For example:

```
complex a, x, y, z;
a = x/y + 3*z;
```

The idea of treating an object as a black box is further supported by a mechanism for hierarchically constructing classes out of other classes. For example:

```
class shape { ··· };
class circle : shape { ··· };
```

The class `circle` can be used as a simple `shape` in addition to being used as a `circle`. Class `circle` is said to be a derived class with class `shape` as its base class. It is possible to leave the resolution of the type of objects sharing common base classes to run time. This allows objects of different types to be manipulated in a uniform manner.

III. RESTRICTION OF ACCESS TO DATA

Consider a simple old C fragment,† outlining an implementation of the concept of a date:

```
struct date { int day, month, year; };
struct date today;
extern void set_date();
extern void next_date();
extern void next_today();
extern void print_date();
```

There are no explicit connections between the functions and the data type, and no indication that these functions should be the only ones to access the members of the structure `date`. It ought to be possible to state such an intent.

A simple way of doing this is to declare a data type that can only be manipulated by a specific set of functions. For example:

```
class date {
    int day, month, year;
    friend void set_date(date*, int, int, int),
                next_date(date*),
                next_today(),
                print_date(date*);
};
```

The key word `class` indicates that only functions mentioned as "friends" in the declaration can use the class member names `day`, `month`, and `year`; otherwise a `class` behaves like a traditional C `struct`. That is, the class declaration itself defines a new type of which variables can be declared. For example:

```
date my_birthday, today;

set_date(&my_birthday,30,12,1950);
set_date(&today,23,6,1983);
print_date(&today);
next_date(&today);
```

Friend functions are defined in the usual manner. For example:

```
void next_date(date* d)
{
    if ( ++d->day > 28 ) {
```

† The key word *void* specifies that a function does not return a value. It was introduced into C about 1980.

```
        /* do the hard part */
    }
}
```

This solution to the problem of data hiding is simple, and often quite effective. It is not perfectly flexible because it allows access by the "friends" to all variables of a type. For example, it is not possible to have a different set of friends for the dates `my_birthday` and `today`. A function can, however, be the friend of more than one class. The importance of this will be demonstrated in Section XIX. There is no requirement that a friend should only manipulate variables passed to it as arguments. For example, the name of a global variable may be built into a function:

```
void next_today()
{
    if ( ++today.day > 28 ) {
        /* do the hard part */
    }
}
```

The protection of the data from functions that are not friends relies on restricting the use of class member names. It can therefore be circumvented by address manipulation and explicit type conversion.

There are several benefits to be obtained from restricting a data structure's access to an explicitly declared list of functions. Any error causing an illegal state of a `date` must be caused by code in the friend functions, so the first stage of debugging, localization, is completed before the program is even run. This is a special case of the general observation that any change to the behavior of the type `date` can and must be effected by changes to its friends. Another advantage is that a potential user of such a type need only examine the definition of the friends to learn to use it. Experience with C++ has amply demonstrated this.

IV. DIGRESSION: ARGUMENT TYPES

The argument types of the functions above were declared. This could not have been done in old C, nor would the matching function definition syntax used for `next_date` have been accepted. In C++ the semantics of argument passing are identical to those of initialization. In particular, the usual arithmetic conversions are performed. A function declaration that does not specify an argument type, for example `next_today()`, specifies that the function does not accept any arguments. This is different from old C; see Section XXII, "Implementa-

tion and Compatibility" below. The argument types of all declarations and the definition of a function must match exactly.

It is still possible to have functions that take an unspecified and possibly variable number of arguments of unspecified types, but such relaxation of the type checking must be explicitly declared. For example:

```
int wild(···);
int fprintf(FILE*, char* ···);
```

The ellipsis specifies that any arguments (or none) will be accepted without any checking or conversion exactly as in old C. For example:

```
wild(); wild("asdf",10); wild(1.3,"ghjk",wild);
fprintf(stdout,"x=%d",10);
fprintf(stderr,"file %s line %d\n", f_name, l_no);
```

Note that the first two arguments of `fprintf` must be present and will be checked. It has been noted, however, that functions with partly specified argument types are far less useful in C++ than they are in old C. Such functions are primarily useful for specifying interfaces to old C libraries. Default function arguments (Section IX), overload function names (Section VII), and operator overloading (Section VIII) are used instead. See also Section XVIII.

As ever, undeclared functions may be used and will be assumed to return integers. They must, however, be used consistently. For example:

```
undef1(1, "asdf"); undef1(2, "ghjk"); /* fine */
undef2(1, "asdf"); undef2("ghjk", 2); /* error */
```

The inconsistent use of `undef2` is detected by the compiler.

V. OBJECTS

The structure of a program using the *class/friend* mechanism to restrict access to the representation of a data type is exactly the same as the structure of a program not using it. This implies that no advantage has been taken of the new facility to make the functions implementing the operations on the type easier to write. For many types, a more elegant solution can be obtained by incorporating such functions into the new type itself. For example:

```
class date {
    int day, month, year;
public:
    void set(int, int, int);
    void next();
```

```
    void print();
};
```

Functions declared this way are called member functions and can be invoked only for a specific variable of the appropriate type using the standard C structure member syntax. Since the function names no longer are global, they can be shorter:

```
my_birthday.print();
today.next();
```

On the other hand, to define a member function, one must specify both the name of the function and the name of its class:

```
void date.next()
{
    if ( ++day > 28 ) {
        /* do the hard part */
    }
}
```

Variables of such types are often referred to as objects. The object for which the function is invoked constitutes a hidden argument to the function. In a member function, class member names can be used without explicit reference to a class object. In that case, like the use of `day` above, the name refers to that member of the object for which the function was invoked. A member function sometimes needs to refer explicitly to this object, for example to return a pointer to it. This is achieved by having the key word `this` denote that object in every class function. Thus, in a member function `this->day` is equivalent to `day` for every member of the class `date`.

The `public` label separates the class body into two parts. The names in the first, "private", part can only be used by member functions (and friends). The second, "public", part constitutes the interface to objects of the class. A class function may access both public and private members of every object of its class, not just members of the one for which it was invoked.

The relative merits of friends and member functions will be discussed in Section XIX after a larger body of examples has been presented. For now, it is sufficient to notice that a friend is not affected by the "public/private" mechanism and operates on objects in a standard and explicit manner. A member, on the other hand, must be invoked for an object and treats that object differently from all others.

VI. STATIC MEMBERS

A class is a type, not a data object, and each object of the class has its own copy of the data members of the class. However, there are

concepts (abstractions) that are best supported if the different objects of the class share some data. For example, to manage tasks in an operating system or a simulation, a list of all tasks is often useful:

```
class task {
    ...
    task* next;
    static task* task_chain;
    void schedule(int);
    void wait(event);
    ...
};
```

Declaring the member `task_chain` as `static` ensures that there will only be one copy of it, not one copy per task object. It is still in the scope of class `task`, however, and can only be accessed from "the outside" if it was declared public. In that case its name must be qualified by its class name:

```
task::task_chain
```

In a member function it can be referred to as plain `task_chain`. The use of `static` class members can reduce the need for global variables considerably.

The operator :: (colon colon) is used to specify the scope of a name in expressions. As a unary operator it denotes external (global) names. For example, if the task function `wait` in a simulator needs to call a nonmember function `wait`, it can be done like this:

```
void task.wait(event e)
{
    ...
    ::wait(e);
}
```

VII. CONSTRUCTORS AND OVERLOADED FUNCTIONS

The use of functions like `set_date()` to provide initialization for class objects is inelegant and error prone. Since it is nowhere stated that an object must be initialized, a programmer can forget to do so or, often with equally disastrous results, do so twice. A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type, it is called a constructor. A constructor is recognized by having the same name as the class itself. For example:

```
class date {
    ...
```

```
    date(int, int, int);
};
```

When a class has a constructor all objects of that class must be initialized:

```
date today = date(23, 6, 1983);
date xmas(25, 12, 0);      /* legal abbreviated form */
date july4 = today;
date my_birthday;    /* illegal, initializer missing */
```

It is often nice to provide several ways of initializing a class object. This can be done by providing several constructors. For example:

```
class date {
    ...
date(int, int, int);            /* day month year */
date(char*);              /* date in string representation
date(int);            /* day, assume current month and year
date();               /* default date: today */
};
```

As long as the constructor functions differ in their argument types, the compiler can select the correct one for each use:

```
date today(4);
date july4("July 4, 1983");
date guy("5 Nov");
date now;                       /* default initialized */
```

Constructors are not restricted to initialization, but can be used wherever it is meaningful to have a class object:

```
date us_date(int month, int day, int year)
{
    return date(day, month, year);
}
...
some_function( us_date(12,24,1983) );
some_function(    date(24,12,1983) );
```

When several functions are declared with the same name, that name is said to be overloaded. The use of overloaded function names is not restricted to constructors. However, for nonmember functions the function declarations must be preceded by a declaration specifying that the name is to be overloaded; for example:

```
overload print;
```

```
void print(int);
void print(char*);
```

or possibly abbreviated like this:

```
overload void print(int), print(char*);
```

As far as the compiler is concerned, the only thing common for a set of functions of the same name is that name. Presumably they are in some sense similar, but the language does not constrain or aid the programmer. Thus, overloaded function names are primarily a notational convenience. This convenience is significant for functions with conventional names like `sqrt`, `print`, and `open`. Where a name is semantically significant, as in the case of constructors, this convenience becomes essential. For example, consider writing a single constructor for class `date` above.

For arguments to functions with overloaded names the C type conversion rules do not apply fully. The conversions that may destroy information are not performed, leaving only `char->short->int->long`, `float->double`, and `int->double`. It is, however, possible to provide different functions for integral and floating types. For example:

```
overload print(int), print(double);
```

The list of functions for an overloaded name will be searched in order of appearance for a match, so that `print(1)` will invoke the integer print function, and `print(1.0)` the floating-point print function. Had the order of declaration been reversed, both calls would have invoked the floating-point print function with the `double` representation of 1.

VIII. OPERATOR OVERLOADING AND TYPE CONVERSION

Some languages provide a `complex` data type, so that programmers can use the mathematical notion of complex numbers directly. Since C does not, it is an obvious test of an abstraction facility to see to what extent the conventional notion of complex numbers can be supported (Note, however, that `complex` is an unusual data type in that it has an extremely simple representation and there are very strong traditions for its proper use. It is, therefore, primarily a test of the abstraction facility's power to imitate conventional notation. In most other cases the designer's attention will be directed towards finding a good representation of the abstraction and towards finding a suitable way of presenting the abstraction to its users.) The aim of the exercise is to be able to write code like this:

```
complex x;
complex a = complex(1, 1.23);
```

```
complex b = 1;
complex c = PI;

if (x!=a) x = a+log(b*c)/2;
```

That is, the standard arithmetic and comparison operators must be defined for complex numbers and for mixtures of complex and scalar constants and variables.

Here is a declaration of a very simple class `complex`:

```
class complex {
    double re, im;

    friend complex operator+   (complex, complex);
    friend complex operator*   (complex, complex);
    friend int     operator!=  (complex, complex);
public:
    complex()                      { re=im=0; }
    complex(double r)              { re=r; im=0; }
    complex(double r, double i)    { re=r; im=i; }
};
```

An operator is recognized as a function name when it is preceded by the key word `operator`. When an operator is used for a class type, the compiler will generate a call to the appropriate function, if declared. For example, for complex variables `xx` and `yy` the addition `xx+yy` will be interpreted as `operator+(xx,yy)`, given the declaration of class `complex` above. The complex add function could be defined like this:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re, a1.im+a2.im);
}
```

Naturally, all names of the form `operator@` are overloaded. To ensure that the language is only extendable and not mutable, an operator function must take at least one class object argument. By declaring operator functions the programmer can assign meaning to the standard C operators applied to objects of user-specified data types. These operators retain their usual places in the C syntax, and it is not possible to add new operators. It is, therefore, not possible to change the precedence of an operator or to introduce a new operator (for example, `**` for exponentiation). This restriction keeps the analysis of C expressions simple.

Declarations of functions for unary and binary operators are distinguished by their number of arguments. For example:

```
class complex {
    ...
    friend complex operator-(complex);
    friend complex operator-(complex, complex);
};
```

There are three ways the designer of class `complex` could decide to handle mixed-mode arithmetic, like `xx+1`, where `xx` is a complex variable. It can simply be considered illegal, so that the user has to write the conversion from `double` to `complex` explicitly: `xx+complex(1)`. Alternatively, several complex add functions may be specified:

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);
```

so that the compiler will choose the appropriate function for each call. Finally, if a class has constructors that take a single argument, then they will be taken to define conversions from their argument type to the type for which they construct values. Thus, with the declaration of class `complex` above `xx+1` would automatically be interpreted as `operator+(xx,complex(1))`.

This last alternative violates many people's idea of strong typing. However, using the second solution will nearly triple the number of functions needed and the first provides little notational convenience to the user of class `complex`. Note that complex numbers are typical with respect to the desirability of mixed-mode arithmetic. A typical data type does not exist in a vacuum. Furthermore, for many types there exists a trivial mapping from the C numeric and/or string constants into a subset of the values of the type (similar to the mapping of the C numeric constants into the complex values on the real axis).

The `friend` approach was chosen in favor of using member functions for the `operator` functions. The inherent asymmetry in the notion of objects does not match the traditional mathematical view of complex numbers.

IX. DIGRESSION: DEFAULT ARGUMENTS AND INLINE FUNCTIONS

Class complex had three constructors, two of which simply provided the default value zero for notational convenience of the programmer. This use of overloading is typical for constructors, and also has been found to be quite common for other functions. However, overloading is a quite elaborate and indirect way of providing default argument values and, in particular for more complicated constructors, quite verbose. Consequently, a facility for expressing default arguments directly is provided. For example:

```
class complex {
    ...
public:
    complex(double r = 0, double i = 0) { re=r; im=i; }
};
```

When a trailing argument is missing the default constant expression can be used. For example:

```
complex a(1,2);
complex b(1);        /* b = complex(1,0) */
complex c;           /* c = complex(0,0) */
```

When a member function, like `complex` above, is not only declared, but also defined (that is, its body is presented) in a class declaration, it may be inline substituted when called, thus eliminating the usual function call overhead. An inline substituted function is not a macro; its semantics are identical to other functions. Any function can be declared inline by preceding its definition by the key word `inline`. Inline functions can make class declarations quite untidy; they will only improve run-time efficiency if used judiciously, and will always increase the time and space needed to compile a program. They should therefore be used only when a significant improvement of run-time is expected. They are included in C++ because of experience with C macros. Macros are sometimes essential for an application (and it is not possible to have a class member macro), but more often they create chaos by appearing to be functions without obeying the syntax, scope, and argument passing rules of functions.

X. STORAGE MANAGEMENT

There are three storage classes in C++: static, automatic (stack), and free (dynamic). Free store is managed by the programmer through the operators `new` and `delete`. No standard garbage collector is provided.†

Constructors are handy for hiding details of free store management. For example:

```
class string {
    char* rep;
```

† It is, however, not that difficult to write a garbage-collecting implementation of the *new* operator, as has been done for the old C free store allocator function `malloc()`. It is not in general possible to distinguish pointers from other data items when looking at the memory of a running C program, so a garbage collector must be conservative in its choice of what to delete, and it must examine unappealingly large amounts of data. They have been found useful for some applications, though.

```
    string(char*);
    ~string()       { delete rep; }
    ...
};

string.string(char* p)
{
    rep = new char[strlen(p)+1];
    strcpy(rep,p);
}
```

Here the use of free store is encapsulated in the constructor `string()` and its inverse, the destructor `~string()`. Destructors are implicitly called when an object goes out of scope. They are also called when an object is explicitly deleted by `delete`. For static objects destructors are called after all parts of the program as the program terminates. The `new` operator takes a type as its argument and returns a pointer to an object of that type; `delete` takes such a pointer as argument. A `string` may itself be allocated on the free store. For example:

```
string* p = new string("asdf");
delete p;
p = new string("qwerty");
```

It is furthermore possible for a class to take over the free store management for its objects. For example:

```
class node {
    int type;
    node* l;
    node* r;
    node()    { if (this==0) this = new_node(); }
   ~node()    { free_node(this); this = 0; }
    ...
};
```

For an object created by `new`, the `this` pointer will be zero when a constructor is entered. If the constructor does not assign to `this` the standard allocator function is used. The standard deallocator function will be used at the end of a destructor if and only if `this` is nonzero. An allocator provided by the programmer for a specific class or set of classes can be much simpler and at least an order of magnitude faster than the standard allocator.

Using constructors and destructors, the designer may specify data types, like `string` above, where the size of the representation of an object can vary, even though the size of every static and automatic variable must be known at load time and compile time, respectively.

The class object itself is of fixed size, but its class maintains a variable-sized secondary data structure.

XI. HIDING STORAGE MANAGEMENT

Constructors and destructors cannot completely hide storage management details from the user of a class. When an object is copied, either by explicit assignment or by passing it as a function argument, the pointers to secondary data structures are copied too. This is sometimes undesirable. Consider the problem of providing value semantics for a simple data type `string`. A user sees a `string` as a single object, but the implementation consists of two parts, as outlined above. After the assignment `s1=s2` both strings refer to the same representation, and the store used for the old representation of `s1` is unreferenced. To avoid this, the assignment operator can be overloaded.

```
class string {
    char* rep;
    void operator=(string);
    ...
};

void string.operator=(string source)
{
    if (rep != source.rep) {
        delete rep;
        rep = new char[ strlen(source.rep)+1 ];
        strcpy(rep,source.rep);
    }
}
```

Since the function needs to modify the target `string`, it is best written as a member function taking the source `string` as argument. The assignment `s1=s2` will now be interpreted as `s1.operator=(s2)`.

This leaves the problem of what to do with initializers and function arguments. Consider

```
string s1 = "asdf";
string s2 = s1;
do_something(s2);
```

This leaves the strings `s1`, `s2`, and the argument of `do_something` with the same `rep`. The standard bitwise copy clearly does not preserve the desired value semantics for strings.

The semantics of argument passing and initialization are identical; both involve copying an object into an uninitialized variable. They

differ from the semantics of assignment (only) in that an object assigned to is assumed to contain a value, and an object being initialized is not. In particular, constructors are used in argument passing exactly as in initialization. Consequently, the undesirable bitwise copy can be avoided if we can specify a constructor to perform the proper copy operation. Unfortunately, using the obvious constructor

```
class string {
    ...
    string(string);
}
```

leads to infinite recursion. It is therefore illegal. To solve this problem, a new type "reference" is introduced. It is syntactically identified by the declarator &, which is used in the same way as the pointer declarator *. When a variable is declared to be a `T&`, that is a reference to `T`, it can be initialized either by a pointer to type `T` or an object of type `T`. In the latter case the address of operator & is implicitly applied. For example:

```
int x;
int& r1 = &x;
int& r2 = x;
```

assigns the address of `x` to both `r1` and `r2`. When used, a reference is implicitly dereferenced; so, for example:

```
r1 = r2
```

means copy the object pointed to by `r2` into the object pointed to by `r1`. Note that initialization of a reference is quite different from assignment to it.

Using references class `string` can now be declared like this:

```
class string {
    char* rep;
    string(char*);
    string(string&);
    ~string();
    void operator=(string&);
    ...
};

string(string& source)
{
    rep = new char[ strlen(source.rep)+1 ];
    strcpy(rep,source.rep);
}
```

Initialization of one string with another (and passing a string as an argument) will now involve a call of the constructor `string (string&)` that will correctly duplicate the representation. The `string` assignment operator was redeclared to take advantage of references. For example:

```
void string.operator=(string& source)
{
    if (this != &source) {
        delete rep;
        rep = new char[ strlen(source.rep)+1 ];
        strcpy(rep,source.rep);
    }
}
```

This type `string` will not be efficient enough for many applications. It is, however, not difficult to modify it so that the representation is only copied when necessary and shared otherwise.

XII. FURTHER NOTATIONAL CONVENIENCE

It is curious that references, a facility with great similarity to the "call by reference" rules for argument passing in many languages, are introduced primarily to enable a programmer to specify "call by value" semantics for argument passing. They have several other uses as well, however, including of course "by reference" argument passing. In particular, references provide a way of having nontrivial expressions on the left-hand side of assignments. Consider a `string` type with a substring operator:

```
class string {
    ...
    void      operator=(string&);
    void      operator=(char*);
    string&   operator()(int pos, int length);
};
```

where `operator()` denotes function application. For example,

```
string s1 = "asdf";
string s2 = "ghjkl";
s1(1,2) = "xyz";        /* s1 = "axyzf" */
s2 = s1(0,3);           /* s2 = "axy"   */
```

The two assignments will be interpreted as:

```
( s1.operator()(1,2) )->operator=("xyz");
s2.operator=( s1.operator()(0,3) );
```

The `operator()` function need not know whether it is invoked on the

left-hand or the right-hand side of the assignment. The `operator=` function can take care of that.

Vector element selection can be similarly overloaded by defining `operator[]`.

XIII. DIGRESSION: REFERENCES AND TYPE CONVERSION

Conversions defined for a class are applied even when references are involved. Consider a class `string` where assignment of simple character strings is not defined, but the construction of a string from such a character string is:

```
class string {
        ...
            string(char*);
        void operator=(string&);
};

string s = "asdf";
```

The assignment

```
s = "ghjk";
```

is legal, and will produce the desired effect. It is interpreted as

```
s.operator=( (temp.string("ghjk"),&temp) )
```

where `temp` is a temporary variable of type `string`. Applying constructors before taking the address as required by the reference semantics ensures that the expressive power provided by constructors is not lost for variables of reference type. In other words, the set of values accepted by a function expecting an argument of type `T` is the same as that accepted by a function expecting a `T&`(reference to `T`).

XIV. DERIVED CLASSES

Consider writing a system for managing geometric shapes on a terminal screen. An attractive approach is to treat each shape as an object that can be requested to perform certain actions like "rotate" and "change color". Each object will interpret such requests in accordance with its type. For example, the algorithm for rotation is likely to be different (simpler) for a circle than for a triangle. What is needed is a single interface to a variety of co-existing implementations. The different kind of shapes cannot be assumed to have similar representations. They may differ widely in complexity, and it would be a pity to be unable to utilize the inherent simplicity of basic shapes like circle and triangle because of the need to support complex shapes like "mouse" and "British Isles".

The general approach is to provide a class `shape` defining the common properties of shapes, in particular a "standard interface". For example:

```
class shape {
    point    center;
    int      color;
    shape*   next;
    static   shape* shape_chain;
    ...
public:
    void     move(point to) { center=to; draw(); }
    point    where()        { return center; }
    virtual  void rotate(int);
    virtual  void draw();
        ...
};
```

The functions that cannot be implemented without knowledge of the specific `shape` are declared `virtual`. A virtual function is expected to be defined later. At this stage only its type is known; this, however, is sufficient to check calls to it.

A class defining a particular shape may be defined like this:

```
class circle : public shape {
    float radius;
public:
    void rotate(int angle) {}
    void draw();
    ...
};
```

This specifies a `circle` to be a `shape`, and as such it has all the members of class `shape` in addition to its own members. The class `circle` is said to be derived from its "base class" `shape`. Circles can now be declared and used:

```
circle cl;
shape* sh;
point p(100,30);

cl.draw();
cl.move(p);
sh = &cl;
sh->draw();
```

Naturally the function called by `cl.draw()` is `circle::draw()`, and since `circle` did not define its own `move()`, the function called by

`cl.move(p)` is `shape::move()`, which class `circle` inherited from class `shape`. However, the function called by `sh->draw()` is also `circle::draw()`, despite the fact that no reference to class `circle` is found in the declaration of class `shape`. A virtual function is redefined when a class is derived from its class. Each object of a class with virtual functions contains a type indicator. This enables the compiler to find the proper virtual function for a call even when the type of the object is not known at compile time. Calling a virtual function is the only way of using the hidden type indicator in a class (a class without virtual functions does not have such an indicator).

A shape may also provide facilities that cannot be used unless the programmer knows its particular type. For example:

```
class clock_face : public circle {
    line   hour_hand, minute_hand;
public:
    void   draw();
    void   rotate(int);
    void   set(int, int);
    void   advance(int);
    ...
}
```

The time displayed by the clock can be `set()` to a particular time, and one can `advance()` the displayed time a number of minutes. The `draw()` in `clock_face` hides `circle::draw()`, so that the latter can only be called by its full name. For example:

```
void clock_face.draw() {
    circle::draw();
    hour_hand.draw();
    minute_hand.draw();
}
```

Note that a virtual function must be a member. It cannot be a friend, and there is no equivalent in the *class/friend* style of programming to the use of dynamic typing presented here and in the following section.

XV. DIGRESSION: STRUCTURES AND UNIONS

The C constructs `struct` and `union` are legal, but conceptually absorbed into classes. A `struct` is a class with all members public; that is

```
struct s { ··· };
```

is equivalent to

```
class s { public: ··· };
```

A `union` is a `struct` that can hold exactly one data member at a time.

These definitions imply that `struct` or a `union` can have function members. In particular, they can have constructors. For example:

```
union uu {
    int   i;
    char* p;
    uu(int ii)    { i=ii; }
    uu(char* pp)  { p=pp; }
};
```

This takes care of most problems concerning initialization of unions. For example:

```
uu u1 = 1;
uu u2 = "asdf";
```

XVI. POLYMORPHIC FUNCTIONS

By using derived classes, one can design interfaces providing uniform access to objects of unknown and/or different classes. This can be used to write polymorphic functions, that is, functions where the algorithm is specified so that it will apply to a set of different argument types. For example:

```
void sort(common* v[], int size)
{
    /* sort the vector of commons "v[size]" */
}
```

The `sort` function need only be able to compare objects of class `common` to perform its task. So, if class `common` has a virtual function `cmpr()`, `sort()` will be able to sort vectors of objects of any class derived from class `common` for which `cmpr()` is defined. For example:

```
class common {
    ...
    virtual int cmpr(common*);
};

class apple : public common {
    ...
    int key;
    int cmpr(common* arg)
    {   /* assume that arg is also an apple */
```

```
        int k = ((apple*)arg)->key;
        return (key==k) ? 0 : (key<k) ? -1 : 1;
    }
};

class orange : public common {
    ...
    int cmpr(common*);
};
```

The `cmpr()` function was preferred to the superficially more attractive approach of overloading the "<" operator because my favorite sort algorithm uses a three-way compare. To write a `sort()` to operate on a vector of class `common`, rather than on a vector of pointers to class `common`, a virtual "size" function would be needed.

Should it be desirable to compare an `apple` with an `orange`, some way for the comparison function to find its sort key would be needed. Class `common` could, for example, contain a virtual sort-key extraction function.

XVII. POLYMORPHIC CLASSES

Polymorphic classes can be constructed in the same way as polymorphic functions. For example:

```
class set : public common {
    class set_mem {
        set_mem* next;
        object* mem;
        set_mem(common* m, set_mem* n)
                { mem=m; next=n; }
    } *tail;
public:
    int insert(common*);
    int remove(common*);
    int member(common*);
        set()
        { tail = 0; }
        ~set()
         { if (tail) error("non-empty set"); }
};
```

That is, a set is implemented as a linked list of `set_mem` objects, each of which points to a class `common`. Pointers to objects (not objects) are inserted. For completeness a set is itself a `common` so that you can create a set of sets. Since class set is implemented without relying on data in the member objects, an object can be a member of two or more

sets. This model is quite general and can be (and indeed has been) used to create "abstractions" like `set`, `vector`, `linked_list`, and `table`. The most distinctive feature of this model for "container classes" is that in general the container cannot rely on data stored in the contained objects nor can the contained objects rely on data identifying their container (or containers). This is often an important structural advantage; classes can be designed and used without concerns about what kind of data structures the programs using them may need. Its most obvious disadvantage is that there is a minimum overhead of one pointer per member (two pointers in the linked list implementation of class `set` above).† Another advantage is that such container classes are capable of holding heterogeneous collections of members. Where this is undesirable, it is trivial to derive a class that will accept only members of one particular class. For example:

```
class apple_set : public set {
public:
    int insert(apple* a)     { return set::insert(a); }
    int remove(apple* a)     { return set::remove(a); }
    int member(apple* a)     { return set::member(a); }
};
```

Note that since the functions of class `apple_set` do not perform any actions in addition to those performed by the base class `set`, they will be optimized away. They serve only to provide compile time type checking.

A class `common` with a "matching" set of polymorphic classes and functions is being designed. The intention is to provide it as a standard library.

XVIII. INPUT AND OUTPUT

C does not have special facilities for handling input and output. Traditionally the programmer relies on library functions like `printf()` and `scanf()`. For example, to print a data structure representing a complex number one might write:

```
printf("(%g,%g)\n", zz.real, zz.imag);
```

Unfortunately, since the old C standard input/output functions know only the standard types, it is necessary to print a structure member by member. This is often tedious and can only be done where the members are accessible. The paradigm cannot be cleanly and generally extended to handle user-defined types and input/output formats.

---

† Plus another pointer for the implementation of the virtual function mechanism. See Section XXI, "Efficiency", below.

The approach taken in C++ is to provide (in a "standard" library, not in the language itself) the operator ≪ ("put to") for a data type `ostream` and each basic and user-defined type. Given an output stream `cout`, one can write

```
cout<<zz;
```

The implementor of class complex defines ≪ for a complex number. For example:

```
ostream& operator<<(ostream& s, complex& c)
{
    return s<<"("<<c.real <<","<<c.imag<<")\n";
}
```

The ≪ operator was chosen in preference to a function name to avoid the tedium of having to write a separate call for each argument. For example:

```
put(cout,"(");     /* intolerably verbose */
put(cout,c.real);
put(cout,",");
put(cout,c.imag);
put(cout,"\n");
```

There is a loss of control over the formatting of output when using ≪ compared with using `printf`. Where such finer control is necessary, one can use "formatting functions". For example:

```
cout<<"hex = "<<hex(x)<<"octal x = "<<oct(x);
```

where `hex()` and `oct()` return a string representation of their first argument.

Input is handled by providing the operator ≫ ("get from") for a data type `istream` and each basic and user-defined type. If an input operation fails, the stream is put into an error state that will cause subsequent operations on it to fail. For a variable `zz` of any type one can write code like this

```
while ( cin>>zz ) cout<<zz;
```

Surprisingly enough, the input operations are typically trivial to write, since there invariably is a constructor to do the nontrivial part of the job, and the arguments to the constructor(s) give a good first approximation of the input format. For example:

```
istream& operator>> (istream& s, complex& zz)
{
    if (!s) return s;
    double re = 0, im = 0;
```

```
    char c1 = 0, c2 = 0, c3 = 0;
    s>>c1>>re>>c2>>im>>c3;
    if (c1!='(' || c2!=',' || c3!=')') s.state = _bad;
    if (s) zz = complex(re,im);
    return s;
}
```

The convention for functions implementing the input and output operators is to return the argument stream and indicate success or failure in its state. This example is a bit too simple for real use, but it will change the value of its argument `zz` and return the stream in a nonerror state if and only if a complex number of the form `(double,double)` was found. The interpretation of a test on a stream as a test on its state is handled by overloading the `!=` operator for an `istream`. For example, the test `if (s)` above is interpreted as `if (s!=0)`, which in turn is interpreted as a call to `istream::operator!=()`, which finally examines `s.state`.

Note that there is no loss of type information when using ≪ and ≫, so, compared with the `printf/scanf` paradigm, a large class of errors has been eliminated. Furthermore, ≪ and ≫ can be defined for a new (user-defined) type without affecting the "standard" classes `istream` and `ostream` in any way, and without any knowledge of the internals of these classes. An `ostream` can be bound to a real output device (buffered or unbuffered) or simply to an in-core buffer, as can an `istream`. This extends the range of uses considerably and eliminates the need for the old C functions `sscanf` and `sprintf`.

Character-level operations `put()` and `get()` are also available for I/O streams.

XIX. FRIENDS VS. MEMBERS

When a new operation is to be added to a class, there are typically two ways it can be implemented, as a friend or as a member. Why are two alternatives provided, and for what kind of operations should each alternative be preferred?

A friend function is a perfectly ordinary function, distinguished only by its permission to use private member names. Programming using friends is essentially programming as if there were no data hiding. The friend approach cleanly implements the traditional mathematical view of values that can be used in computation, assigned to variables, but never really modified. This paradigm is then compromised by using pointer arguments.

A member function, on the other hand, is tied to a single class and invoked for one particular object. The member approach cleanly implements the idea of operations that change the state of an object,

for example, assignment. Because a single object is distinguished, the language can take advantage of local knowledge to provide notational convenience and efficient implementation, and to let the meaning of the operation depend on the value of that object. Note that it is not possible to have a virtual friend. Constructors, too, must be members.

As the first approximation, use a member to implement an operation if it might conceivably modify the state of an object. Note that type conversion, if declared, is performed on arguments, but not on the object for which a member is invoked. Consequently, the member implementation should also be chosen for operations where type conversion is undesirable.

A friend function can be the friend of two or more classes, while a member function is a member of a single class. This makes it convenient to implement operations on two or more classes as friends. For example:

```
class matrix {
    friend matrix operator*(matrix, vector);
    ...
};

class vector {
    friend matrix operator*(matrix, vector);
    ...
};
```

It would take two members, `matrix.operator*()` and `vector.operator*()`, to achieve what the friend `operator*()` does.

The name of a friend is global, while the scope of a member name is restricted to its class. When structuring a large program, one tries to minimize the amount of global information; therefore, friends should be avoided in the same way that global data are. Ideally, at this level, all data are encapsulated in classes and operated on using member functions. However, at a more detailed level of programming this becomes tedious and often inefficient; here friends come into their own.

Finally, if there is no obvious reason for preferring one implementation of an operation over another, make that operation a member.

XX. SEPARATE COMPILATION

For separate compilation the traditional C approach has been retained. Type specifications are shared by textually including them in separately compiled source files. There is no automatic mechanism that ensures that the header files contain complete type specifications and that they are used consistently. Such checks must be specifically

requested and performed separately from the compilation process. The names of external variables and functions from the resulting object files are matched up by a loader that has no concept of data type. A loader that could check types would be of great help, and would not be difficult to provide.

A class declaration specifies a type so it can be included in several source files without any ill effects. It must be included in every file using the class. Typically, member functions do not reside in the same file as the class declaration. The language does not have any expectations of where member functions are stored. In particular, it is not required that all member functions for a class should be in one file, or that they should be separated from other declarations.

Since the private and the public parts of a class are not physically separated, the private part is not really "hidden" from a user of a class, as it would be in the ideal data abstraction facility. Worse, any change to the class declaration may necessitate recompilation of all files using it. Obviously, if the change was to the private part, only the files containing member functions or friends have to be recompiled. (The addition of a new member function will in most cases not create a need for any recompilation. The addition may, however, hide an extern function used in some other member function, thus changing the meaning of the program. Unfortunately, this rare event is quite hard to detect.) A facility that could determine the set of functions (or the set of source files) that needs to be recompiled after a change to a class declaration would be extremely useful. It is unfortunately nontrivial to provide one that does not slow down the compiler significantly.

XXI. EFFICIENCY

Run-time efficiency of the generated code was considered of primary importance in the design of the abstraction mechanisms. The general assumption was that if a program can be made to run faster by not using classes, many programmers will perfer speed. Similarly, if a program can be made to use less store by not using classes, many programmers will prefer compact representation. It is demonstrated below that classes can be used without any loss of run-time efficiency or data representation compactness compared to "old C" programs.

This insistence on efficiency led to the rejection of facilities requiring garbage collection. To compensate, the overloading facility was designed to allow complete encapsulation of storage management issues in a class. Furthermore, it has been made easy for a programmer to provide special-purpose free store managers. As described above, constructors and destructors can be used to handle allocation and deallocation of class objects. In addition, the functions `operator`

`new()` and `operator delete()` can be declared to redefine the meaning of the `new` and `delete` operators.

A class that does not use virtual functions uses exactly as much space as an C struct with the same data members. There is no hidden per object store overhead. There is no per class store overhead either. A member function does not differ from other functions in its store requirements. If a class uses virtual functions, there is an overhead of one pointer per object plus one pointer per virtual function.

When a (nonvirtual) member function is called, for example `ob.f(x)`, the address of the object is passed as a hidden argument: `f(&ob,x)`. Thus call of a member function is as least as efficient as a call of a nonmember function. The call of a virtual function `p->f(x)` is roughly equivalent to an indirect call `(*(p->virtual[5]))(p,x)`. Typically, this causes three memory references more than a call of an equivalent nonvirtual function.

If the function call overhead is unacceptable for an operation on a class object, the operation can be implemented as an in-line function, thus achieving the same run-time efficiency as if the object had been directly accessed.

XXII. IMPLEMENTATION AND COMPATIBILITY

The C++ compiler front end, `cfront`, consists of a YACC parser[8] and a C++ program. Classes are used extensively. It is about same size as the equivalent part of the PCC compiler for old C (13,500 lines including comments, etc.). It runs a bit faster, but uses more store. The amount of store used depends on the number of external variables and the size of the largest function. It will never run on machines with a 128K-byte address space (like a PDP-11/70[†]); three times that amount of store appears to be more reasonable. A completely type-checked internal representation is produced. This can then be transformed into suitable input for a range of new and old code generators. In particular, an "old C" version of any C++ program can be produced. This makes it trivial to transfer `cfront` to any system with an old C compiler.

With few exceptions the C++ compiler accepts old C. The run-time environment, the linkage conventions, and the method for specifying separate compilation remain unchanged. The major incompatibility is that a function declaration, for example,

```
int f();
```

in old C declares a function with an unknown number of arguments

[†] Trademark of Digital Equipment Corporation.

of unknown types. In C++, that declaration specifies that `f` takes no arguments. A C++ version of the declarations for the standard libraries exists, and a program producing the "missing declarations" for a set of source files is being written. Another difference is that in C++ a nonlocal name can only be used in the file in which it occurs, unless it is explicitly declared to be extern; in old C a nonlocal name is common to all files in a multifile program, unless it is explicitly declared to be static. Name clashes with the new key words `class`, `const`, `delete`, `friend`, `inline`, `new`, `operator`, `overload`, `public`, `this`, and `virtual` may cause minor irritations.

It is often claimed that one of C's major virtues is that it is so small that every programmer understands every construct in the language. In contrast, languages like PL/1 and Ada are presented as if every programmer writes in his own subset of the language and can understand programs written by others only with great difficulty. It follows from this view that extension of C is bad. This argument against "big languages" ignores the simple fact that the dependencies between data structures and the functions using them exist in a program independently of whether or not they have been recorded in a class declaration. Programs using classes tend to be marginally shorter than their unstructured counterparts (1 to 10 percent shorter is typical; 50 percent shorter has been seen; the author has yet to see a program that grew without functionality being added). Furthermore, C is already large enough for subcultures using subsets of the language to exist, and the macro facilities are often used to create arbitrarily incomprehensible variations of the language.

The `cfront` manual is only 14 percent longer than the "old C" manual, so the effort of learning the new language facilities should not be prohibitively large. In particular, it should be a small effort compared with learning a new language containing data abstraction features. However, when classes are used to create new data types, a new dialect of the language is in fact created. This will lead to different incompatible "dialects". This is not that much different from the current state of affairs, and hopefully "standard" classes providing basic facilities like input/output, sets, tables, strings, graphics, etc., will win wide acceptance.

XXIII. COMPARISON WITH OTHER LANGUAGES

To compare two languages takes a whole paper, if not a book. Consequently, this section can provide only a few personal opinions and pointers to the main areas of difference between the languages. For completeness C itself is criticized in the same way as the other languages.

The C class facility is modeled on the original Simula67 classes.[6,7]

Simula relies on garbage collection both for class objects and procedure activation records, and does not provide facilities for function name or operator overloading. It is, however, a most beautiful and expressive language, and C classes owe more to it than to any other language.

Smalltalk[9] is another language with the same kind of facilities for creating class hierarchies. There, however, all functions are virtual and all type checking is done at run time. This means that where a C base class provides a fixed type-checked interface to a set of derived classes, a Smalltalk superclass provides a minimal untyped set of facilities that can be arbitrarily modified. Smalltalk relies on garbage collection and on dynamic resolution of member function names. It does not provide operator overloading in the usual sense, but an operator may be the name of a member function. Smalltalk provides an extremely nice integrated environment for program construction. The resulting programs are very demanding of resources, however.

Modula-2[10] provides a rudimentary abstraction facility called a module. A *module* is not a type but a single object containing data and access functions. It is somewhat similar to a class with all data members static. There is no facility equivalent to derived classes. It does not allow overloading of function names or operators. No garbage collection is provided.

Mesa's[11] modules are distinguished by a clean and flexible separation of the interface of a module from its implementation. This enables and requires sophisticated facilities for separate compilation and linking. A module can import and export both procedure and type names. The rules for instantiation of modules (object creation and initialization) are so general as to make them inelegant. Some space and time overheads are incurred by using modules. There are no facilities for constructing module hierarchies and no facilities for operator overloading. Mesa relies on garbage collection both for data objects and procedure activation records. Consequently, it will run efficiently only where hardware support for garbage collection is available.

Ada's[12] data abstraction facility, the *package*, is essentially similar to the *class/friend* facility in C. There is no equivalent to member functions or constructors; this leads to verbosity. Nor is there an equivalent to derived classes, so the `shape` example above does not appear to have an elegant solution in Ada. Operators and function names can be overloaded, assignment cannot. Packages can be generic. That is, a package can be defined with types as arguments. The standard example is a stack of elements where the type of an element is an argument. The facility is far less flexible than C "polymorphic classes", but more space-efficient for simple abstractions. Ada does not provide garbage collection.

C provides no integrated environment for editing, debugging, control

of separate compilation, and source code control. The C programming environment under the *UNIX*™ system[1,8] provides a tool kit of such services, but it leaves much to be desired. No garbage collection is provided. C classes distinguish themselves by combining facilities for creating class hierarchies with efficient implementation. The facilities for object creation and initialization are notable. The facilities for overloading assignment and argument passing are unique to C.

XXIV. CONCLUSION

The addition of classes represents a quantum jump for the C language, the least extension that provides facilities for data abstraction for systems programming. The experience of three years with intermediate versions ("C with classes") demonstrated both the usefulness of classes and the need for the more general facilities presented here. The efficiency of both the compiled code and the compiler itself compares favorably with old C.

XXV. ACKNOWLEDGMENTS

The concepts presented here never would have matured without the constant help and constructive criticism from my colleagues and users, notably, Tom Cargill, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Ravi Sethi, and Jon Shopiro.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice Hall, 1978.
2. G. Orwell, *1984*, London: Secker and Warburg, 1949.
3. B. Stroustrup, *C++ Reference Manual*, Murray Hill, NJ: AT&T Bell Laboratories CSTR-108, January 1, 1984.
4. B. Stroustrup, "Classes: An Abstract Data Type Facility for the C Language," ACM SIGPLAN Notices, *17*, No. 1 (January 1982), pp. 42–52.
5. B. Stroustrup, "Adding Classes to C: An Exercise in Language Evolution," Software Practice and Experience, *13* (1983), pp. 139–61.
6. O-J. Dahl and C. A. R. Hoare, *Hierarchical Program Structures, Structured Programming*, New York: Academic Press, 1972, pp. 174–220.
7. O-J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Common Base Language*, Oslo, Norway: Norwegian Computing Center, S-22, 1970.
8. *Unix Programmer's Manual*, Murray Hill, NJ: AT&T Bell Laboratories, 1979.
9. A. Goldberg and D. Robson, *Smalltalk-80 The Language and Its Implementation*, Reading, MA: Addison Wesley, 1983.
10. N. Wirth, *Programming in Modula-2*, Berlin: Springer-Verlag, 1982.
11. J. G. Mitchell et al., *Mesa Reference Manual*, Palo Alto, CA: Xerox PARC CSL-79-3, 1979.

AUTHOR

Bjarne Stroustrup, Cand. Scient. (Mathematics and Computer Science), 1975, University of Aarhus Denmark; Ph.D. (Computer Science), 1979, Cambridge University; AT&T Bell Laboratories 1979—. Mr. Stroustrup's research interests include distributed systems, operating systems, simulation, programming methodology, and programming languages. He is currently a member of the Computer Science Research Center. Member, ACM and IEEE.

Multiparadigm Programming in Standard C++

Bjarne Stroustrup

AT&T Labs – Research

http://www.research.com/~bs

Abstract

- Multi-paradigm programming is programming applying different styles of programming, such as object-oriented programming and generic programming, where they are most appropriate. This talk presents simple example of individual styles in ISO Standard C++ and examples where these styles are used in combination to produce cleaner, more maintainable code than could have been done using a single style only.

- 60 minutes, incl Q&A

Overview

- Standard C++
  - C++ aims, standardization, overview
- Abstraction: Classes and templates
  - Range example
  - Resource management
- Generic Programming: Containers and algorithms
  - Vector and sort example
  - Function objects
- Object-Oriented Programming: class hierarchies and interfaces
  - Ye olde shape example
- Multi-paradigm Programming
  - Algorithms on shapes example
- Implications for Large Systems
  - Libraries, coupling, and lock-in

Standard C++

- ISO/IEC 14882 – Standard for the C++ Programming Language
  - Core language
  - Standard library
- Implementations
  - Borland, Compaq, IBM, EDG, GNU, Metrowerks, Microsoft, SGI, Sun, Etc.
    + many ports
  - All approximate the standard: portability is improving
  - Some are free
  - For all platforms: BeOS, Mac, IBM, Linux/Unix, Windows, Symbion, Palm, embedded systems, etc.
- Probably the world's most widely used general-purpose programming language

Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

- A multiparadigm programming language

  (if you must use long words)
  - The most effective styles use a combination of techniques

Elegant, direct expression of ideas

- Declarative information is key:

```
Matrix<double,100,50,Sparse> ms;
Matrix<Quad,100,50,Dense,Triangular<upper> > mt;
Matrix<double,100,50> m;     // defaults to rectangular and dense
```

- Write expressions using “natural” notation:

```
m3 = add(mul(m,v),v2);       // functional
m2 = m*v+v2;                 // algebraic
```

- Execute without spurious function calls or temporaries

Uncompromising performance

My aims for this presentation

- Here, I want to show small, elegant, examples
  - building blocks of programs
  - building blocks of programming styles
- Elsewhere, you can find
  - huge libraries
    - Foundation libraries: vendor libs, Threads++, ACE, QT, boost.org, …
    - Scientific libraries: POOMA, MTL, Blitz++, ROOT, …
    - Application-support libraries: Money++, C++SIM, BGL, …
    - Etc.: C++ Libraries FAQ: http://www.trumphurst.com
  - powerful tools and environments
  - in-depth tutorials
  - reference material

C++’s weakness: poor use

- C style
  - Arrays, pointers, casts, macros, complicated use of free store (heap)
- Reinventing the wheel
  - Strings, vectors, lists, maps, GUI, graphics, numerics, units, concurrency, graphs, persistence, …
- Smalltalk-style hierarchies
  - “brittle” base classes
  - Overuse of hierarchies

Here, I focus on alternatives

- Primarily relying on abstract classes, templates, and function objects

C++ Classes

- Primary tool for representing concepts
  - Represent concepts directly
  - Represent independent concepts independently
- Play a multitude of roles
  - Value types
  - Function types (function objects)
  - Constraints
  - Resource handles (e.g. containers)
  - Node types
  - Interfaces

Classes as value types

- Built-in types
  - bool, char, int, float, double, unsigned char, long int, pointers, arrays, …
- Standard-library types
  - string, vector, list, map, set, ostream, complex, priority_queue, auto_ptr, …
- User-defined types
  - Date, Socket, hash_map, Point, Range, Real, Buffer, Line_segment, Person, …

Classes as value types

```
void f(Range arg)           // Range: y in [x,z)
try
{
    Range v1(0,3,10);       // 3 in range [0,10)
    Range v2(7,9,100);      // 9 in range [7,100)
    v1 = v2;        // ok: 9 is in [0,10)
    v2 = v1;        // will throw exception: 3 is not in [7,100)
    v1 = v2-v1;     // ok: 9-3 is in [0,10)
    arg = v1;       // may throw exception
    v2 = arg;       // may throw exception
}
catch(Range_error) {
    cerr << “Oops: range error in f()”;
}
```

Classes as value types

```
class Range {                          // simple value type
    int value, low, high;              // invariant: low <= value < high
    void check(int v) { if (v<low || high<=v) throw Range_error(); }
public:
    Range(int lw, int v, int hi) : low(lw), value(v), high(hi) { check(v); }
    Range(const Range& a) :low(a.low), value(a.value), high(a.high) { }

    Range& operator=(const Range& a)
        { check(a.value); value=a.value; return *this; }
    Range& operator=(int a) { check(a); value=a; return *this;}

    operator int() const { return value; }     // extract value
};
```

Classes as value types: Generalize

```
template<class T> class Range {                 // simple value type
    T value, low, high;                         // invariant: low <= value < high
    void check(T v) { if (v<low || high<=v) throw Range_error(); }
public:
    Range(T lw,T v, T hi) : low(lw), value(v), high(hi) { check(v); }
    Range(const Range& a) :low(a.low), value(a.value), high(a.high) { }

    Range& operator=(const Range& a)
        { check(a.value); value=a.value; return *this; }
    Range& operator=(const T& a) { check(a); value=a; return *this; }

    operator T() const { return value; }        // extract value
};
```

Classes as value types

```
Range<int> ri(10, 10, 1000);
Range<double> rd(0, 3.14, 1000);
Range<char> rc('a', 'a', 'z');
Range<string> rs("Algorithm", "Function", "Zero");
Range< complex<double> > rc(0,z1,100);  // error: < is not defined for complex
```

Templates: Constraints

- How can we check template parameter constraints?
  - The compiler always checks
    - late and gives poor error messages
  - The programmer can specify a check
    - Checking arbitrary constraints
      - Not just subtype/subclass relationships
      - Correspondence between several types
      - Specific properties of types
    - Readable compile-time error messages
    - No spurious code generated when constraints are met

Templates: Constraints

```
Template<class T> struct Comparable {
    static void constraints(T a, T b) { a<b; a<=b; }  // the constraint check
    Comparable() { void (*p)(T,T) = constraints; }  // trigger the constraint check
};

Template<class T> struct Assignable { /* … */ };

template<class T> class Range
    : private Comparable<T>, private Assignable<T> {
    // …
};

Range<int> r1(1,5,10);                    // ok
Range< complex<double> > r2(1,5,10);      // constraint error: no < or <=
```

Managing Resources

- Examples of resources
  - Memory, file handle, thread handle, socket
- General structure (“resource acquisition is initialization”)
  - Acquire resources at initialization
  - Control access to resources
  - Release resources when destroyed
- Key to exception safety
  - No object is created without the resources needed to function
  - Resources implicitly released when an exception is thrown

Managing Resources

```
//      unsafe, naïve use:

void f(const char* p)
{
    FILE* f = fopen(p,"r");     // acquire
    // use f
    fclose(f);                  // release
 }
```

Managing Resources

```
//      naïve fix:

void f(const char* p)
{
    FILE* f = 0;
    try {
      f = fopen(p,”r”);
      // use f
    }
    catch (…) {          // handle every exception
      // …
    }
    if (f) fclose(f);
  }
```

Managing Resources

// use an object to represent a resource (“resource acquisition is initialization”)

```
class File_handle {      // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r) { p = fopen(pp,r); }
        File_handle(const string& s, const char* r) { p = fopen(s.c_str(),r); }
        ~File_handle() { if (p) fclose(p); }  // destructor
        // copy operations
        // access functions
};

void f(string s)
{
        File_handle f(s,”r”);
        // use f
}
```

Generic Programming

- First aim: Standard Containers
  - Type safe
    - without the need for run-time checking
  - Efficient
    - Without excuses
  - Interchangeable
    - Where reasonable
- Consequential aim: Standard Algorithms
  - Applicable to many/all containers
- General aim: The most general, most efficient, most flexible representation of concepts
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful

Read and sort example

```
int n;
while (cin>>n) vi.push_back(n);        // read integers
sort(vi.begin(), vi.end());            // sort integers

string s;
while (cin>>s) vs.push_back(s);        // read strings
sort(vs.begin(),vs.end());             // sort strings


template<class T> void read_and_sort(vector<T>& v)    // use < for comparison
{
    T t;
    while (cin>>t) v.push_back(t);
    sort(v.begin(),v.end());
}
```

Read and sort example

```
template<class T, class Cmp>
void read_and_sort(vector<T>& v, Cmp c = less<T>())
{
    T t;
    while (cin>>t) v.push_back(t);
    sort(v.begin(), v.end(), c);
}

vector<double> vd;
read_and_sort(vd);              // sort using the default <

vector<string> vs;
read_and_sort(vs,No_case());  // sort case insensitive
```

Generality/flexibility is affordable

- Read and sort floating-point numbers
  - C: read using stdio; **qsort(buf,n,sizeof(double),compare)**
  - C++: read using iostream; **sort(v.begin(),v.end());**

#elements	C++	C	C/C++ ratio
500,000	2.5	5.1	2.04
5,000,000	27.4	126.6	4.62

- How?
  - clean algorithm
  - inlining

(Details: May’99 issue of C/C++ Journal; http://www.research.att.com/~bs/papers.html)

Matrix optimization example

```
struct MV {        // object representing the need to multiply
    Matrix* m;
    Vector* v;
    MV(Matrix& mm, Vector& vv) : m(&mm), v(&vv) { }
};

MV operator*(const Matrix& m, const Vector& v)
    { return MV(m,v); }

MVV operator+(const MV& mv, const Vector& v)
    { return MVV(mv.m,mv.v,v); }

v = m*v2+v3;  // operator*(m,v2) -> MV(m,v2)
              // operator+(MV(m,v2),v3) -> MVV(m,v2,v3)
              // operator=(v,MVV(m,v2,v3)) -> mul_add_and_assign(v,m,v2,v3);
```

Function Objects

- Function objects
  - Essential for flexibility
  - Efficient
    - in practice, more so than inline functions
    - important: **sort()** vs. **qsort()**
  - Some find them tedious to write
    - Standard function objects
      - e.g., **less**, **plus**, **mem_fun**
    - Can be automatically written/generated
      - **Vector v2 = m*v+k;** // matrix and vector libraries
      - **find_if(b,e, 0<x && x<=max);** // lambda libraries

Object-oriented Programming

- Hide details of many variants of a concepts behind a common interface

```
void draw_all(vector<Shape*>& vs)
{
   typedef vector<Shape*>::iterator VI;
   for (VI p = vs.begin(); p!=vs.end(), ++p) p->draw();
}
```

- Provide implementations of these variants as derived classes

Class Hierarchies

- One way (often flawed):

```
class Shape {       // define interface and common state
    Color c;
    Point center;
    // …
public:
    virtual void draw();
    virtual void rotate(double);
    // …
};

class Circle : public Shape { double radius; /* … */ void rotate(double) { }  };
class Triangle : public Shape { Point a,b,c; / * … */ void rotate(double); };
```

Class Hierarchies

Class Hierarchies

- Fundamental advantage: you can manipulate derived classes through the interface provided by a base:

```
void f(Shape* p)
{
        p->rotate(90);
        p->draw();
}
```

- You can add new **Shape**s to a program without changing or recompiling code such as **f()**

Class Hierarchies

- Another way (usually better):

```
class Shape {       // abstract class: interface only
    // no representation
public:
    virtual void draw() = 0;
    virtual void rotate(double) = 0;
    virtual Point center() = 0;
    // …
};

class Circle : public Shape { Point center; double radius; Color c; /* … */ };
class Triangle : public Shape {  Point a, b, c; Color c; / * … */ };
```

Class Hierarchies

Class Hierarchies

- One way to handle common state:

```
class Shape {       // abstract class: interface only
public:
    virtual void draw() = 0;
    virtual void rotate(double) = 0;
    virtual Point center() = 0;
    // …
};

class Common { Color c; /* … */ };          // common state for Shapes

class Circle : public Shape, protected Common{ /* … */ };
class Triangle : public Shape, protected Common { / * … */ };
class Logo: public Shape { /* … */ };       // Common not needed
```

Class Hierarchies

Multiparadigm Programming

- The most effective programs often involve combinations of techniques from different “paradigms”

- The real aims of good design
  - Represent ideas directly
  - Represent independent ideas independently in code

Algorithms on containers of polymorphic objects

```
void draw_all(vector<Shape*>& v)                    // for vectors
{
    for_each(v.begin(), v.end(), mem_fun(&Shape::draw));
}

template<class C> void draw_all(C& c)               // for all standard containers
{
    Contains<Shape*,C>();              // constraints check
    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}

template<class For> void draw_all(For first, For last)        // for all sequences
{
    Points_to<Shape*,For>();           // constraints check
    for_each(first, last, mem_fun(&Shape::draw));
}
```

Implications for Larger Systems

- First build or buy extensive libraries
  - Without suitable libraries everything is difficult
  - With suitable libraries most things are easy
  - Where possible, build on the C++ standard library
- Focus design/implementation on
  - Abstract classes as interfaces
  - Templates for type safety and efficiency
  - Function objects for flexible parameterization
- Avoid large single-rooted hierarchies
  - More generally, avoid unnecessary dependencies/coupling

Summary

- Think of Standard C++ as a new language
  - not just C plus a bit
  - not just class hierarchies
- Experiment
  - Be adventurous: Many techniques that didn’t work years ago now do
  - Be careful: Not every technique works for everybody, everywhere
- Prefer the C++ standard library style to C style
  - vector, list, string, etc. rather than array, pointers, and casts
  - Small free-standing classes are essential for flexibility
  - General algorithms should be free-standing (not members) for flexibility
- Use abstract classes to define major interfaces
  - Don’t get caught with “brittle” base classes

More information

- Books
  - Stroustrup: The C++ Programming language (Special Edition)
    - New appendices: Standard-library Exception safety, Locales
  - Stroustrup: The Design and Evolution of C++
  - C++ In-Depth series
    - Koenig & Moo: Accelerated C++ (innovative C++ teaching approach)
    - Sutter: Exceptional C++ (exception handling techniques and examples)
  - Book reviews on ACCU site
- Papers
  - Stroustrup:
    - Learning Standard C++ as a New Language
    - Why C++ isn't just an Object-oriented Programming Language
  - Higley and Powell: Expression templates … (The C++ Report, May 2000)
- Links: http://www.research.att.com/~bs
  - FAQs libraries, the standard, free compilers, garbage collectors, papers, chapters, C++ sites, interviews
  - Open source C++ libraries: Boost.org, ACE, …

Speaking C++ as a Native

Bjarne Stroustrup

AT&T Labs

http://www.research.com/~bs

Overview

- Standard C++
- Classes and templates
  - Values
  - Constraints checking
  - Resource management
  - Wrapping
- Containers and Algorithms
  - Find, sort
  - Matrices
- Class Hierarchies
  - Abstract classes
  - Algorithms on polymorphic containers

Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

- A Multi-paradigm programming language
  (if you must use long words)
- The most effective styles use a combination of techniques

Standard C++

- ISO/IEC 14882 – the C++ Programming Language
  - Core language
  - Standard library
- Implementations
  - Borland, IBM, EDG, DEC, GNU, Metrowerks, Microsoft, SGI, Sun, …

    + many ports
  - All approximate the standard: portability is improving
  - Some are free
  - For all platforms:
    - BeOS, Mac, IBM, Linux/Unix, Windows, embedded systems, …

My aims for this presentation

- Here, I want to show small, elegant, examples
  - building blocks of programs
  - building blocks of programming styles
- Elsewhere, you can find
  - huge libraries
    - Foundation libraries: vendor libs, Threads++, ACE, QT, boost.org, …
    - Scientific libraries: POOMA, MTL, Blitz++, ROOT, …
    - Application-support libraries: Money++, C++SIM, BGL, …
    - Etc.: C++ Libraries FAQ: http://www.trumphurst.com
  - powerful tools and environments
  - in-depth tutorials
  - reference material

C++ Classes

- Primary tool for representing concepts
  - Represent concepts directly
  - Represent independent concepts independently
- Play a multitude of roles
  - Value types
  - Function types (function objects)
  - Constraints
  - Resource handles (e.g. containers)
  - Node types
  - Interfaces

Classes as value types

```
void f(Range& r, int n)
try {
    Range v1(0,3,10);
    Range v2(7,9,100);
    v1 = 7;         // ok: 7 is in [0,10)
    v2 = 3;         // will throw exception: 3 is not in [7,100)
    int i = v2;     // extract the value from v2
    r = 7;          // may throw exception
    v2 = n;         // may throw exception
}
catch(Range_error) {
    cerr << “Oops: range error in f()”;
}
```

Classes as value types

```
class Range {                    // simple value type
    int value, low, high;                // invariant: low <= value < high
    void check(int v) { if (v<low || high<=v) throw Range_error(); }
public:
    Range(int lw, int v, int hi) : low(lw), value(v), high(hi) { check(v); }
    Range(const Range& a) { low=a.low; value=a.value; high=a.high; }

    Range& operator=(const Range& a) { check(a.value); value=a.value; }
    Range& operator=(int a) { check(a); value=a; }

    operator int() const { return value; }     // extract value
};
```

Classes as value types: Generalize

```
template<class T> class Range {   // simple value type
    T value, low, high;           // invariant: low <= value < high
    void check(const T& v) { if (v<low || high<=v) throw Range_error(); }
public:
    Range(const T& lw, const T& v, const T& hi)
        : low(lw), value(v), high(hi) { check(v); }
    Range(const Range& a) { low=a.low; value=a.value; high=a.high; }

    Range& operator=(const Range& a) { check(a.value); value=a.value; }
    Range& operator=(const T& a) { check(a); value=a; }

    operator T() const { return value; }      // extract value
};
```

Classes as value types

```
Range<int> ri(10, 10, 1000);
Range<double> rd(0, 3.14, 1000);
Range<char> rc(‘a’, ’a’, ’z’);
Range<string> rs(“Algorithm”, “Function”, “Zero”);
```

Templates: Constraints

```
Template<class T> struct Comparable {
    static void constraints(T a, T b) { a<b; a<=b; }  // the constraint check
    Comparable() { void (*p)(T,T) = constraints; }  // trigger the constraint check
};

Template<class T> struct Assignable { /* … */ };

template<class T> class Range
    : private Comparable<T>, private Assignable<T> {
    // …
};

Range<int> r1(1,5,10);                    // ok
Range< complex<double> > r2(1,5,10);      // constraint error: no < or <=
```

Templates: Constraints

- How can we check template parameter constraints?
  - The compiler always checks
    - late and gives poor error messages
  - The programmer can specify a check
    - Checking arbitrary constraints
      - Not just subtype/subclass relationships
      - Correspondence between several types
      - Specific properties of types
    - Readable compile-time error messages
    - No spurious code generated when constraints are met

Managing Resources

- Examples of resources
  - Memory, file handle, thread handle, socket
- General structure (“resource acquisition is initialization”)
  - Acquire resources at initialization
  - Control access to resources
  - Release resources when destroyed
- Key to exception safety
  - No object is created without the resources needed to function
  - Resources implicitly released when an exception is thrown

Managing Resources

```
//      unsafe, naïve use:

void f(const char* p)
{
    FILE* f = fopen(p,”r”);
    // use f
    fclose(f);
 }
```

Managing Resources

```
//      unsafe, naïve use:

void f(const char* p)
{
    FILE* f = fopen(p,"r");     // acquire
    // use f
    fclose(f);                  // release
 }
```

Managing Resources

```
//      naïve fix:

void f(const char* p)
{
    FILE* f = 0;
    try {
      f = fopen(p,”r”);
      // use f
    }
    catch (…) {          // handle exception
      // …
    }
    if (f) fclose(f);
 }
```

Managing Resources

// use an object to represent a resource (“resource acquisition in initialization”)

```
class File_handle {      // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r) { p = fopen(pp,r); }
        File_handle(const string& s, const char* r) { p = fopen(s.c_str(),r); }
        ~File_handle() { if (p) fclose(p); }  // destructor
        // access functions
};

void f(string s)
{
        File_handle f(s,”r”);
        // use f
}
```

Wrapping
(simple control abstraction)

- 20+ year old problem, guard/wrap operations
  - A prefix/suffix could be lock/unlock, transaction_start/transaction_commit, trace_on/trace_off, acquire_resource/release_resource
  - **Every** major application uses some form of guard/wrap
  - Simple example of use:

```
void f(X& x)
{
        Wrap<X> xx(x,prefix,suffix);
        int n = xx->count();        // prefix(); n=xx.count(); suffix();
        xx->g(99);                  // prefix(); xx.g(99); suffix();
}
```

- Optimal performance – inline prefix and suffix
- General: works for any “class X” – even pre-existing ones
- Wrap is 16 lines of standard C++

Wrapper implementation

```
template<class T, class Suf> class Wrap_proxy {
    T* p;
     Suf suffix;
public:
    Wrap_proxy(T* pp, Suf s) :p(pp), suffix(s) { }
    ~Wrap_proxy() { suffix(); }
    T* operator->() { return p; }
};

template<class T, class Pre, class Suf> class Wrap {
    T* p;
    Pre prefix;
    Suf suffix;
public:
    Wrap(T& x, Pre pref, Suf s) :p(&x), prefix(pref), suffix(s) { }
    Wrap_proxy<T,Suf> operator->()
            { prefix(); return Wrap_proxy<T,Suf>(p,suffix); }
};
```

Algorithms: Genericity

- Lots of useful containers
  - vector, list, map, …
- What do you do with containers?
  - Use algorithms (Knuth, Sedgewick, …)
    - find elements, sort container, add elements, remove elements, copy container, …
    - In any container: We don’t want to re-do each of the approximately 60 algorithms for each of the approximately 12 containers

Algorithms and Containers: Iterators and sequences

Conventional C notation

++	make iterator point to next element
*	dereference iterator

// Pseudo code (we want to make it real code):

copy(begin,end,output) // copy sequence to output

find(begin,end,value) // find value in sequence

count(begin,end,value) // count number of occurrences of value in sequence

Algorithms and containers

- Keeps independent concerns independent
  - Kind of “container” (sequence abstraction)
    - containers are not required to be part of a hierarchy
  - Element type
    - elements are not required to be part of a hierarchy
      - containers are nonintrusive
  - Algorithm
    - **not** member of class
  - Comparison criteria

Algorithms: find()

```
template<class In, class T>
In find(In first, In last, T val)          // find val in sequence [first,last)
{
    while (first!=last && *first!=val) // while we haven't reach the end and haven't found val
        ++first;                           // carry on
    return first;
}

void f(vector<int>& v, int x, list<string>& lst, string s)
{
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end()) { /* we found x */ }

    list<string>::iterator q = find(lst.begin(), lst.end(), s);
    if (q != lst.end()) { /* we found s */ }
}
```

Algorithms: find_if()

```
template<class In, class Pred>
In find_if(In b, In e, Pred p)
{
    while(b!=e && !p(*b))   // while we haven't reached the end
                            // and while we haven't found what we are looking for
        ++b;                // carry on
    return b;
}

void f(vector<string>& v, list<record>& lst, const Record& my_rec)
{
    vector<string>::iterator p =find_if(v.begin(), v.end(), Less_than<string>("foo"));
    if (p != v.end()) { /* found: *p < "foo" */ }

    list<Record>::iterator q = find_if(lst.begin(), lst.end(), Name_eq(my_rec));
    if (q != lst.end()) { /* found: *q has the same key as my_rec */ }
}
```

Function Objects

```
class Name_eq {
        const string s;
public:
        Name_eq(const Record& r) :s(r.name) { }
        static bool operator()(const Record& r) { return r.n == s; }
};

void f(vector<string>& v, list<record>& lst, const Record& my_rec)
{
        // …
                find_if(lst.begin(), lst.end(), Name_eq(my_rec));
        // …
}
```

Function Objects

```
template<class S> class F {          // simple, general example of function object
    S s;   // state
public:
    F(const S& ss) :s(ss) { /* establish initial state */ }
    void operator()(const S& ss) { /* do something with ss to s */ }
    operator S() { return s; }       // reveal state
};
```

Note, function objects:

- are more general than functions
- inline better than functions
- Can be generated from “natural” notation (e.g., x=y*z)

Algorithms: avoid temporaries

```
Matrix m;
Vector v, v2, v3;
// …
v = m*v2+v3;      // kindly evaluate without using temporaries
```

We need to generate: mul_add_and_assign(v,m,v2,v3);

Algorithms: Avoid temporaries

```
struct MV {        // object representing the need to multiply
    Matrix* m; Vector* v;
    MV(Matrix& mm, Vector& vv) : m(&mm), v(&vv) { }
};

MV operator*(const Matrix& m, const Vector& v)
    { return MV(m,v); }

MVV operator+(const MV& mv, const Vector& v)
    { return MVV(mv.m,mv.v,v); }

v = m*v2+v3;   // mul_add_and_assign(MVV(MV(m,v2),v3),v);
```

Algorithms: Delayed evaluation

- General technique:
  - collect information until you have everything you need
    - e.g. value, format, and stream
    - e.g. matrix, vector, …
  - optimize, vectorize, etc. given full information
  - relies on functions objects, inlining, pass by value
  - function objects often end up being templates
    - E.g. Matrix<double,Dense>

Class Hierarchies

- One way (often flawed):

```
class Shape {       // define interface and common state
    Color c;
    Point center;
    // …
public:
    virtual void draw();
    virtual void rotate(double);
    // …
};

class Circle : public Shape { double radius; /* … */ };
class Triangle : public Shape { Point a, b, c; / * … */ };
```

Class Hierarchies

Class Hierarchies

- Another way (usually better):

```
class Shape {       // abstract class: interface only
    // no representation
public:
    virtual void draw() = 0;
    virtual void rotate(double) = 0;
    virtual Point center() = 0;
    // …
};

class Circle : public Shape { Point c; double r; Color c; /* … */ };
class Triangle : public Shape {  Point a, b, c; Color c; / * … */ };
```

Class Hierarchies

Class Hierarchies

- One way to handle common state:

```
class Shape {       // abstract class: interface only
public:
    virtual void draw() = 0;
    virtual void rotate(double) = 0;
    virtual Point center() = 0;
    // …
};

class Common { Color c; /* … */ };            // common state for Shapes

class Circle : public Shape, protected Common{ Point c; double r; /* … */ };
class Triangle : public Shape, protected Common { point a, b, c; / * … */ };
class Odd_shape : public Shape { /* … */ };
```

Class Hierarchies

Algorithms on containers of polymorphic objects

```
void draw_all(vector<Shape*>& v)                    // for vectors
{
    for_each(v.begin(), v.end(), mem_fun(&Shape::draw));
}

template<class C> void draw_all(C& c)               // for all standard containers
{
    Contains<Shape*,C>();           // constraints check
    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}

template<class For> void draw_all(For first, For last)    // for all sequences
{
    Points_to<Shape*,For>();        // constraints check
    for_each(first, last, mem_fun(&Shape::draw));
}
```

Summary

- Think of Standard C++ as a new language
  - not just C plus a bit
  - not just class hierarchies
- Experiment
  - Be adventurous: Many techniques that didn’t work years ago now do
  - Be careful: Not every technique works for everybody, everywhere
- Prefer the C++ standard-library style to C style
  - vector, list, string, etc. rather than array, pointers, and casts
  - Small free-standing classes are essential for flexibility
  - General algorithms should be free-standing functions for flexibility
- Use abstract classes to define major interfaces
  - Don’t get caught with “brittle” base classes

More information

- Books
  - Stroustrup: The C++ Programming language (Special Edition)
  - Stroustrup: The Design and Evolution of C++
  - C++ In-Depth series
    - Koenig & Moo: Accelerated C++ (innovative C++ teaching approach)
    - Sutter: Exceptional C++ (exception handling techniques and examples)
  - Book reviews on ACCU site
- Papers
  - Stroustrup: Learning Standard C++ as a New Language
  - Stroustrup: Why C++ isn't just an Object-oriented Programming language
- Links: http://www.research.att.com/~bs
  - FAQs libraries, the standard, free compilers, garbage collectors, papers, chapters, C++ sites, interviews

What is ‘‘Object-Oriented Programming’’? (1991 revised version)

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

‘‘Object-Oriented Programming’’ and ‘‘Data Abstraction’’ have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula-2, Simula, and Smalltalk. The general idea is to equate ‘‘support for data abstraction’’ with the ability to define and use new types and equate ‘‘support for object-oriented programming’’ with the ability to express type hierarchies. Features necessary to support these programming styles in a general purpose programming language will be discussed. The presentation centers around C++ but is not limited to facilities provided by that language.

1 Introduction

Not all programming languages can be ‘‘object oriented’’. Yet claims have been made to the effect that APL, Ada, Clu, C++, CLOS, and Smalltalk are object-oriented programming languages. I have heard discussions of object-oriented design in C, Pascal, Modula-2, and CHILL. As predicted in the original version of this paper, proponents of object-oriented Fortran and Cobol programming are now appearing. ‘‘Object-oriented’’ has in many circles become a high-tech synonym for ‘‘good’’, and when you examine discussions in the trade press, you can find arguments that appear to boil down to syllogisms like:

```
Ada is good
Object oriented is good
-----------------------------------
Ada is object oriented
```

We simply *must* be more careful with our concepts and logic.

This paper presents one view of what ‘‘object oriented’’ ought to mean in the context of a general purpose programming language.

- §2 Distinguishes ‘‘object-oriented programming’’ and ‘‘data abstraction’’ from each other and from other styles of programming and presents the mechanisms that are essential for supporting the various styles of programming.
- §3 Presents features needed to make data abstraction effective.
- §4 Discusses facilities needed to support object-oriented programming.
- §5 Presents some limits imposed on data abstraction and object-oriented programming by traditional hardware architectures and operating systems.

The first version of this paper was presented at the Association of Simula Users’ meeting in Stockholm, August 1986. Later, a version was presented as an invited talk at the first European Conference on Object-Oriented Programming in Paris and published by Springer Verlag. It also appeared in the May 1988 issue of IEEE Software Magazine. This version has been revised to reflect the latest version of C++ as described in *The Annotated C++ Reference Manual*[5] approved by the ANSI C++ committee (X3J16) as the basis of formal standardization. Copyright (c) AT&T.

Examples will be presented in C++. The reason for this is partly to introduce C++ and partly because C++ is one of the few languages that supports both data abstraction and object-oriented programming in addition to traditional programming techniques. Issues of concurrency and of hardware support for specific higher-level language constructs are ignored in this paper.

2 Programming Paradigms

Object-oriented programming is a technique for programming – a paradigm for writing ‘‘good’’ programs for a set of problems. If the term ‘‘object-oriented programming language’’ means anything it must mean a programming language that provides mechanisms that support the object-oriented style of programming well.

There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that makes it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran, write type-secure programs in C, and use data abstraction in Modula-2, but it is unnecessarily hard to do because these languages do not support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and/or run-time checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and run-time checks can be used to extend linguistic support for paradigms. Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for paradigms.

A language is not necessarily better than another because it possesses a feature the other does not. There are many example to the contrary. The important issue is not so much what features a language possesses but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

[1] All features must be cleanly and elegantly integrated into the language.
[2] It must be possible to use features in combination to achieve solutions that would otherwise have required extra separate features.
[3] There should be as few spurious and ‘‘special purpose’’ features as possible.
[4] A feature should be such that its implementation does not impose significant overheads on programs that do not require it.
[5] A user need only know about the subset of the language explicitly used to write a program.

The last two principles can be summarized as ‘‘what you don’t know won’t hurt you.’’ If there are any doubts about the usefulness of a feature it is better left out. It is *much* easier to add a feature to a language than to remove or modify one that has found its way into the compilers or the literature.

I will now present some programming styles and the key language mechanisms necessary for supporting them. The presentation of language features is not intended to be exhaustive.

Procedural Programming

The original (and probably still the most commonly used) programming paradigm is:

Decide which procedures you want;
use the best algorithms you can find.

The focus is on the design of the processing, the algorithm needed to perform the desired computation. Languages support this paradigm by facilities for passing arguments to functions and returning values from functions. The literature related to this way of thinking is filled with discussion of ways of passing arguments, ways of distinguishing different kinds of arguments, different kinds of functions (procedures, routines, macros, ...), etc. Fortran is the original procedural language; Algol60, Algol68, C, and Pascal are later inventions in the same tradition.

A typical example of ‘‘good style’’ is a square root function. Given an argument, it produces a result.

To do this, it performs a well understood mathematical computation:

```
double sqrt(double arg)
{
    // the code for calculating a square root
}

void some_function()
{
    double root2 = sqrt(2);
    // ...
}
```

From a program organization point of view, functions are used to create order in a maze of algorithms.

Data Hiding

Over the years, the emphasis in the design of programs has shifted away from the design of procedures towards the organization of data. Among other things, this reflects an increase in program size. A set of related procedures with the data they manipulate is often called a *module*. The programming paradigm becomes:

Decide which modules you want;
partition the program so that data is hidden in modules.

This paradigm is also known as the ‘‘data hiding principle’’. Where there is no grouping of procedures with related data the procedural programming style suffices. In particular, the techniques for designing ‘‘good procedures’’ are now applied for each procedure in a module. The most common example is a definition of a stack module. The main problems that have to be solved are:

[1] Provide a user interface for the stack (for example, functions `push()` and `pop()`).
[2] Ensure that the representation of the stack (for example, a vector of elements) can only be accessed through this user interface.
[3] Ensure that the stack is initialized before its first use.

Here is a plausible external interface for a stack module:

```
// declaration of the interface of module stack of characters
char pop();
void push(char);
const stack_size = 100;
```

Assuming that this interface is found in a file called `stack.h`, the ‘‘internals’’ can be defined like this:

```
#include "stack.h"
static char v[stack_size];    // ``static'' means local to this file/module
static char* p = v;           // the stack is initially empty

char pop()
{
    // check for underflow and pop
}

void push(char c)
{
    // check for overflow and push
}
```

It would be quite feasible to change the representation of this stack to a linked list. A user does not have access to the representation anyway (since `v` and `p` were declared `static`, that is, local to the file/module in which they were declared). Such a stack can be used like this:

```
#include "stack.h"

void some_function()
{
    push('c');
    char c = pop();
    if (c != 'c') error("impossible");
}
```

Pascal (as originally defined) doesn't provide any satisfactory facilities for such grouping: the only mechanism for hiding a name from ''the rest of the program'' is to make it local to a procedure. This leads to strange procedure nestings and over-reliance on global data.

C fares somewhat better. As shown in the example above, you can define a ''module'' by grouping related function and data definitions together in a single source file. The programmer can then control which names are seen by the rest of the program (a name can be seen by the rest of the program *unless* it has been declared `static`). Consequently, in C you can achieve a degree of modularity. However, there is no generally accepted paradigm for using this facility and the technique of relying on `static` declarations is rather low level.

One of Pascal's successors, Modula-2, goes a bit further. It formalizes the concept of a module, making it a fundamental language construct with well defined module declarations, explicit control of the scopes of names (import/export), a module initialization mechanism, and a set of generally known and accepted styles of usage.

The differences between C and Modula-2 in this area can be summarized by saying that C only *enables* the decomposition of a program into modules, while Modula-2 *supports* that technique.

Data Abstraction

Programming with modules leads to the centralization of all data of a type under the control of a type manager module. If one wanted two stacks, one would define a stack manager module with an interface like this:

```
class stack_id;  // stack_id is a type
                 // no details about stacks or stack_ids are known here

stack_id create_stack(int size); // make a stack and return its identifier
destroy_stack(stack_id);         // call when stack is no longer needed

void push(stack_id, char);
char pop(stack_id);
```

This is certainly a great improvement over the traditional unstructured mess, but ''types'' implemented this way are clearly very different from the built-in types in a language. Each type manager module must define a separate mechanism for creating ''variables'' of its type, there is no established norm for assigning object identifiers, a ''variable'' of such a type has no name known to the compiler or programming environment, nor do such ''variables'' obey the usual scope rules or argument passing rules.

A type created through a module mechanism is in most important aspects different from a built-in type and enjoys support inferior to the support provided for built-in types. For example:

```
void f()
{
    stack_id s1;
    stack_id s2;

    s1 = create_stack(200);
    // Oops: forgot to create s2

    push(s1,'a');
    char c1 = pop(s1);
    if (c1 != 'a') error("impossible");
```

```
    push(s2,'b');
    char c2 = pop(s2);
    if (c2 != 'b') error("impossible");

    destroy_stack(s2);
    // Oops: forgot to destroy s1
}
```

In other words, the module concept that supports the data hiding paradigm enables this style of programming, but does not support it.

Languages such as Ada, Clu, and C++ attack this problem by allowing a user to define types that behave in (nearly) the same way as built-in types. Such a type is often called an *abstract data type*†. I prefer the term ''user-defined type.'' A way of defining types that are somewhat more abstract is demonstrated in the ''Multiple Implementations'' subsection of §3. The programming paradigm becomes:

Decide which types you want;
provide a full set of operations for each type.

Where there is no need for more that one object of a type the data hiding programming style using modules suffices. Arithmetic types such as rational and complex numbers are common examples of user-defined types:

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    complex(double r) { re=r; im=0; }   // float->complex conversion

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);   // binary minus
    friend complex operator-(complex);            // unary minus
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...
};
```

The declaration of class (that is, user-defined type) `complex` specifies the representation of a complex number and the set of operations on a complex number. The representation is *private*; that is, `re` and `im` are accessible only to the functions specified in the declaration of class `complex`. Such functions can be defined like this:

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

and used like this:

```
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2.3);
// ...
c = -(a/b)+2;
```

Most, but not all, modules are better expressed as user defined types. For concepts where the ''module representation'' is desirable even when a proper facility for defining types is available, the programmer can

† ''*Those types are not "abstract"; they are as real as* `int` *and* `float`.'' – Doug McIlroy. An alternative definition of *abstract data types* would require a mathematical ''abstract'' specification of all types (both built-in and user-defined). What is referred to as types in this paper would, given such a specification, be concrete specifications of such truly abstract entities.

declare a type and only a single object of that type. Alternatively, a language might provide a module concept in addition to and distinct from the class concept.

Problems with Data Abstraction

An abstract data type defines a sort of black box. Once it has been defined, it does not really interact with the rest of the program. There is no way of adapting it to new uses except by modifying its definition. This can lead to severe inflexibility. Consider defining a type `shape` for use in a graphics system. Assume for the moment that the system has to support circles, triangles, and squares. Assume also that you have some classes:

```
class point{ /* ... */ };
class color{ /* ... */ };
```

You might define a shape like this:

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where()       { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};
```

The ‘‘type field’’ `k` is necessary to allow operations such as `draw()` and `rotate()` to determine what kind of shape they are dealing with (in a Pascal-like language, one might use a variant record with tag `k`). The function `draw()` might be defined like this:

```
void shape::draw()
{
    switch (k) {
    case circle:
        // draw a circle
        break;
    case triangle:
        // draw a triangle
        break;
    case square:
        // draw a square
    }
}
```

This is a mess. Functions such as `draw()` must ‘‘know about’’ all the kinds of shapes there are. Therefore the code for any such function grows each time a new shape is added to the system. If you define a new shape, every operation on a shape must be examined and (possibly) modified. You are not able to add a new shape to a system unless you have access to the source code for every operation. Since adding a new shape involves ‘‘touching’’ the code of every important operation on shapes, it requires great skill and potentially introduces bugs into the code handling other (older) shapes. The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed sized framework presented by the definition of the general type `shape`.

Object-Oriented Programming

The problem is that there is no distinction between the general properties of any shape (a shape has a color, it can be drawn, etc.) and the properties of a specific shape (a circle is a shape that has a radius, is drawn by a circle-drawing function, etc.). Expressing this distinction and taking advantage of it defines object-oriented programming. A language with constructs that allows this distinction to be expressed and used supports object-oriented programming. Other languages don't.

The Simula inheritance mechanism provides a solution. First, specify a class that defines the general properties of all shapes:

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

The functions for which the calling interface can be defined, but where the implementation cannot be defined except for a specific shape, have been marked ''virtual'' (the Simula and C++ term for ''may be re-defined later in a class derived from this one''). Given this definition, we can write general functions manipulating shapes:

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

To define a particular shape, we must say that it is a shape and specify its particular properties (including the virtual functions).

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {}    // yes, the null function
};
```

In C++, class `circle` is said to be *derived* from class `shape`, and class `shape` is said to be a *base* of class `circle`. An alternative terminology calls `circle` and `shape` subclass and superclass, respectively.

The programming paradigm is:

> *Decide which classes you want;*
> *provide a full set of operations for each class;*
> *make commonality explicit by using inheritance.*

Where there is no such commonality data abstraction suffices. The amount of commonality between types that can be exploited by using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming to an application area. In some areas, such as interactive graphics, there is clearly enormous scope for object-oriented programming. For other areas, such as classical arithmetic types and computations based on them, there appears to be hardly any scope for more than data abstraction and the facilities needed for the support of object-oriented programming seem unnecessary†.

† However, more advanced mathematics may benefit from the use of inheritance: Fields are specializations of rings and vector spaces a special case of modules.

Finding commonality among types in a system is not a trivial process. The amount of commonality to be exploited is affected by the way the system is designed. When designing a system, commonality must be actively sought, both by designing classes specifically as building blocks for other types, and by examining classes to see if they exhibit similarities that can be exploited in a common base class.

For attempts to explain what object-oriented programming is without recourse to specific programming language constructs see Nygaard[14] and Kerr[10]. For a case study in object-oriented programming see Cargill[3].

Having examined the minimum support needed for procedural programming, data hiding, data abstraction, and object-oriented programming we will go into some detail describing features that – while not essential – can make data abstraction and object-oriented more effective.

3 Support for Data Abstraction

The basic support for programming with data abstraction consists of facilities for defining a set of operations (functions and operators) for a type and for restricting the access to objects of the type to that set of operations. Once that is done, however, the programmer soon finds that language refinements are needed for convenient definition and use of the new types. Operator overloading is a good example of this.

Initialization and Cleanup

When the representation of a type is hidden some mechanism must be provided for a user to initialize variables of that type. A simple solution is to require a user to call some function to initialize a variable before using it. For example:

```
class vector {
    int  sz;
    int* v;
public:
    void init(int size);   // call init to initialize sz and v
                           // before the first use of a vector
    // ...
};

vector v;
// don't use v here
v.init(10);
// use v here
```

This is error prone and inelegant. A better solution is to allow the designer of a type to provide a distinguished function to do the initialization. Given such a function, allocation and initialization of a variable becomes a single operation (often called instantiation or construction) instead of two separate operations. Such an initialization function is often called a constructor. In cases where construction of objects of a type is non-trivial, one often needs a complementary operation to clean up objects after their last use. In C++, such a cleanup function is called a destructor. Consider a vector type:

```
class vector {
    int  sz;                      // number of elements
    int* v;                       // pointer to integers
public:
    vector(int);                  // constructor
    ~vector();                    // destructor
    int& operator[](int index);   // subscript operator
};
```

The `vector` constructor can be defined to allocate space like this:

```
vector::vector(int s)
{
    if (s<=0) error("bad vector size");
    sz = s;
    v = new int[s];   // allocate an array of "s" integers
}
```

The `vector` destructor frees the storage used:

```
vector::~vector()
{
    delete v;         // deallocate the memory pointed to by v
}
```

C++ does not support garbage collection. This is compensated for, however, by enabling a type to maintain its own storage management without requiring intervention by a user. This is a common use for the constructor/destructor mechanism, but many uses of this mechanism are unrelated to storage management.

Assignment and Initialization

Controlling construction and destruction of objects is sufficient for many types, but not for all. It can also be necessary to control all copy operations. Consider class `vector`:

```
vector v1(100);
vector v2 = v1;  // make a new vector v2 initialized to v1
v1 = v2;         // assign v2 to v1
```

It must be possible to define the meaning of the initialization of `v2` and the assignment to `v1`. Alternatively it should be possible to prohibit such copy operations; preferably both alternatives should be available. For example:

```
class vector {
    int* v;
    int  sz;
public:
    // ...
    void operator=(const vector&);  // assignment
    vector(const vector&);          // initialization
};
```

specifies that user-defined operations should be used to interpret `vector` assignment and initialization. Assignment might be defined like this:

```
vector::operator=(const vector& a) // check size and copy elements
{
    if (sz != a.sz) error("bad vector size for =");
    for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

Since the assignment operation relies on the ‘‘old value’’ of the vector being assigned to, the initialization operation *must* be different. For example:

```
vector::vector(const vector& a)    // initialize a vector from another vector
{
    sz = a.sz;                // same size
    v = new int[sz];          // allocate element array
    for (int i = 0; i<sz; i++) v[i] = a.v[i];    // copy elements
}
```

In C++, a copy constructor, for example `X(const X&)` defines all initialization of objects of type `X` with another object of type `X`. In addition to explicit initialization copy constructors are used to handle arguments passed ‘‘by value’’ and function return values.

In C++ assignment of an object of class `X` can be prohibited by declaring assignment private:

```
class X {
    void operator=(const X&);   // only members of X can
    X(const X&);                // copy an X
    // ...
public:
    // ...
};
```

Ada does not support constructors, destructors, overloading of assignment, or user-defined control of argument passing and function return. This severely limits the class of types that can be defined and forces the programmer back to ‘‘data hiding techniques’’; that is, the user must design and use type manager modules rather than proper types.

Parameterized Types

Why would you want to define a vector of integers anyway? A user typically needs a vector of elements of some type unknown to the writer of the `vector` type. Consequently the vector type ought to be expressed in such a way that it takes the element type as an argument:

```
template<class T> class vector {     // vector of elements of type T
    T* v;
    int sz;
public:
    vector(int s)
    {
        if (s <= 0) error("bad vector size");
        v = new T[sz = s];    // allocate an array of "s" "T"s
    }
    T& operator[](int i);
    int size() { return sz; }
    // ...
};
```

A `template` specifies a family of types generated by specifying the the templats argument(s).

Vectors of specific types can now be defined and used:

```
vector<int> v1(100);      // v1 is a vector of 100 integers
vector<complex> v2(200);  // v2 is a vector of 200 complex numbers

v2[i] = complex(v1[x],v1[y]);
```

Ada, Clu, ML, and C++ support parameterized types†. There need not be any run-time overheads compared with a class where all types involved are specified directly.

A problem with parameterized types is that each instantiation creates an independent type. For example, the type `vector<char>` is unrelated to the type `vector<complex>`. Ideally one would like to be able to express and utilize the commonality of types generated from the same parameterized type. For example, both `vector<char>` and `vector<complex>` have a `size()` function that is independent of the parameter type. It is possible, but not trivial, to deduce this from the definition of class `vector` and then allow `size()` to be applied to any `vector`. An interpreted or dynamically compiled language (such as Smalltalk) or a language supporting both parameterized types and inheritance (such as C++) has an advantage here.

Exception Handling

As programs grow, and especially when libraries are used extensively, standards for handling errors (or more generally: ‘‘exceptional circumstances’’) become important. Ada, Algol68, Clu, and C++ each support a standard way of handling exceptions††.

---

† The ANSI C++ committee, X3J16, only accepted templates into C++ in July 1990 so only a few C++ implementations support templates at the time of writing.

†† The ANSI C++ committee, X3J16, only accepted exception handling into C++ in November 1990 so C++ implementations that sup-

Consider again the `vector` example:

```
class vector {
        // ...
        class range { }; // type to be used for exceptions
};

int& vector::operator[](int i)
{
    if (i<0 || sz<=i) throw range();
    return v[i];
}
```

Instead of calling an error function, `vector::operator[]()` can invoke the exception handling code, ‘‘throw the range exception.’’ This will cause the call stack to be unraveled until an exception handler for `vector::range` is found; this handler will than be executed.

An exception handler may be defined for a specific block:

```
void f(int i) {
    try {               // exceptions in this try block are handled by the
                        // exception handler defined below
        vector v(i);
        // ...
        v[i] = 7;       //  causes vector::range exception
        // ...
        int i = g();    // might cause a vector::range exception
    }
    catch (vector::range) {
            error("f(): vector range error");
            return;
    }
}
```

There are many ways of defining exceptions and the behavior of exception handlers. The facility sketched here resembles the ones found in Clu and ML.

A poor implementation of exception handling can be a serious drain on run-time efficiency and the portability of language implementations. The C++ exception handling can be implemented so that code is not executed unless an exception is thrown or portably across C implementations by (implicitly) using the C standard library functions `setjmp()` and `longjmp()`.

Type conversions

User-defined type conversions, such as the one from floating point numbers to complex numbers implied by the constructor `complex(double)`, have proven unexpectedly useful in C++. Such conversions can be applied explicitly or the programmer can rely on the compiler to add them implicitly where necessary and unambiguous:

```
complex a = complex(1);
complex b = 1;            // implicit: 1 -> complex(1)
a = b+complex(2);
a = b+2;                  // implicit: 2 -> complex(2)
```

User-defined type conversions were introduced into C++ because mixed mode arithmetic is the norm in languages for numerical work and because most user-defined types used for ‘‘calculation’’ (for example, matrices, character strings, and machine addresses) have natural mappings to and/or from other types.

One use of coercions has proven especially useful from a program organization point of view:

---

port exception handling are rare at the time of writing.

```
complex a = 2;
complex b = a+2;  // interpreted as operator+(a,complex(2))
b = 2+a;          // interpreted as operator+(complex(2),a)
```

Only one function is needed to interpret ‘‘+’’ operations and the two operands are handled identically by the type system. Furthermore, class `complex` is written without any need to modify the concept of integers to enable the smooth and natural integration of the two concepts. This is in contrast to a ‘‘pure object-oriented system’’ where the operations would be interpreted like this:

```
a+2;    // a.operator+(2)
2+a;    // 2.operator+(a)
```

making it necessary to modify class `integer` to make `2+a` legal. Modifying existing code should be avoided as far as possible when adding new facilities to a system. Typically, object-oriented programming offers superior facilities for adding to a system without modifying existing code. In this case, however, data abstraction facilities provide a better solution.

Iterators

It has been claimed that a language supporting data abstraction must provide a way of defining control structures[12]. In particular, a mechanism that allows a user to define a loop over the elements of some type containing elements is often needed. This must be achieved without forcing a user to depend on details of the implementation of the user-defined type. Given a sufficiently powerful mechanism for defining new types and the ability to overload operators, this can be handled without a separate mechanism for defining control structures.

For a vector, defining an iterator is not necessary since an ordering is available to a user through the indices. I’ll define one anyway to demonstrate the technique. There are several possible styles of iterators. My favorite relies on overloading the function application operator `()`†:

```
class vector_iterator {
    vector& v;
    int i;
public:
    vector_iterator(vector& r) { i = 0; v = r; }
    int operator()() { return i<v.size() ? v.elem(i++) : 0; }
};
```

A `vector_iterator` can now be declared and used for a `vector` like this:

```
void f(vector& v)
{
        vector_iterator next(v);
        int i;
        while (i=next()) print(i);  // maybe too ‘cute’
}
```

More than one iterator can be active for a single object at one time, and a type may have several different iterator types defined for it so that different kinds of iteration may be performed. An iterator is a rather simple control structure. More general mechanisms can also be defined. For example, the C++ standard library provides a co-routine class[16].

For many ‘‘container’’ types, such as `vector`, one can avoid introducing a separate iterator type by defining an iteration mechanism as part of the type itself. A `vector` might be defined to have a ‘‘current element’’:

† This style also relies on the existence of a distinguished value to represent ‘‘end of iteration’’. Often, in particular for C++ pointer types, `0` can be used.

```
class vector {
    int* v;
    int  sz;
    int current;
public:
    // ...
    int next() { return (current < sz) ? v[current++] : 0; }
    int prev() { return (0 <= --current) ? v[current] : 0; }
};
```

Then the iteration can be performed like this:

```
vector v(sz);
int i;
while (i=v.next()) print(i);
```

This solution is not as general as the iterator solution, but avoids overhead in the important special case where only one kind of iteration is needed and where only one iteration at a time is needed for a vector. If necessary, a more general solution can be applied in addition to this simple one. Note that the ‘‘simple’’ solution requires more foresight from the designer of the container class than the iterator solution does. The iterator-type technique can also be used to define iterators that can be bound to several different container types thus providing a mechanism for iterating over different container types with a single iterator type.

Multiple Implementations

The basic mechanism for supporting object-oriented programming, derived classes, and virtual functions can be used to support data abstraction by allowing several different implementations for a given type. Consider again the stack example:

```
template<class T>
        class stack {
        public:
                virtual void push(T) = 0; // pure virtual function
                virtual T pop() = 0;      // pure virtual function
        };
```

The `=0` notation specifies that no definition is required for the virtual function and that the class is abstract, that is, the class can only be used as a base class. This allows stacks to be used, but not created:

```
stack<cat> s; // error: stack is abstract

void some_function(stack<cat>& s, cat kitty) // ok
{
        s.push(kitty);
        cat c2 = s.pop();
        // ...
}
```

Since no representation is specified in the stack interface, its users are totally insulated from implementation details.

We can now provide several distinct implementations of stacks. For example, we can provide a stack implemented with an array

```
template<class T>
        class astack : public stack<T> {
                // actual representation of a stack object
                // in this case an array
                // ...
        public:
                astack(int size);
                ~astack();

                void push(T);
                T pop();
        };
```

and elsewhere a stack implemented using a linked list:

```
template<class T>
        class lstack : public stack<T> {
                // ...
        };
```

We can now create and use stacks:

```
void g()
{
        lstack<cat> s1(100);
        astack<cat> s2(100);

        cat ginger;
        cat snowball;

        some_function(s1,ginger);
        some_function(s2,snowball);
}
```

Only the creator of `stacks`, `g()`, needs to worry about different kinds of `stacks`, the user `some_function()` is totally insulated from such details. The price of this flexibility is that each operation on such a type must be a virtual function.

Implementation Issues

The support needed for data abstraction is primarily provided in the form of language features implemented by a compiler. However, parameterized types are best implemented with support from a linker with some knowledge of the language semantics, and exception handling requires support from the run-time environment. Both can be implemented to meet the strictest criteria for both compile time speed and efficiency without compromising generality or programmer convenience.

As the power to define types increases, programs to a larger degree depend on types from libraries (and not just those described in the language manual). This naturally puts greater demands on facilities to express what is inserted into or retrieved from a library, facilities for finding out what a library contains, facilities for determining what parts of a library are actually used by a program, etc.

For a compiled language facilities for calculating the minimal compilation necessary after a change become important. It is essential that the linker/loader – with suitable help from the compiler – is capable of bringing a program into memory for execution without also bringing in large amounts of related, but unused, code. In particular, a library/linker/loader system that brings the code for every operation on a type into core just because the programmer used one or two operations on the type is worse than useless.

4 Support for Object-Oriented programming

The basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance and a mechanism that allows calls of member functions to depend on the actual type of an object (in cases where the actual type is unknown at compile time). The design of the member function calling mechanism is critical. In addition, facilities supporting data abstraction techniques (as described above) are important because the arguments for data abstraction and for its refinements to support elegant use of types are equally valid where support for object-oriented programming is available. The success of both techniques hinges on the design of types and on the ease, flexibility, and efficiency of such types. Object-oriented programming allows user-defined types to be far more flexible and general than the ones designed using only data abstraction techniques.

Calling Mechanisms

The key language facility supporting object-oriented programming is the mechanism by which a member function is invoked for a given object. For example, given a pointer `p`, how is a call `p->f(arg)` handled? There is a range of choices.

In languages such as C++ and Simula, where static type checking is extensively used, the type system can be employed to select between different calling mechanisms. In C++, two alternatives are available:

[1] A normal function call: the member function to be called is determined at compile time (through a lookup in the compiler's symbol tables) and called using the standard function call mechanism with an argument added to identify the object for which the function is called. Where the ''standard function call'' is not considered efficient enough, the programmer can declare a function `inline` and the compiler will attempt to inline expand its body. In this way, one can achieve the efficiency of a macro expansion without compromising the standard function semantics. This optimization is equally valuable as a support for data abstraction.

[2] A virtual function call: The function to be called depends on the type of the object for which it is called. This type cannot in general be determined until run time. Typically, the pointer `p` will be of some base class `B` and the object will be an object of some derived class `D` (as was the case with the base class `shape` and the derived class `circle` above). The call mechanism must look into the object and find some information placed there by the compiler to determine which function `f` is to be called. Once that function is found, say `D::f`, it can be called using the mechanism described above. The name `f` is at compile time converted into an index into a table of pointers to functions. This virtual call mechanism can be made essentially as efficient as the ''normal function call'' mechanism. In the standard C++ implementation, only five additional memory references are used.

In languages with weak static type checking a more elaborate mechanism must be employed. What is done in a language like Smalltalk is to store a list of the names of all member functions (methods) of a class so that they can be found at run time:

[3] A method invocation: First the appropriate table of method names is found by examining the object pointed to by `p`. In this table (or set of tables) the string `"f"` is looked up to see if the object has an `f()`. If an `f()` is found it is called; otherwise some error handling takes place. This lookup differs from the lookup done at compile time in a statically checked language in that the method invocation uses a method table for the actual object.

A method invocation is inefficient compared with a virtual function call, but more flexible. Since static type checking of arguments typically cannot be done for a method invocation, the use of methods must be supported by dynamic type checking.

Type Checking

The shape example showed the power of virtual functions. What, in addition to this, does a method invocation mechanism do for you? You can attempt to invoke *any* method for *any* object.

The ability to invoke any method for any object enables the designer of general purpose libraries to push the responsibility for handling types onto the user. Naturally this simplifies the design of libraries. However, it then becomes the responsibility of the user to avoid type mismatches like this:

```
// assume dynamic type checking.
// *** NOT C++ ***

Stack s;  // Stack can hold pointers to objects of any type

cs.push(new Saab900);
cs.push(new Saab37B);

cs.pop()->takeoff();   // fine: a Saab37B is a plane

cs.pop()->takeoff();   // Oops! Run time error: a Saab 900 is a car
                       // a car does not have a takeoff method.
```

An attempt to use a `car` as a `plane` will be detected by the message handler and an appropriate error handler will be called. However, that is only a consolation when the user is also the programmer. The absence of static type checking makes it difficult to guarantee that errors of this class are not present in systems delivered to end-users.

Combinations of parameterized classes and the use of virtual functions can approach the flexibility, ease of design, and ease of use of libraries designed with method lookup without relaxing the static type checking or incurring significant run time overheads (in time or space). For example:

```
stack<plane*> cs;

cs.push(new Saab900);    // Compile time error:
                         // type mismatch: car* passed, plane* expected
cs.push(new Saab37B);

cs.pop()->takeoff();            // fine: a Saab 37B is a plane
cs.pop()->takeoff();
```

The use of static type checking and virtual function calls leads to a somewhat different style of programming than does dynamic type checking and method invocation. For example, a Simula or C++ class specifies a fixed interface to a set of objects (of any derived class) whereas a Smalltalk class specifies an initial set of operations for objects (of any subclass). In other words, a Smalltalk class is a minimal specification and the user is free to try operations not specified whereas a C++ class is an exact specification and the user is guaranteed that only operations specified in the class declaration will be accepted by the compiler.

Inheritance

Consider a language having some form of method lookup without having an inheritance mechanism. Could that language be said to support object-oriented programming? I think not. Clearly, you could do interesting things with the method table to adapt the objects' behavior to suit conditions. However, to avoid chaos, there must be some systematic way of associating methods and the data structures they assume for their object representation. To enable a user of an object to know what kind of behavior to expect, there would also have to be some standard way of expressing what is common to the different behaviors the object might adopt. This ''systematic and standard way'' would be an inheritance mechanism.

Consider a language having an inheritance mechanism without virtual functions or methods. Could that language be said to support object-oriented programming? I think not: the shape example does not have a good solution in such a language. However, such a language would be noticeably more powerful than a ''plain'' data abstraction language. This contention is supported by the observation that many Simula and C++ programs are structured using class hierarchies without virtual functions. The ability to express commonality (factoring) is an extremely powerful tool. For example, the problems associated with the need to have a common representation of all shapes could be solved. No union would be needed. However, in the absence of virtual functions, the programmer would have to resort to the use of ''type fields'' to determine actual types of objects, so the problems with the lack of modularity of the code would remain†.

This implies that class derivation (subclassing) is an important programming tool in its own right. It

† This is the problem with Simula's `inspect` statement and the reason it does not have a counterpart in C++.

can be used to support object-oriented programming, but it has wider uses. This is particularly true if one identifies the use of inheritance in object-oriented programming with the idea that a base class expresses a general concept of which all derived classes are specializations. This idea captures only part of the expressive power of inheritance, but it is strongly encouraged by languages where every member function is virtual (or a method). Given suitable controls of what is inherited (see Snyder[18] and Stroustrup[19]), class derivation can be a powerful tool for creating new types. Given a class, derivation can be used to add and/or subtract features. The relation of the resulting class to its base cannot always be completely described in terms of specialization; factoring may be a better term.

Derivation is another tool in the hands of a programmer and there is no foolproof way of predicting how it is going to be used – and it is too early (even after almost 25 years of Simula) to tell which uses are simply mis-uses.

Multiple Inheritance

When a class `A` is a base of class `B`, a `B` inherits the attributes of an `A`; that is, a `B` is an `A` in addition to whatever else it might be. Given this explanation it seems obvious that it might be useful to have a class `B` inherit from two base classes `A1` and `A2`. This is called multiple inheritance[23].

A fairly standard example of the use of multiple inheritance would be to provide two library classes `displayed` and `task` for representing objects under the control of a display manager and co-routines under the control of a scheduler, respectively. A programmer could then create classes such as

```
class my_displayed_task : public displayed, public task {
    // my stuff
};

class my_task : public task {  // not displayed
    // my stuff
};

class my_displayed : public displayed {  // not a task
    // my stuff
};
```

Using (only) single inheritance only two of these three choices would be open to the programmer. This leads to either code replication or loss of flexibility – and typically both. In C++ this example can be handled as shown above with to no significant overheads (in time or space) compared to single inheritance and without sacrificing static type checking[20].

Ambiguities are handled at compile time:

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B { /* ... */ };

void g(C* p)
{
        p->f(); // error: ambiguous
}
```

In this, C++ differs from the object-oriented Lisp dialects that support multiple inheritance. In these Lisp dialects ambiguities are resolved by considering the order of declarations significant, by considering objects of the same name in different base classes identical, or by combining methods of the same name in base classes into a more complex method of the highest class.

In C++, one would typically resolve the ambiguity by adding a function:

```
class C : public A, public B {
        // ...
public:
        void f();
        // ...
};

void f()
{
        // C's own stuff
        A::f();
        B::f();
}
```

In addition to this straightforward concept of independent multiple inheritance there appears to be a need for a more general mechanism for expressing dependencies between classes in a multiple inheritance lattice. In C++, the requirement that a sub-object should be shared by all other sub-objects in a class object is expressed through the mechanism of a virtual base class:

```
class W { /* ... */ }; // window

class Bwindow: public virtual W { // window with border
        // ...
};

class Mwindow : public virtual W { // window with menu
        // ...
};

class BMW : public Bwindow, public Mwindow {
        // window with border and menu
        // ...
};
```

Here the (single) `window` sub-object is shared by the `Bwindow` and `Bwindow` sub-objects of a `BMW`. The Lisp dialects provide concepts of method combination to ease programming using such complicated class hierarchies. C++ does not.

Encapsulation

Consider a class member (either a data member or a function member) that needs to be protected from ‘‘unauthorized access.’’ What choices can be reasonable for delimiting the set of functions that may access that member? The ‘‘obvious’’ answer for a language supporting object-oriented programming is ‘‘all operations defined for this object,’’ that is, all member functions. A non-obvious implication of this answer is that there cannot be a complete and final list of all functions that may access the protected member since one can always add another by deriving a new class from the protected member’s class and define a member function of that derived class. This approach combines a large degree of protection from accident (since you do not easily define a new derived class ‘‘by accident’’) with the flexibility needed for ‘‘tool building’’ using class hierarchies (since you can ‘‘grant yourself access’’ to protected members by deriving a class). For example:

```
class Window {
        // ...
protected:
        Rectangle inside;
        // ...
public:
        // ...
};
```

```
class Dumb_terminal : Window {
        // ...
public:
        void prompt();
        // ...
};
```

Here `Window` specifies `inside` as protected so that derived classes such as `Dumb_terminal` can read it and figure out what part of the `Window`'s area it may manipulate.

Unfortunately, the ''obvious'' answer for a language oriented towards data abstraction is different: ''list the functions that need access in the class declaration.'' There is nothing special about these functions. In particular, they need not be member functions. A non-member function with access to private class members is called a `friend` in C++. Class `complex` above was defined using `friend` functions. It is sometimes important that a function may be specified as a `friend` in more than one class. Having the full list of members and friends available is a great advantage when you are trying to understand the behavior of a type and especially when you want to modify it.

Encapsulation issues increase dramatically in importance with the size of the program and with the number and geographical dispersion of its users. See Snyder[18] and Stroustrup[19] for more detailed discussions of language support for encapsulation.

Implementation Issues

The support needed for object-oriented programming is primarily provided by the run-time system and by the programming environment. Part of the reason is that object-oriented programming builds on the language improvements already pushed to their limit to support for data abstraction so that relatively few additions are needed†.

The use of object-oriented programming blurs the distinction between a programming language and its environment further. Since more powerful special- and general-purpose user-defined types can be defined their use pervades user programs. This requires further development of both the run-time system, library facilities, debuggers, performance measuring, monitoring tools, etc. Ideally these are integrated into a unified programming environment. Smalltalk is the best example of this.

5 Limits to Perfection

A major problem with a language defined to exploit the techniques of data hiding, data abstraction, and object-oriented programming is that to claim to be a general purpose programming language it must

[1] Run on traditional machines.

[2] Coexist with traditional operating systems.

[3] Compete with traditional programming languages in terms of run time efficiency.

[4] Cope with every major application area.

This implies that facilities must be available for effective numerical work (floating point arithmetic without overheads that would make Fortran appear attractive), and that facilities must be available for access to memory in a way that allows device drivers to be written. It must also be possible to write calls that conform to the often rather strange standards required for traditional operating system interfaces. In addition, it should be possible to call functions written in other languages from a object-oriented programming language and for functions written in the object-oriented programming language to be called from a program written in another language.

Another implication is that an object-oriented programming language cannot completely rely on mechanisms that cannot be efficiently implemented on a traditional architecture and still expect to be used as a general purpose language. A very general implementation of method invocation can be a liability unless there are alternative ways of requesting a service.

Similarly, garbage collection can become a performance and portability bottleneck. Most object-

† This assumes that an object-oriented language does indeed support data abstraction. However, the support for data abstraction is often deficient in such languages. Conversely, languages that support data abstraction are typically deficient in their support of object-oriented programming.

oriented programming languages employ garbage collection to simplify the task of the programmer and to reduce the complexity of the language and its compiler. However, it ought to be possible to use garbage collection in non-critical areas while retaining control of storage use in areas where it matters. As an alternative, it is feasible to have a language without garbage collection and then provide sufficient expressive power to enable the design of types that maintain their own storage. C++ is an example of this.

Exception handling and concurrency features are other potential problem areas. Any feature that is best implemented with help from a linker can become a portability problem.

The alternative to having ‘‘low level’’ features in a language is to handle major application areas using separate ‘‘low level’’ languages.

6 Conclusions

Object-oriented programming is programming using inheritance. Data abstraction is programming using user-defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction. These techniques need proper support to be effective. Data abstraction primarily needs support in the form of language features and object-oriented programming needs further support from a programming environment. To be general purpose, a language supporting data abstraction or object-oriented programming must enable effective use of traditional hardware.

7 Acknowledgements

An earlier version of this paper was presented to the Association of Simula Users meeting in Stockholm. The discussions there caused many improvements both in style and contents. Brian Kernighan and Ravi Sethi made many constructive comments. Also thanks to all who helped shape C++.

8 References

[1] Birtwistle, Graham et.al.: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1971. Chartwell-Bratt Ltd, UK. 1980.

[2] Dahl, O-J. and Hoare, C.A.R.: *Hierarchical Program Structures*. In *Structured Programming*. Academic Press 1972.

[3] Cargill, Tom A.: *PI: A Case Study in Object-Oriented Programming*. SIGPLAN Notices, November 1986, pp 350-360.

[4] C.C.I.T.T Study Group XI: *CHILL User’s Manual*. CHILL Bulletin no 1. vol 4. March 1984.

[5] Ellis, M.A and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley 1990.

[6] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley 1983.

[7] Ichbiah, J.D. et.al.: *Rationale for the Design of the Ada Programming Language*. SIGPLAN Notices, June 1979.

[8] Kernighan, B.W. and Ritchie, D.M.: *The C Programming Language*. Prentice-Hall 1978. 2nd Edition 1988.

[9] Keene, Sonya E.: *Object-Oriented Programming in COMMON LISP* Addison-Wesley 1988.

[10] Kerr, Ron: *Object-Based Programming: A Foundation for Reliable Software*. Proceedings of the 14th SIMULA Users’ Conference. August 1986, pp 159-165. An abbreviated version of this paper can be found under the title *A Materialistic View of the Software ‘‘Engineering’’ Analogy* in SIGPLAN Notices, March 1987, pp 123-125.

[11] Liskov, Barbara et. al.: *Clu Reference Manual*. MIT/LCS/TR-225, October 1979.

[12] Liskov, Barbara et. al.: *Abstraction Mechanisms in Clu*. CACM vol 20, no 8, August 1977, pp 564-576.

[13] Milner, Robert: *A Proposal for Standard ML*. ACM Symposium on Lisp and Functional Programming. 1984, pp 184-197.

[14] Nygaard, Kristen: *Basic Concepts in Object Oriented Programming*. SIGPLAN Notices, October 1986, pp 128-132.

[15] Rovner, Paul: *Extending Modula-2 to Build Large, Integrated Systems*. IEEE Software, Vol. 3. No. 6. November 1986, pp 46-57.

[16] Shopiro, Jonathan: *Extending the C++ Task System for Real-Time Applications*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[17] SIMULA Standards Group, 1984: *SIMULA Standard*. ASU Secretariat, Simula a.s. Post Box 150 Refstad, 0513 Oslo 5, Norway.

[18] Snyder, Alan: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. SIGPLAN Notices, November 1986, pp 38-45.

[19] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986. 2nd Edition 1991.

[20] Stroustrup, Bjarne: *Multiple Inheritance for C++*. Proceedings of the Spring'87 EUUG Conference. Helsinki, May 1987.

[21] Stroustrup, Bjarne: *The Evolution of C++: 1985-1989*. USENIX Computer Systems, Vol 2 No 3, Summer 1989.

[22] Stroustrup, Bjarne: *Possible Directions for C++: 1985-1987*. Proc. USENIX C++ Workshop, Santa Fe, November 1987.

[23] Weinreb, D. and Moon, D.: *Lisp Machine Manual*. Symbolics, Inc. 1981.

[24] Wirth, Niklaus: *Programming in Modula-2*. Springer-Verlag, 1982.

[25] Woodward, P.M. and Bond, S.G.: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974.

The Design of C++0x

Bjarne Stroustrup
Texas A&M University
http://www.research.att.com/~bs

Caveat

- There are people who want "just the facts, just the technical details"
  - I'm writing the C++0x FAQ for you
  - If you really want all the details, read the draft standard (hard)
  - Technical details in isolation are sterile
- There are people who want "grand theory, fundamental principles, and no distracting details"
  - I don't do that
  - Theory in isolation is sterile
- This talk
  - Gives a bit of background (history) and some simple design principles illustrated by the simplest code examples I can find

Overview

- Aims, Ideals, and history
- C++
- Design rules for C++0x
  - With examples
- Case studies
  - Initialization
  - Concurrency

8000+ Programming Languages

- C++'s family tree (part of)

- And this is a gross oversimplification!

Programming languages

- A programming language exists to help people express ideas
- Programming language features exist to serve design and programming techniques
- The primary value of a programming language is in the applications written in it

- The quest for better languages has been long and continues

Assembler –1951

- Machine code to assembler and libraries
  - Abstraction
  - Efficiency
  - Testing
  - documentation

THE USE OF SUB-ROUTINES IN PROGRAMMES

D. J. Wheeler

Cambridge & Illinois Universities

absolutely necessary and that the prime objectives to be born in mind when constructing them are simplicity of use, correctness of codes and accuracy of description. All complexities should-if possible -be buried out of sight.

Fortran –1956

- A notation fit for humans
  - For a specific application domain
    - **A(I) = B(I)+C*D(I)**
  - Efficiency a premium
  - Portability

Simula –1967

- Organize code to “model the real world”
  - Object-oriented design
- Let the users define their own types (classes)
  - In general: concepts/ideas map to classes
  - “Data abstraction”
- Organize classes into hierarchies
  - Object-oriented programming

C –1974

- An simple and general notation for systems programming
  - Somewhat portable
  - Direct mapping of objects and basic operations to machine
    - Performance becomes somewhat portable

C with Classes –1980

- General abstraction mechanisms to cope with complexity
  - From Simula
- General close-to-hardware machine model for efficiency
  - From C

  - Became C++ in 1984
  - Commercial release 1985
  - ISO standard 1998
  - 2nd ISO standard 200x (‘x’ is hex ☹)

ISO Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

From day 1 (1980)

From mid-1983

From about 1990

- The most effective styles use a combination of techniques

Focus of C++0x work

What's distinctive about C++?

- Stability
  - Essential for real-world software
  - 1985-2008
  - 1978-2008 (C and C with Classes)
- Non-proprietary
  - Yet almost universally supported
  - ISO standard from 1998
- Direct interface to other languages
  - Notably C, assembler, Fortran
- Abstraction + machine model
  - Zero overhead principle
    - For basic operations and abstraction mechanisms
  - User-defined types receive the same support as built-in types
  - Standard library written in the language itself
    - And most non-standard libraries

C++ is everywhere

- http://www.research.att.com/~bs/applications.html

- Telecommunications
- Google
- Microsoft applications and GUIs
- Linux tools and GUIs
- Games
- PhotoShop
- Finance
- …

- Mars Rovers
- Marine diesel engines
- Cell phones
- Human genome project
- Micro electronics design and manufacturing
- …

C++ ISO Standardization

- Slow, bureaucratic, democratic, formal process
- About 22 nations (5 to 12 at a meeting)
- Membership have varied
  - 100 to 200+
    - 200+ members currently
  - 40 to 100 at a meeting
    - ~60 currently
- Most members work in industry
- Most members are volunteers
  - Even many of the company representatives
- Most major platform, compiler, and library vendors are represented
  - E.g., IBM, Intel, Microsoft, Sun
- End users are underrepresented

Design?

- Can a committee design?
  - No (at least not much)
  - Few people consider or care for the whole language
- Is C++0x designed
  - Yes
    - Well, mostly
    - You can see traces of different personalities in C++0x
- Committees
  - Discuss
  - Bring up problems
  - "Polish"

What is C++?

Template meta-programming!

A hybrid language

A multi-paradigm programming language

Buffer overflows

It's C!

Embedded systems programming language

Too big!

Supports generic programming

Low level!

An object-oriented programming language

A random collection of features

C++0x

- It *feels* like a new language
  - Compared to C++98
- How can I categorize/characterize it?
- It's *not* just "object oriented"
  - Many of the key user-defined abstractions are not objects
    - Types
    - Classifications and manipulation of types (types of types)
      - I miss "concepts"
    - Algorithms (generalized versions of computation)
    - Resources and resource lifetimes
- The pieces (language features) fit together much better than they used to

C++

A language for building software infrastructures and resource-constrained applications

A light-weight abstraction programming language

So, what does “light-weight abstraction” mean?

- The design of programs focused on the design, implementation, and use of abstractions
  - Often abstractions are organized into libraries
    - So this style of development has been called “library-oriented”
- C++ emphasis
  - Flexible static type system
  - Performance (in time and space)
  - Small abstractions

Overall goals for C++0x

- Make C++ a better language for systems programming and library building
  - Rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development)
  - Build directly on C++'s contributions to systems programming

- Make C++ easier to teach and learn
  - Through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

Rules of thumb / Ideals

- Integrating features to work in combination is the key
  - And the most work
  - The whole is much more than the simple sum of its part

- Maintain stability and compatibility
- Prefer libraries to language extensions
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Make only changes that change the way people think
- Fit into the real world

Maintain stability and compatibility

- "Don't break my code!"
  - There are billions of lines of code "out there"
  - There are millions of C++ programmers "out there"

- "Absolutely no incompatibilities" leads to ugliness
  - We introduce new keywords as needed: **concept**, **auto** (recycled), **decltype**, **constexpr**, **thread_local**, **nullptr**, **axiom**
  - Example of incompatibility:

```
static_assert(4<=sizeof(int),"error: small ints");
```

- "Absolutely no incompatibilities" leads to absurdities

```
_Bool                  // C99 boolean type
typedef _Bool bool;    // C99 standard library typedef
```

Support both experts and novices

- *Example*: minor syntax cleanup

```
vector<list<int>> vl;          // note the "missing space"
```

- *Example*: simplified iteration

```
for (auto x : v) cout << x <<'\n';
```

- *Note*: Experts don't easily appreciate the needs of novices
  - Example of what we couldn't get just now

```
string s = "12.3";
double x = lexical_cast<double>(s);          // extract value from string
```

Prefer libraries to language extensions

- Libraries deliver more functionality
- Libraries are immediately useful
- *Problem*: Enthusiasts prefer language features
  - see library as 2nd best
- *Example*: New library components
  - **std::thread**, **std::unique_future**, …
    - Threads ABI; not thread built-in type
  - **std::unordered_map**, **std::regex**, …
    - Not built-in associative array
- *Example*: Mixed language/library extension
  - The new **for** works for every type with **std::begin()** and **std::end()**
  - The new initializer lists are based on **std::initializer_list<T>**

```
vector<string> v = { "Nygaard ", "Ritchie" };
for (auto& x : {y,z,ae,ao,aa}) cout << x <<'\n';
```

Prefer generality to specialization

- *Example*: Prefer improvements to abstraction mechanisms over separate new features
  - Inherited constructor

```
template<class T> class Vector : std::vector<T> {
    using vector::vector<T>;          // inherit all constructors
    // …
};
```

  - Move semantics supported by rvalue references

```
template<class T>  class vector {
    // …
    void push_back(const T&& x);      // move x into vector
                                      // avoid copy if possible
};
```

- *Problem*: people love small isolated features

Increase type safety

- Approximate the unachievable ideal
  - *Example*: Strongly-typed enumerations

```
enum class Color { red, blue, green };
int x = Color::red;          // error: no Color->int conversion
Color y = 7;                 // error: no int->Color conversion
Color z = red;               // error: red not in scope
Color c = Color::red;        // fine
```

  - *Example*: Support for general resource management
    - **std::unique_ptr** (for ownership)
    - **std::shared_ptr** (for sharing)
    - Garbage collection ABI

Improve performance and the ability to work directly with hardware

- Embedded systems programming is very important
  - *Example*: address array/pointer problems
    - **array<int,7> s;** *// fixed-sized array*
  - *Example*: Generalized constant expressions (think ROM)

```
constexpr int abs(int i) { return (0<=i) ? i : -i; }

struct Point {
   int x, y;
   constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point p1(1,2);          // ok
constexpr Point p2(1,abs(x));  // error unless x is a constant expression
```

Make only changes that change the way people think

- Think/remember:
  - Object-oriented programming
  - Generic programming
  - Concurrency
  - …
- But, most people prefer to fiddle with details
  - So there are dozens of small improvements
    - All useful somewhere
    - **long long**, **static_assert**, raw literals, **thread_local**, unicode types, …
  - *Example*: A null pointer keyword

```
void f(int);
void f(char*);
f(0);          // call f(int);
f(nullptr);    // call f(char*);
```

Fit into the real world

- *Example*: Existing compilers and tools must evolve
  - Simple complete replacement is impossible
  - Tool chains are huge and expensive
  - There are more tools than you can imagine
  - C++ exists on *many* platforms
    - So the tool chain problems occur N times
      - (for each of M tools)

- *Example*: Education
  - Teachers, courses, and textbooks
    - Often mired in 1970s thinking (C is the perfect language)
    - or 1980s thinking (OOP Rah Rah Rah)
  - "We" haven't completely caught up with C++98!
    - "legacy code breeds more legacy code"

Areas of language change

- Machine model and concurrency Model
  - Threads library (**std::thread**)
  - Atomic ABI
  - Thread-local storage (**thread_local**)
  - Asynchronous message buffer (**std::future**)
- Support for generic programming
  - (concepts ☹)
  - uniform initialization
  - **auto**, **decltype**, lambdas, template aliases, move semantics, variadic templates, range-**for**, …
- Etc.
  - **static_assert**
  - improved **enum**s
  - **long long**, C99 character types, etc.
  - …

Case studies

- Concurrency
  - “driven by necessity”
- Initialization
  - “language maintenance”

Case study: Concurrency

- What we want
  - Ease of programming
    - Writing correct concurrent code is hard
  - Portability
  - Uncompromising performance
  - System level interoperability
- We can't get everything
  - No one concurrency model is best for everything
  - De facto: we can't get all that much
  - "C++ is a systems programming language"
    - (among other things) implies serious constraints

Concurrency

- Not
  - Massively parallel (scientific) computing
  - Web services
  - Simple high-level abstract model
  - System of real-time guarantees

- Instead
  - A systems-level foundation for all

Concurrency overview

- Foundation
  - Memory model
  - atomics
- Concurrency library components
  - **std::thread**
  - **std::mutex** (several)
  - **std::lock** (several)
  - **std::condition** (several)
  - **std::future, std::promise, std::packaged_task**
  - **std::async()**
- Resource management
  - **std::unique_ptr, std::shared_ptr**
  - GC ABI

Memory model

- A memory model is an agreement between the machine architects and the compiler writers to ensure that most programmers do not have to think about the details of modern computer hardware.

```
// thread 1:          // thread 2:
char c;               char b;
c = 1;                b = 1;
int x = c;            int y = b;
```

x==1 and **y==0** as anyone would expect

(but don't try that for two bitfields of the same word)

Atomics ("here be dragons!")

- Components for fine-grained atomic access
  - provided via operations on atomic objects (in **<cstdatomic>**)
  - Low-level, messy, and shared with C (making the notation messy)
  - what you need for lock-free programming
  - what you need to implement **std::thread**, **std::mutex**, etc.
  - Several synchronization models, CAS, fences, …

```
enum memory_order { // regular (non-atomic) memory synchronization order
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
};
C atomic_load_explicit(const volatile A* object, memory_order);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
bool atomic_compare_exchange_weak_explicit(volatile A* object, C * expected, C
    desired, memory_order success, memory_order failure);
// … lots more …
```

Concurrency: std::thread

```
#include<thread>

void f() { std::cout << "Hello ";
struct F {
    void operator()() { std::cout << "parallel world "; }
};
int main()
{
    std::thread t1{f};        // f() executes in separate thread
    std::thread t2{F()};      // F()() executes in separate thread
}  // spot the bugs
```

Concurrency: std::thread

```
int main()
{
    std::thread t1{f};        // f() executes in separate thread
    std::thread t2{F()};      // F()() executes in separate thread

    t1.join();        // wait for t1
    t2.join();        // wait for t2
}

// and another bug: don't write to cout without synchronization
```

Mutual exclusion: std::mutex

- A **mutex** is a primitive object use for controlling access in a multi-threaded system.
- A **mutex** is a shared object (a resource)
- Simplest use:

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// manipulate shared data:
sh+=1;
m.unlock();
```

Mutual exclusion: std::mutex

- Not all **mutex** uses are simple:

```
std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_for(std::chrono::seconds(10))) {          // Note: time
    // manipulate shared data:
    sh+=1;
    m.unlock();
}
else {
    // we didn't get the mutex; do something else
}
```

RAII for mutexes: std::lock

- A lock represents local ownership of a non-local resource (the **mutex**)

```
std::mutex m;
int sh; // shared data

void f()
{
    // ...
    std::unique_lock lck(m);    // grab (acquire) the mutex
    // manipulate shared data:
    sh+=1;
}   // implicitly release the mutex
```

RAII for mutexes: std::lock

- We can safely use several locks

```
void f() {
    // ...
    std::unique_lock lck1(m1,std::defer_lock);  // make locks but don't yet
                                                // try to acquire the mutexes
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    lock(lck1,lck2,lck3);
    // manipulate shared data
}
```

Future and promise

- future+promise provides a simple way of passing a value from one thread to another
  - No explicit synchronization
  - Exceptions can be transmitted between threads

Future and promise

- Get from a future:

```
X v = f.get();// if necessary wait for the value to get
```

- Put to a promise:

```
try {
    X res;
    // compute a value for res
    p.set_value(res);
} catch (...) {
    // oops: couldn't compute res
    p.set_exception(std::current_exception());
}
```

async()

- Simple launcher (warning: only approved in principle)

```
template<class T, class V> struct Accum {
    // accumulator function object
};

void comp(vector<double>& v) // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);
    auto f0 = async(Accum{&v[0],&v[v.size()/4],0.0});
    auto f1 = async(Accum{&v[v.size()/4],&v[v.size()/2],0.0});
    auto f2 = async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0});
    auto f3 = async(Accum{&v[v.size()*3/4],&v[v.size()],0.0});
    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Future

- Lots of use
  - C++98, C++0x, C++1x, …
- Is there a future for "the C++ model" beyond C++?
  - Direct map to hardware
  - Zero-overhead abstraction
  - Minimal run-time environment
  - Destructor-based resource management
  - Heavy use of stack
- Challenges
  - Small language (or at least much, much smaller)
  - Complete and enforced type safety
  - Concurrency

yes

I think it can be done

Thanks!

- C and Simula
  - Brian Kernighan
  - Doug McIlroy
  - Kristen Nygaard
  - Dennis Ritchie
  - …
- ISO C++ standards committee
  - Steve Clamage
  - Francis Glassborow
  - Andrew Koenig
  - Tom Plum
  - Herb Sutter
  - …
- C++ compiler, tools, and library builders
  - Beman Dawes
  - David Vandevoorde
  - …
- Application builders

More information

- My HOPL-II and HOPL-III papers
- The Design and Evolution of C++ (Addison Wesley 1994)
- My home pages
  - Papers, FAQs, libraries, applications, compilers, …
    - Search for "Bjarne" or "Stroustrup"
  - C++0x FAQ
- The ISO C++ standard committee's site:
  - All documents from 1994 onwards
    - Search for "WG21"
- The Computer History Museum
  - Software preservation project's C++ pages
    - Early compilers and documentation, etc.
      - http://www.softwarepreservation.org/projects/c_plus_plus/
      - Search for "C++ Historical Sources Archive"

C++0x examples

```
// bind a template argument (Currying):
template<class T> using  Vec = std::vector<T,My_alloc<T>>; // an alias
Vec<double> v = { 1, 2.2, 3, 9 };     // Note: general and uniform initialization

sort(v);  // simplicity is the ultimate sophistication (and no spurious overheads)
sort({"Nygaard", "Ritchie", "Richards"}); // error: can sort a constant

for (auto x : v ) cout << x <<'\n';   // simple traversal

// run in parallel:
auto x = asynch([&v]() { return accumulate(v.begin(), v.end(), 0.0); }); // a lambda
// …
double d = x.get();            // if necessary, wait for result
```

C++0x examples

```
struct F {              // function object
    F(const string&);
    double operator()(double, int); // application (function call) operator
    // …
};

auto f = bind(F("Hello"), _1, 42); // _1 is a placeholder
double dd = f(1.23);               // F("hello")(1.23,42);
```

Problem #1: irregularity

- There are four notations and none can be used everywhere

```
int a = 2;              // "assignment style"
int[] aa = { 2, 3 };    // assignment style with list
complex z(1,2);         // "functional style" initialization
x = Ptr(y);             // "functional style" for conversion/cast/construction
```

- Sometimes, the syntax is inconsistent/confusing

```
int a(1);       // variable definition
int b();        // function declaration
int b(foo);     // variable definition or function declaration
```

- We can't use initializer lists except in a few cases

```
string a[] = { "foo", " bar" };          // ok: initialize array variable
vector<string> v = { "foo", " bar" };    // error: initialize vector variable
void f(string a[]);
f( { "foo", " bar" } );                  // error: initializer array argument
```

Is irregularity a real problem?

- Yes, a major source of confusion and bugs
- Can it be solved by restriction?
  - No existing syntax can be used in all cases

```
int a [] = { 1,2,3 };                  // can't use () here
complex<double> z(1,2);                // can't use { } here
struct S { double x,y; } s = {1,2};    // can't use ( ) here
int* p = new int(4);                   // can't use { } or = here
```

  - No existing syntax has the same semantics in all cases

```
typedef char* Pchar;
Pchar p(7);              // error (good!)
Pchar p = Pchar(7);      // "legal" (ouch!)
```

- Principle violated:
  - Uniform support for types (user-defined and built-in)

Problem #2: list workarounds

- Initialize a vector (using push_back)
  - Clumsy and indirect

```
template<class T> class vector {
   // …
   void push_back(const T&) { /* … */ }
   // …
};

vector<double> v;
v.push_back(1.2);
v.push_back(2.3);
v.push_back(3.4);
```

- Principle violated:
  - Support fundamental notions directly ("state intent")

Problem #2: list workarounds

- Initialize vector (using general iterator constructor)
  - Awkward, error-prone, and indirect
  - Spurious use of (unsafe) array

```
template<class T> class vector {
   // …
   template <class Iter>
      vector(Iter first, Iter last) { /* … */ }
   // …
};

int a[ ] = { 1.2, 2.3, 3.4 };                          // bug
vector<double> v(a, a+sizeof(a)/sizeof(int));          // hazard
```

- Principle violated:
  - Support user-defined and built-in types equally well

C++0x: initializer lists

- An initializer-list constructor
  - defines the meaning of an initializer list for a type

```
template<class T> class vector {
        // …
        vector(std::initializer_list<T>);    // initializer list constructor
        // …
};

vector<double> v = { 1, 2, 3.4 };

vector<string> geek_heros = {
    "Dahl", "Kernighan", "McIlroy", "Nygaard ", "Ritchie", "Stepanov"
};
```

C++0x: initializer lists

- Not just for templates and constructors
  - but **std::initializer list** is simple – does just one thing well

```
void f(int, std::initializer_list<int>, int);

f(1, {2,3,4}, 5);
f(42, {1,a,3,b,c,d,x+y,0,g(x+a),0,0,3}, 1066);
```

Uniform initialization syntax

- Every form of initialization can accept the { … } syntax

```
X x1 = X{1,2};
X x2 = {1,2};        // the = is optional
X x3{1,2};
X* p2 = new X{1,2};

struct D : X {
   D(int x, int y) :X{x,y} { /* … */ };
};

struct S {
   int a[3];
   S(int x, int y, int z) :a{x,y,z} { /* … */ };   // solution to old problem
};
```

Uniform initialization semantics

- **X { a }** constructs the same value in every context
  - **{ }** initialization gives the same result in all places where it is legal

```
X x{a};
X* p = new X{a};
z = X{a};          // use as cast
f({a});            // function argument (of type X)
return {a};        // function return value (function returning X)
…
```

- **X { … }** is always an initialization
  - **X var{}** *// no operand; default initialization*
    - Not a function definition like **X var();**
  - **X var{a}** *// one operand*
    - Never a function definition like **X var(a);** (if **a** is a type name)

Initialization problem #3: narrowing

- C++98 implicitly truncates

```
int x = 7.3;                    // Ouch!
char c = 2001;                  // Ouch!
int a[] = { 1,2,3.4,5,6 };      // Ouch!

void f1(int);       f1(7.3);            // Ouch!
void f2(char);      f2(2001);           // Ouch!
void f3(int[]);     f3({ 1,2,3.4,5,6 }); // oh! Another problem
```

- A leftover from before C had casts!
- Principle violated:
  - Type safety
- Solution:
  - C++0x { } initialization doesn't narrow.
    - all examples above are caught

Uniform Initialization

- Example

```
Table phone_numbers = {
    { "Donald Duck", 2015551234 },
    { “Mike Doonesbury", 9794566089 },
    { "Kell Dewclaw", 1123581321 }
};
```

- What is **Table**?
  - a **map**? An array of **struct**s? A **vector** of **pair**s? My own **class** with a constructor? A **struct** needing aggregate initialization? Something else?
  - We don’t care as long as it can be constructed using a C-style string and an integer.
  - Those numbers cannot get truncated

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221249736

C++ Dynamic Cast in Autonomous Space Systems

Conference Paper · May 2008

DOI: 10.1109/ISORC.2008.20 · Source: DBLP

CITATION
1

READS
43

4 authors, including:

Rabi N. Mahapatra
Texas A&M University
164 PUBLICATIONS **1,492** CITATIONS

SEE PROFILE

Bjarne Stroustrup
Morgan Stanley
157 PUBLICATIONS **8,359** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Thermal Management of Photonic NoC View project

Project Power Management of Photonic NoC View project

All content following this page was uploaded by Bjarne Stroustrup on 29 May 2014.

The user has requested enhancement of the downloaded file.

C++ Dynamic Cast in Autonomous Space Systems

Damian Dechev [1], Rabi Mahapatra [1], Bjarne Stroustrup [1], David Wagner [2]
dechev@tamu.edu, rabi@cs.tamu.edu, bs@cs.tamu.edu, david.a.wagner@jpl.nasa.gov

Texas A&M University [1]
College Station, TX 77843-3112

Jet Propulsion Laboratory, California Institute of Technology [2]
4800 Oak Grove Drive, M/S 301-270, Pasadena, CA

Abstract

The dynamic cast operation allows flexibility in the design and use of data management facilities in object-oriented programs. Dynamic cast has an important role in the implementation of the Data Management Services (DMS) of the Mission Data System Project (MDS), the Jet Propulsion Laboratory's experimental work for providing a state-based and goal-oriented unified architecture for testing and development of mission software. DMS is responsible for the storage and transport of control and scientific data in a remote autonomous spacecraft. Like similar operators in other languages, the C++ dynamic cast operator does not provide the timing guarantees needed for hard real-time embedded systems. In a recent study, Gibbs and Stroustrup (G&S) devised a dynamic cast implementation strategy that guarantees fast constant-time performance. This paper presents the definition and application of a co-simulation framework to formally verify and evaluate the G&S fast dynamic casting scheme and its applicability in the Mission Data System DMS application. We describe the systematic process of model-based simulation and analysis that has lead to performance improvement of the G&S algorithm's heuristics by about a factor of 2.

1 Introduction

ISO Standard C++ [8] has become a common choice for hard real-time embedded systems such as the Jet Propulsion Laboratory's Mission Data System [7]. This is so because ISO C++ offers efficient abstraction model, good hardware use, and predictability. C++'s model of computation has helped engineers deliver more correct, maintainable, and comprehensible software compared to code relying on lower-level programming concepts [19]. However, several C++ features are usually considered unsuitable for programming real-time systems because they do not guarantee predicable constant-time performance [5]. ISO C++ does not provide the necessary timing guarantees for free store (heap) allocation, exception handling, and dynamic casting. In particular, the most common compiler implementations of the dynamic cast operator traverse the representation of the inheritance tree (at run time) searching for a match. Such implementations of dynamic cast are not predictable and are unsuitable for real-time programming. Gibbs and Stroustrup(G&S) [3] describe a technique for implementing dynamic cast that delivers significantly improved and constant-time performance. The key idea is to replace the runtime search through the class hierarchy with a simple (constant-time) calculation, much as the common implementations of the C++ virtual function calls search the class hierarchy at compile time to reduce the runtime action to a simple array subscripting operation. In the G&S scheme, a heuristic algorithm assigns an integer type ID at link time to each class. The type ID assignment rules guarantee that at run time a simple modulo operation can reveal the type information and check the validity of the cast. The requirements for the heuristics assigning the type IDs are that:

(1) They must keep the size of the type ID to a small number of bits. A 64-bit type ID should be sufficient for very large class hierarchies

(2) Avoid conflicts and type ID assignments that create ambiguous or erroneous type resolution at run time

(3) Handle virtual inheritance

There are four heuristic schemes and a few possible optimizations suggested in [3]. However, none of those heuristics guarantee the best solution for every possible class hierarchy. The quality of the type ID assignment heuristics has a critical importance for the performance of the G&S scheme. With better heuristics, a smaller type ID size would be sufficient to facilitate complex and large class hierarchies that would certainly need a significantly bigger type ID size with a poor assignment scheme. The main contribution of this work is to present how the algorithm optimization problem

encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme.

As pointed out by Lowry [10], the increasing complexity of future space missions, such as the Mars Science Laboratory [20] and Project Constellation [17], raises concerns whether it is possible to establish their reliability in a cost-effective manner. Lowry's analysis indicates that at the present moment the verification and certification cost of mission critical software exceeds its development cost. Perrow [13] studies the risk factors in the modern high technology systems. His work identifies two significant hazard dimensions: *interactions* and *coupling*. Complex interactions represent unexpected and unknown sequences and thus cannot be entirely comprehensible at the time of system development. A tightly-coupled system has a number of time-dependent processes that cannot tolerate delays. Perrow classifies space missions in the riskiest category since both hazard factors are present. The dominant paradigms for software development, assurance, and management at NASA rely on the principle "test-what-you-fly and fly-what-you-test". Born out of experience and hindsight, this methodology had been applied in a large number of robotic space missions at the Jet Propulsion Laboratory. For such missions, it has proven suitable in achieving adherence to some of the most stringent standards of man-rated certification such as the DO-178B [14], the Federal Aviation Administration (FAA) software standard. Its Level A requirements demand 100% coverage of all high and low level assurance policies. However, the present certification methodologies are prohibitively expensive for systems of high complexity [16].

In this paper we present a co-simulation framework based on the SPIN model checker [6] to simulate, evaluate, and formally verify the G&S fast dynamic casting algorithm and its application in mission critical code such as the Data Management Services [21] of the Mission Data System. The aim of the Mission Data System is to provide a unified state-based and goal-oriented architecture for building complete data and control systems for autonomous space missions. The framework's state- and model-based methodology and its associated systems engineering processes and development tools have been successfully applied on a number of test systems including the physical rovers Rocky 7 and Rocky 8 and a simulated Entry, Descent, and Landing (EDL) system for the Mars Science Laboratory mission. We use the feedback from the model checker to perform systematic analysis of the G&S scheme and look for improvements to the heuristics for type ID assignment. SPIN is an on-the-fly, linear-time logic model-checking tool that was designed for the formal verification of dynamic systems with asynchronously executed processes. The most recent advances in the state space reduction techniques has made it possible to validate large software applications. Model-checking tools have been widely applied for the verification of a large variety of systems, including flight software [4], network protocols [11], and scheduling algorithms [15]. We are unaware of work suggesting its use for the analysis and optimization of compiler heuristics. Compiler verification usually focuses on seeking a proof on the preservation of the program semantics during the various compiler passes [9]. Our work presents the application of a model-checking tool for the analysis and refinement of the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our co-simulation framework consists of the following components:

(1) An abstract model of the G&S type ID assignment heuristics

(2) An procedure for exhaustive search of the state space discovering the best type ID assignment

The analysis of the heuristics simulation performed in SPIN provides us with ideas of possible improvements to the G&S type ID assignment. We include and evaluate the proposed improvements in the abstract model in order to seek refinement of the G&S type ID assignment scheme. The experiments we have executed show that the G&S priority assignment is not optimal with respect to the best possible type ID assignment where non-virtual multiple inheritance is used. While potentially dangerous if not constructed carefully, such hierarchies happen to be of significant practical importance [18]. Based on our experiments, we suggest optimizations that lead to significant improvement of the G&S heuristics performance. This paper makes the following contributions:

(1) Introduces the use of a co-simulation framework based on model-checking for the analysis and improvement of a compiler-heuristics optimization problem

(2) Verifies and analyzes the G&S C++ fast dynamic casting scheme and its application in mission critical code such as the MDS Data Management Services

(3) Implements optimizations to the G&S heuristics leading to the discovery of optimal type ID assignment in 85% of the class hierarchies, in contrast to 48% for the original G&S algorithm

The rest of the paper is organized as follows: section 2: a brief description of the G&S fast dynamic cast algorithm, section 3: our approach to co-simulation and improvements to the G&S heuristics, section 4: discussion on the challenges of mission critical code and the applicability of the G&S dynamic cast section 5: performance results for the G&S algorithm and the proposed improvements, and section 6: conclusion.

2 Fast Dynamic Casting Algorithm

The G&S fast constant-time implementation of the dynamic cast operator works as follows: at link time, a static integer type ID number, preferably 32 or 64-bit long, is assigned to each class. The ID numbers are selected so that the operation id_a *modulus* id_b yields zero if and only if the class with id_a is derived from the class with id_b. This is done by exploiting the uniqueness of factorization of integers into prime factors. Each class is assigned a *key* prime number. The *type ID* of a class is calculated by multiplying its *key number* with the key numbers of each of its base classes. In the cases where a class contains more than a single copy of a base class, the type ID is computed by taking the square of the corresponding base class ID. The only constraint of the approach is the desire to limit the ID size to fit the machine's built-in integer types. The key primes are not required to be unique and the same prime key can be used for classes that belong to different groups (i.e. do not share common descendants). Gibbs and Stroustrup suggest four approaches for assigning the type IDs in a space-efficient manner. Each method is based on a preliminary computation of the priority factor of each class. The priority reflects the class impact on the growth of the type ID numbers in the hierarchy. Thus, classes with greater number of descendants should receive higher priority and smaller key prime number values respectively. The four possible schemes suggest that:

1 The priority of a class is the *maximum* number of ancestors that any of its descendants has. This scheme was chosen for the initial implementation and testing of the G&S algorithm and also closely followed in the implementation of the abstract model used for our simulation

2 ,3, 4. If a range of primes is assigned to every level with wider levels receiving larger initial values, then each node could be assigned an additional value that is proportional to the logarithm of the (*2. minimum, 3. mean, 4. maximum*) prime in its level. Priorities of hierarchy leaves are computed by taking the sum of these additional values for the leaf itself and all of its ancestor classes

After the priority of each class has been computed, the classes with the highest priority get the smallest prime numbers. According to this scheme, prime numbers can be reused only if there are two classes on the same level of the class hierarchy and only if they do not share common descendants, they are not siblings, and also that none of their parents share a common descendant. According to the ID assignment rules, we know that:

(1) $id_x = k_x \times (k_a)^2 \times k_{a_1} \times (k_b)^2 \times k_{b_1} \times k_c$

(2) $id_y = k_y \times k_c \times k_{c_1} \times (k_d)^2 \times k_{d_1} \times k_b$

Figure 1. A class hierarchy with 11 classes

(3) $id_z = k_z \times k_d \times k_{d_1} \times k_c$

Given a set C with 11 classes in the hierarchy and the set of the first 11 prime numbers *P* = {*2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31*}, we must assign each class V a key $k_v \in P$ such that, the maximum of the set $id_{leaf} = \{id_x, id_y, id_z\}$, the set consisting of the ID numbers of all leaf nodes in C, is minimal. As we already know, prime numbers need not be unique for each class and can be reused in same circumstances.

3 A Co-simulation Framework

The goals of the co-simulation framework are to validate the main invariants of the G&S heuristics, improve its performance, and establish its applicability in mission critical systems. The co-simulation process in the framework (Figure 2) consists of three consecutive stages: verification, evaluation, and analysis. The verification phase is a straightforward application of model checking where an abstract description of the system's behavior is checked against a set of invariants. In the evaluation stage the simulation results from the probabilistic approach are contrasted to the outcome of the deterministic approach. The aim of the analysis stage is to closely examine the instances where the solutions yielded by the two implementations differ. We identify patterns among the inconsistent results that reveal the weaknesses of the probabilistic solution. The framework works by executing two independent models, the G&S model and the exhaustive search model. The first input component to the co-simulation framework (Figure 2) is an abstract model of the G&S fast dynamic casting heuristics, implemented in Promela (SPIN's input language) and the embedded C primitives it allows. The G&S abstract model is subsequently used to verify the main invariants of the G&S heuristics and at the same time provide us with a simulation testbed to examine the heuristics performance. The second component of the framework is the exhaustive search model that simply looks into all possible type ID assignments to discover the optimal solution for a given class hierarchy. We employ SPIN's search engine to perform the exhaustive search. In Algorithm 1 we present the pseudocode of our co-simulation approach. The following sections elaborate in more details on each of the stages of

Figure 2. A Co-Simulation Framework for G&S Improvement and Verification

the framework.

Algorithm 1 Pseudocode of the co-simulation approach.

```
const int MAX_NUMBER_TESTS
VERIFY :
repeat
    Formal Verification (G&S Model) → error report
    if (no errors) then
        goto EVALUATE
    else
        study counter example
        correct G&S
until TRUE
EVALUATE :
count = 0
for (count < MAX_NUMBER_TESTS) do
    Simulation(G&S Model) → G&S solution
    Enumeration(Exhaustive Search Model) → best solution
    if (G&S solution ≠ best solution) then
        add instance to SIS
    count ++
ANALYZE :
for all i ∈ SIS do
    look for a pattern
    modify G&S
    goto EVALUATE
```

3.1 Formal Verification

Every G&S implementation operates under the assumption that when a prime number is reused, it is assigned to non-conflicting classes. In addition, the maximum type ID must fit within the boundaries of a memory word. We check these invariants during the program verification phase. Establishing the validity of the G&S invariants is done by straightforward application of model-checking with SPIN. In SPIN the critical system properties are expressed in the syntax of linear time logic. Based on the G&S abstract specification, the model-checker performs a systematic exploration of all possible states. In case of failure, SPIN provides a counterexample that demonstrates a behavior that has lead to an illegal state. In our model, the invariants are expressed as a *never claim* [6], and are checked just before and after the execution of every statement.

3.2 Evaluation

SPIN has been previously employed to implement solutions of scheduling [1] and discrete optimization [15] problems. The problem we face in the G&S heuristics is a combinatorial optimization problem [12]. Given a finite set I, a collection F of subsets of I, and a real-valued function w defined on I, a discrete optimization problem could be defined as the task of finding a member S of F, such that: $\sum_{e \in S} w(e)$ is as small (or as large) as possible.

Except for the simplest cases, a discrete optimization problem is difficult because its design space is typically disjoint and nonconvex. Therefore, the optimization methods applied to continuous optimization problems cannot be utilized in this case. In a small discrete problem, it would be possible to exhaustively list all possible combinations. As the number of parameters increase, the state explosion makes optimizations difficult. The two general strategies for approaching a discrete optimization problem can be classified as *deterministic* and *probabilistic*. What we do for the G&S exploration in SPIN could be described as applying a deterministic approach for the evaluation of a set of proposed probabilistic methods. The Branch and Bound method [12] guarantees the discovery of the global optimum in the cases when the problem is linear or convex and is the most frequently used discrete optimization method. It is based on the sequential analysis of the discrete tree of each parameter. The branches that can be estimated to reach invalid or unfeasible solutions are consequently eliminated. This simple optimization could also be applied in some limited cases in the SPIN's Fast Dynamic Casting exhaustive search. Let us explore a class hierarchy with three classes A, B, and C, where B is derived from A, and C is derived from both A and B. In this case, we have $C = \{A, B, C\}$, $P = \{2, 3, 5\}$, and $id_{leaf} = \{id_c\}$. The enumeration is given in Table 1. We assume that the computation starts at a state S_0 where all three keys k_a, k_b, and k_c are uninitialized. Then we assign possible values from the set P to the key variables of the classes A, B, and C. The enumeration shown above can be expressed as the computation shown on Figure 3. The graph shows only the valid states of the computation. There are a number of invalid states that are not shown on the graph. For example, according to the rules defined in G&S, it is possible to reuse some of the prime numbers in

$id_c = k_c \times k_b \times (k_a)^2$	k_a	k_b	k_c
60	2	3	5
60	2	5	3
90	3	2	5
90	3	5	2
150	5	2	3
150	5	3	2

Table 1. Enumeration of all solutions

Figure 3. Exhaustive search computation

P. Thus, we can try and add an edge $k_b = 2$ in state S_1, however the reuse of 2 in this case is invalid since A and B are conflicting classes.

The illustrated automation in Figure 3 provides a foundation for the construction of a Promela model for the deterministic solution. Each possible prime number assignment to a given class key is represented by a separate state transition in the exhaustive search model. SPIN initiates the optimum search at state S_0 and visits all possible states. At each end state the value of the minimum of the set of leaves, in this case represented only by id_c, is computed and compared to the current minimum. This approach is similar to the algorithm described by Ruys in [15] and shown in Algorithm 2. For such an application, we use the model checker in a somewhat unusual fashion. In this scenario, the validation property checks whether the value of id_c is greater than the current minimum. Each time this condition is violated, the current minimum is updated and the process is automatically repeated until SPIN confirms that there are no routes violating the specification. Since the solution is deterministic, it is guaranteed to discover the global optimum for type ID assignment. The performance of the G&S heuristics is measured by running a simulation of the G&S model that has been used earlier for verification. Now we are left with only one important task (not automated at this stage), the comparison of the results from the probabilistic and deterministic solutions. Once we identify a set of inconsistent results, we try to find a pattern and refine the G&S heuristics. Then the refined scheme is implemented in the probabilistic model and the evaluation process is reiterated.

Algorithm 2 Finding the global minimum in the state space.

1: $intput: \ Promela\ model\ M$
2: $output: \ the\ optimal\ minimum\ for\ the\ problem\ M$
3: $min = (worst\ case)\ maximum\ value\ for\ id$
4: **repeat**
5: $\quad use\ SPIN\ to\ check\ M\ with\ condition\ (id_c > min)$
6: $\quad$ **if** $(error\ found)$ **then**
7: $\quad\quad min = id_c$
8: **until** $(error\ found)$

3.3 Analysis

The simulation and enumeration models are continuously executed until, if possible, a set of instances with inconsistent solutions can be identified. Thus, each instance in the Set of Inconsistent Solutions(SIS), represents a given class hierarchy for which the deterministic and probabilistic approach have discovered different solutions. The class hierarchies for each test could be guided or created in a random fashion. For the generation of the test data in our experiments we implemented a pseudo random class hierarchy generation algorithm, in a manner similar to the TGFF(Task-Graphs-For-Free) method as described in [2]. We look for patterns among the collected hierarchies in SIS and seek clues that can lead us to improvements of the G&S scheme. Potential improvements are tested by adding them to the G&S model and evaluate their effect. Such scheme modifications are carefully selected since it is possible that they might enhance a given G&S feature and at the same time weaken another. Ideally, the improvements lead to a heuristic scheme that provides the best solutions for a larger number of the test hierarchies and at the same time has a time complexity equal to or less than the earlier heuristic scheme.

Despite the numerous advanced state space reduction techniques utilized by the SPIN model checker, little can be done to further optimize the exhaustive search. The main goal of our experiments is to reach quick and effective optimization of the G&S scheme, thus the class hierarchies considered were not the largest and most complex that our models can handle. The models developed for our experiments are capable of handling class hierarchies of double or triple the size of the ones presented in the paper, and even larger number of classes can be facilitated with increased computational power. In the framework, the exhaustive search is used to identify flaws in the G&S type ID assignment scheme, thus, there is no need to create and simulate much larger hierarchies. In this work our goal is to demonstrate that the current size of the class hierarchies is sufficient to discover significant flaws of the original heuristics.

4 Application in Mission-Critical Software

Modern space mission systems have evolved from simple embedded devices into complex computing platforms with high autonomy and an exceptionally large demand for human-computer interaction. Consequently, such systems require reliable and flexible data systems managing the collection, storage, and transportation of data. The Mission Data System(MDS) is the Jet Propulsion Laboratory's state- and goal-oriented framework for building embedded control systems with a high degree of autonomy. MDS provides the building blocks for the implementation of embedded platforms based on the concepts of state estimation and control. The Data Management Services(DMS) is the MDS component responsible for the production, storage, processing, and transfer of control and scientific data. In [21] Wagner defines the challenges of data management in MDS as the problems of producing and storing data and converting the data to various formats as needed by its consumers. In addition, DMS needs to ensure the secure and lossless transport of the data with limited resources and through unreliable physical medium. To design and relate the data system entities, DMS employs concepts from high-level ISO C++ including templates, object-oriented class encapsulation, and dynamic casting necessary for the conversion of the data formats.

The actual telemetry data objects in MDS communicate with each other via byte streams produced by the transport protocol (e.g. spacecraft to ground communication). The receiver of the telemetry data needs to recreate the data object from the byte stream and thus invoke type casting in numerous occasions. Constant-time dynamic cast is also needed by the MDS Goal Network in the case when a controller or estimator [21] passes a goal via the Coordinating Goal Network(CGN), typically a large dynamic data structure. In CGN the goal is propagated using only its abstract attributes(start and end time, and the associated state variable). The achiever object who eventually picks up the goal needs to reconstruct the data object via dynamic downcasting to the specific type that conveys the state-specific achievement criteria. The application of the common compiler implementation of dynamic cast has proved to be unacceptable due to poor performance and the lack of the timing guarantees.

The G&S scheme was devised as a solution to a real industrial problem related to C++ use for hard real time code. Inquiries in the C++ community revealed that the problem was fundamental and common, rather than isolated: developers simulate dynamic casting with other language features, leading to type-unsafe special-purpose code or the avoidance of best object-oriented practices. Naturally, such workaround code slows down development, complicates maintenance, and increases the need for testing.

5 Results

We applied the co-simulation process described in the previous section to a large number of class hierarchies. The tested hierarchies are not built into our models. Instead, we have followed a pseudo random generation methodology similar to TGFF task graph generation as described in [2] to automatically generate hundreds of possible test cases. For illustration, we show the results from a set of seven pseudo random class hierarchies (Appendix A). The results of the G&S heuristics model and the exhaustive search are shown in Table 2. A brief comparison of the results indicates that the G&S heuristics do not give optimal performance for class hierarchies with non-virtual multiple-inheritance. A closer look at the algorithm reveals that the priority calculation routine takes into account only the number of descendants that each class has. Let us consider the class hierarchy from test case 7. We notice that according to the current scheme, the base classes 0, 1, and 2 all get the same priority rank since they all share the descendant 6. Class 6 is at the lowest level of the hierarchy and has the largest number of ancestors. If we would like to optimize the heuristics, we must find a way to increase the priority of base class 2. Our reasoning is derived from the fact that Class 2 is ambiguous and the leaf Class 6 contains two copies of it. Similarly, let us have a closer look at test case 1. In the optimal solution, Class 5 takes the lower prime number (11) compared to Class 4, despite the fact that its only descendant has less ancestors compared to Class 4. The reason for this result is the fact that the derived Class 3 contains two ambiguous bases while Class 4 contains only one ambiguous base. As a result of our analysis we conclude that higher priority should be given to derived classes and their ancestors who contain more ambiguous base classes. To fix these weaknesses, we extend the G&S heuristics by adding two simple rules:

(1) We count every ambiguous ancestor twice when we determine the number of ancestors to each class

(2) For each base class, we count the number of derived classes that include more than one copy of it, and add that number directly to its priority

We call this enhanced G&S heuristics Fast Dynamic Casting Plus(FDC+). As Table 2 shows, for the initial set of test cases, FDC+ performance is 100% equivalent to the performance of the deterministic approach. In the performed tests, we have generated 127 pseudo random class hierarchies and applied G&S, FDC+, and the exhaustive search to each one of them. The experimental results showed that FDC+ was able to yield the best type ID assignment in 85% of the class hierarchies compared to 48% for the G&S heuristics. The time performance of the three schemes is shown in Figure

Case No	G&S	Exhaustive search	FDC+
Case 1 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 2, 5, 7, 13, 11, 17)	(3, 2, 5, 7, 13, 11, 17)
Case 1(ids of all leaves)	(16380, 16830)	(13860, 13260)	(13860, 13260)
Case 2 (keys)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)
Case 2 (ids of all leaves)	(1326, 2310)	(1326, 2310)	(1326, 2310)
Case 3 (keys)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)
Case 3 (ids of all leaves)	(26, 51, 2310)	(26, 51, 2310)	(26, 51, 2310)
Case 4 (keys)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)
Case 4 (ids of all leaves)	(2310, 1547)	(2310, 1547)	(2310, 1547)
Case 5 (keys)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)
Case 5 (ids of all leaves)	(42, 66, 70, 110)	(42, 66, 70, 110)	(42, 66, 70, 110)
Case 6 (keys)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)
Case 6 (ids of all leaves)	(66, 78, 420, 170)	(66, 78, 420, 170)	(66, 78, 420, 170)
Case 7 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)
Case 7 (ids of all leaves)	(2552550)	(1021020)	(1021020)

Table 2. Co-simulation of the seven cases from Appendix A

4. While the time performances of the G&S and FDC+ algorithms are equal and both run in a very low constant-time (the function at 00:01 min on Figure 4), logically the time performance of the exhaustive search increases exponentially with the increase of the number of classes nodes in a given class hierarchy. The analysis of the test results indicated that FDC+ finds a better type ID compared to the G&S approach in 39% of the test scenarios. For the greater part of the test cases, FDC+ matched the optimal type ID assignment computed by the exhaustive search. This efficiency boost is due to the optimized performance of FDC+ in the cases where multiple non-virtual inheritance is present in the class hierarchy. We have also observed that the implementation of these optimizations does not lead to efficiency loss in other scenarios and the performance of FDC+ is always at least as good as the performance of G&S. Our experimental results have indicated that the introduced optimizations in FDC+ have fixed a weakness of the original G&S approach and have improved the success rate in finding the best type ID assignment. The G&S scheme requires a key of a memory size that is a function of the size and shape of a class hierarchy. Thus, the improved heuristics almost double the size of class hierarchies that can be handled by a given key size. Since the scheme gets significantly slower when a key gets too large for a machine word, the improvements to the heuristics address the main limitation of the G&S scheme.

Figure 4. Search time for type ID assignment

6 Conclusion

In this work we applied co-simulation of the deterministic and probabilistic solutions to the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our framework proved successful in verifying and refining the existing G&S heuristics. We demonstrated how we use the simulation results to devise improvements to the G&S algorithm and evaluate them. The results from our experiments indicate that the improved G&S heuristics (FDC+) provide the optimal type ID assignment in 85% of the class hierarchies, compared to 48% for the regular G&S algorithm. The efficiency of the type ID assignment scheme has significant importance for the performance of the fast dynamic casting by Gibbs and Stroustrup [3]. This paper presented a practical approach of how to discover improvements to the type ID assignment scheme in a simple and effective manner. The main advantage of the presented approach is the ease and simplicity of the discovery and test for potential improvements. The improved heuristics that we have described in this work almost doubles the size of class hierarchies that can be handled by a given key size. A more extensive simulation might suggest further improvements to the type ID assignment scheme. Our main goal in this work has been to demonstrated how an algorithm optimization problem encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme. In the future we intend to utilize a static analysis tool for automatic class hierarchy analysis and extraction.

7 Acknowledgements

We thank Peter Pirkelbauer, Kirk Reinholtz, Gerard Holzmann, and the anonymous referees for their helpful suggestions.

A Appendix

Figure 5. Test Cases 1 and 2

Figure 6. Test Cases 3 and 4

Figure 7. Test Cases 5 and 6

Figure 8. Test Case 7

References

[1] E. Brinksma and A. Mader. Verification and Optimization of a PLC Control Schedule. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 73–92, London, UK, 2000. Springer-Verlag.

[2] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society.

[3] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36(2):139–156, 2006.

[4] R. Gluck and G. Holzmann. Using spin model checker for flight software verification. In *In Proceedings of the 2002 IEEE Aerospace Conference*, 2002.

[5] L. Goldthwaite. Technical Report on C++ Performance. In *ISO/IEC PDTR 18015*, February 2006.

[6] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[7] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada. Engineering Complex Embedded Systems with State Analysis and the Mission Data System. In *In Proceedings of First AIAA Intelligent Systems Technical Conference 2004*, 2004.

[8] ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.

[9] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231, New York, NY, USA, 2003. ACM Press.

[10] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[11] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[12] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[13] C. Perrow. *Normal Accidents*. Princeton University Press, September 1999.

[14] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[15] T. C. Ruys. Optimal scheduling using branch and bound with spin 4.0. In T. Ball and S. K. Rajamani, editors, *Model Checking Software, Proceedings of the 10th International SPIN Workshop*, volume 2648 of *Lecture notes in Computer Science*, pages 1–17, Berlin, 2003. Springer Verlag.

[16] J. Schumann and W. Visser. Autonomy Software: V&V Challenges and Characteristics. In *In Proceedings of the 2006 IEEE Aerospace Conference*, 2006.

[17] A. Stoica, D. Keymeulen, A. Csaszar, Q. Gan, T. Hidalgo, J. Moore, J. Newton, S. Sandoval, and J. Xu. Humanoids for lunar and planetary surface operations. In *In Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, October 2005.

[18] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[19] B. Stroustrup. Abstraction and the c++ machine model. In Z. Wu, C. Chen, M. Guo, and J. Bu, editors, *ICESS*, volume 3605 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2004.

[20] R. Volpe. Rover Technology Development and Mission Infusion Beyond Mars Exploration Rover. In *IEEE Aerospace Conference*, March 2005.

[21] D. Wagner. Data Management in the Mission Data System. In *In Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.

View publication stats

C and C++: Case Studies in Compatibility

Bjarne Stroustrup

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

This article gives examples of how one might go about increasing the degree of compatibility of C and C++. The ideal is full compatibility. Topics covered includes, variadic functions, ***void****, ***bool***, ***f*(*void*)**, ***const***, ***inline***, and variable length arrays. These topics allows a demonstration of concerns that must be taken into account when trying to increase C/C++ compatibility.

A companion paper [Stroustrup,2002a] provides a ''philosophical'' view of the C/C++ relationship, and another companion paper presents a case for significantly increased C/C++ compatibility and proposed full compatibility as the ideal. [Stroustrup,2002b].

1 Introduction

Making changes to a language in widespread use, such as C [C89] [C99] and C++ [C++98] is not easy. In reality, even the slightest change requires discussion and consideration beyond what would fit in this article. Consequently, each of the eleven ''case studies'' I present here lacks detail from the perspective of the C and C++ ISO standards committees. However, the point here is not to present complete proposals or to try to tell the standards committees how to do their job. The point is to give examples of directions that might (or might not) be taken and examples of the kind of considerations that will be part of any work to improve C and/or C++ in the direction of greater C/C++ compatibility. The examples are chosen to illustrate both the difficulties and the possibilities involved.

In many cases, the reader will ask ''how did the designers of C and C++ get themselves into such a mess?'' My general opinion is that the designers (not excluding me) and committees (not excluding the one on which I serve) got into those messes for reasons that looked good to competent people on the day, but weren't [Stroustrup,2002].

Let me emphasize that my answer is not a variant of ''let C adopt C++'s rules''. That would be both arrogant and pointless. The opposite suggestion, ''let C++ adopt C's rules'', is equally extreme and unrealistic. To make progress, both languages and language communities must move towards a common center. My suggested resolutions are primarily based on considerations of

[1] what would break the least code
[2] how easy is it to recover from code broken by a change
[3] what would give the greatest benefits in the long run
[4] how complicated would it be to implement the resolution.

Many changes suggested here to increase C/C++ compatibility breaks some code and all involve some work from implementers. Some of the suggested changes would, if seen in isolation, be detrimental to the language in which they are suggested. That is, I see them as sacrifices necessary to achieve a greater good (C/C++ compatibility). I would never dream of suggesting each by itself and would fight many of the suggested resolutions except in the context of a major increase in C/C++ compatibility. A language is more than the sum of its individual features.

Part of a three-article series from "The C/C++ Users Journal" July, August, and September 2002

2 Varadic Function Syntax

In C, a variadic function is indicated by a comma followed by an ellipsis at the end of an argument list. In C++ the elipsis suffices. For example:

```
int printf(const char*, ...);   // C and C++
int printf(const char* ...);    // C++
```

The obvious resolution is for C to accept the plain ellipsis in addition to what is currently accepted. This resolution breaks no code, imposes no run-time overhead, and the additional compiler complexity is negligible. Any other resolution breaks lots of code without compensating advantages.

C requires a variadic function to have at least one argument specified; C++ doesn't require that. For example:

```
void f(...);    // C++, not C
```

This case could be dealt with either by allowing the construct in C or by disallowing it in C++. The first solution would break no user code, but could possibly cause problems for some C implementers. The latter could break some code. However, such code is likely to be rare and obscure, and there are obvious ways of rewriting it. Consequently, I suggest adopting the C rule and banning the construct in C++.

Breaking code should never be done lightly. However, sometimes it is better to break code than to let a problem fester. In making such a case, the likely importance of the construct banned should be taken into account, as should the likelyhood of code using the construct hiding errors. The probable benefits of the change have to be major. Whenever possible, meaning almost always, code broken by a language change should be easily detected by a compiler.

Breaking code isn't all bad. The odd easily diagnosable incompatibility that doesn't affect link compatibility, such as a the introduction of a new keyword, can be good for the long-term heath of the community. It reminds people that the world changes and gives encouragement to review old code. Compatibility switches are needed, though, to serve people who can't/won't touch old source code. I'm no fan of compiler options, but they are a fact of life and a compatibility switch providing practical backwards compatibility can be a price worth paying for progress.

3 Pointer to *void* and *NULL*

In C, a *void** can be assigned to any *T** without an explicit cast. In C++, it cannot. The reason for the C++ restriction is that a *void** to *T** conversion can be unsafe [Stroustrup,2002]. On the other hand, this implicit conversion is widely used in C. For example:

```
int* p = malloc(sizeof(int)*n);  // malloc()'s return type is void*
struct X* p = NULL;              // NULL is often a macro for (void*)0
```

From a C++ point of view, *malloc()* is itself best avoided in favor of *new*, but C's use of *(void*)0* provides the benefit of distinguishing a nil pointer from plain *0*. However, C++ retained the Classic C definition of *NULL* and maintained a tradition for using plain *0* rather than the macro *NULL*. Had assignment of *(void*)0* to a pointer been valid C++, it would have helped in overloading:

```
void f(int);
void f(char*);

void g()
{
    f(0);           // 0 is an int, call f(int)
    f((void*)0);    // error in C++, but why not call f(char*)?
}
```

What can be done to resolve this incompatibility:

[1] C++ accepts the C rule.

[2] C accepts the C++ rule.

[3] both languages ban the implicit conversion except for specific cases in the standard library, such as NULL and *malloc()*.

[4] C++ accepts the C rule for *void** and both languages introduce a new type, say *raw**, which

provides the safer C++ semantics.

I could construct several more scenarios, involving features such as a ***null*** keyword, a ***new*** operator for C, and macro magic. Such ideas may have value in their own right. However, among the alternatives listed, the right answer must be [1]. Resolution [2] breaks too much code, and [3] breaks too much code and is also messy. Note that I say this while insisting that much of the code that [3] would break deserves to be broken, and that ''a type violation'' is my primary criterion for deeming a C/C++ incompatibility ''non-gratuitous'' [Koenig,1989]. This is an example where I suggest that in the interest of the C/C++ community, we must leave our ''language theory ivory towers'', accept a wart, and get on with more important things. Alternative [4] allows programmers to preserve type safety in new code (and in code converted to use it), but don't think that benefit is sufficient to add a new feature.

In addition, I would seriously consider a variant of [1] that also introduced a keyword meaning ''the ***NULL*** pointer'' to save C++ programmers from unnecessarily depending on a macro (see [Stroustrup,2002a]).

4 *wchar_t* and *bool*

C introduced the typedef ***wchar_t*** for wide characters. C++ then adopted the idea, but needed a unique wide character type to guide overloading for proper stream I/O etc., so ***wchar_t*** was made a keyword.

C++ introduced a Boolean type named by the keyword ***bool***. C then introduced a macro ***bool*** (in a standard header) naming a keyword ***__Bool***. The C choices were made to increase C++ compatibility while avoiding breaking existing code using ***bool*** as an identifier.

For people using both languages, this is a mess (see the appendix of [Stroustrup,2002]). Again we can consider alternative resolutions:

[1] C adopts ***wchar_t*** and ***bool*** as keywords.

[2] C++ adopts C's definitions, and abolishes ***wchar_t*** and ***bool*** as keywords.

[3] C++ abolishes ***wchar_t*** and ***bool*** as keywords and adopts ***wchar_t*** and ***bool*** as typedefs, defined in some standard library header, for keywords ***__Wchar*** and ***__Bool***. C adopts ***__Wchar*** as the type for which ***wchar_t*** is a typedef.

[4] Both languages adopts a new mechanism, possibly called ***typename*** that is similar to ***typedef*** except that it makes new type rather than just a synonym. ***typename*** is then be used to provide ***bool*** and ***wchar_t*** in some standard header. The keywords ***bool***, ***wchar_t***, and ***__Bool*** would no longer be needed.

[5] C++ introduces ***wchar*** as a keyword, removes ***wchar_t*** as a keyword, and introduces ***wchar_t*** as a typedef for ***wchar*** in the appropriate standard header. C introduces ***bool*** and ***wchar*** as keywords.

Many C++ facilities depend on overloading, so C++ must have specific types, rather than just ***typedef***s. Therefore [2] is not a possible resolution. I consider [3] complicated (introducing two ''special words'' where one would do) and its compatibility advantages are illusory. If ***bool*** is a name in a standard header, all code had better avoid that word because there is no way of knowing whether that header might be used in the future, and any usage that differ from the standard will cause confusion and maintenance problems [Stroustrup,2002]. Suggestion [4] is an intriguing idea, but in this particular context, it shares the weaknesses of [3]. Solution [1] is the simplest, and is a distinct possibility. However, I think that having a keyword, ***wchar_t***, with a name that indicates that it is a typedef is also a mistake, so I suggest that [5] is the best solution.

One way of looking at an incompatibility is ''what could we have done then, had we known what we know now? What is the ideal solution?'' That was how I found [5]. Preserving ***wchar_t*** as a typedef is a simple backwards compatibility hack. In addition, either C must remove ***__Bool*** as a keyword, or C++ add it. The latter should be done because it is easy and breaks no code.

5 Empty Function Argument Specification

In C, the function prototype

```
int f();
```

declares a function ***f*** that may accept any number of arguments of any type (but ***f*** may not be variadic) as long as the arguments and types match the (unknown) definition of the function. In C++, the function declaration

```
int f();
```

declares *f* as a function that accepts no arguments. In both C and C++,

```
int f(void);
```

declares *f* as a function that accepts no arguments.
In addition, the C99 committee (following C89) deprecated

```
int f();
```

This *f*( ) incompatibility could be resolved cleanly by banning the C++ usage. However, that would break a majority of C++ programs ever written and force everyone to use the more verbose notation

```
int f(void);
```

which many consider an abomination [Stroustrup,2002].

The obvious alternative would be to break C code that relies on being able to call a function declared without arguments with arguments. For example:

```
int f();

int g(int a)
{
    return f(a);    // not C++, deprecated in C
}
```

I think that banning such calls is the right solution. It breaks C code, but the usage is most error-prone, has been deprecated in C since 1989, and have been caught by compiler warnings and lint for much longer. Thus,

```
int f();
```

should declare *f* to be a function taking no arguments. Again, a backwards compatibility switch might be useful.

6 Prototypes

In C++, no function can be called without a previous declaration. In C, a non-variadic function may be called without a previous prototype, but doing so has been deprecated since 1989. For example:

```
int f(i)
{
    int x = g(i);    // error in C++, deprecated in C
    // ...
}
```

All compilers and lints that I know of have mechanisms for detecting such usage.

The alternatives are clear:

[1] Allow such calls (as in C, but deprecated)

[2] Disallow calls to undeclared function (as in C++)

The resolution must be [2] to follow C++ and the intent of the C committee, as represented by the deprecation. Choosing [1] would seriously weaking the type checking in C++ and go against the general trend of programming without compensating benefits.

7 Old-style Function Definition

C supports the Classic C function declaration syntax; C++ does not. For example:

```
int f(a,b) double b;    /* not C++ */
{
    /* ... */
}
```

```
int g(char *p)
{
    f(p,p);    // uncaught type error
    // ...
}
```

The problem with the call of ***f*** arise because a function defined using the old-style function syntax has a different semantics from a function declared using the modern (prototype-style, C++-style) definition syntax. The function ***f*** is not considered prototyped so type checking and conversion is not done.

Resolving this cannot be done painlessly. I see three alternatives:

[1] adopt C's rules (i.e. allow old-style definitions with separate semantics)

[2] adopt C++'s rules (i.e. ban old-style definitions)

[3] allow old-style definitions with exactly the same semantics as other function definitions

[1] is not a possible solution because it eliminates important type checking and leads to surprises. I consider [2] the best solution, but see no hope for its acceptance. It has the virtues of simplicity, simple compile-time detection of all errors, and simple conversion to a more modern style. However, my impression is that old-style function definitions are still widely used and sometimes even liked for aesthetic reasons. That leaves [3], which has the virtues of simple implementation and better type checking, but suffers from the possibility of silent changes of the meaning of code. Warnings and a backwards compatibility switch would definitely be needed.

8 Enumerations

In C, an ***int*** can be assigned to a value of type an ***enum*** without a cast. In C++, it cannot. For example:

```
enum E { a, b };

E x = 1;   // error in C++, ok in C
E x = 99;  // error in C++, ok in C
```

I think that the only realistic resolution would be for C++ to adopt the C rule. The C++ rule provides better type safety, but the amount of C code relying on treating an enumerator as an ***int*** is too large to change, and I don't see a third alternative.

The definition of ***enum*** in C and C++ also differ in several details relating to size. However, the simple fact that C and C++ code today interoperate while using ***enum***s indicates that the definitional issues can be reconciled.

9 Constants

In C, the default storage class of a non-local ***const*** is extern and in C++ it is static. The result is one of the hardest-to-resolve incompatibilities. Consider:

```
const int x;              // uninitialized const
const int y = 2;

int f(int i)
{
    switch (i) {
    case y:               // use y as a constant expression
        return i;
    // ...
    }
}
```

An uninitialized ***const*** is not allowed in C++, and the use of a ***const*** in a constant expression is not allowed in C. Both of those uses are so widespread in their respective languages that examples such as the one above must be allowed. This precludes simply adopting the rule from one language or the other.

It follows that some subtlety is needed in the resolution, and subtlety implies the need for experimentation and examination of lots of existing code to see that undesired side effects really are absent. That said, here is a suggestion: Distinguish between initialized and uninitialized ***const***s . Initialized ***const***s follow the C++ rule. This preserves ***const***s in constant expressions, and if an initialized ***const*** needs to be accessed

from another translation unit, ***extern*** must be explicitly used:

```
const int c1 = 17;          // local to this translation unit
extern const int c2 = 7;    // available in other translation units
```

On the other hand, follow the C rule for uninitialized ***const***s. For example:

```
const int c3;          // available in other translation units
static const int c4;   // error: uninitialized const
```

10 Inlining

Both C and C99 provide ***inline***. Unfortunately, the semantics of ***inline*** differ [Stroustrup,2002]. Basically, the C++ rules require an inline function to be defined with identical meaning in every translation unit, even though an implementation is not required to detect violations of this ''one definition rule'', and many implementations can't. On the other hand, C99 allows inline functions in different translation units to differ, while imposing restrictions intented to avoid potential linker problems. A good resolution would

[1] not impose burdens on C linker technology
[2] not break the C++ type system
[3] break only minimal and pathological code
[4] not increase the area of undefined or implementation-specified behavior

An ideal solution would strengthen [3] and [4], but that's unfortunately impossible.

Requirement [2] basically implies that the C++ ODR must be the rule, even if its enforcement must – in the Classic C tradition [Stroustrup,2002a] – be left to a lint-like utility. This leaves the problem of what to do about uses of ***static*** data. For example:

```
// use of static variables in/from inlines ok in C++, errors in C:

static int a;
extern inline int count() { return ++a; }

extern inline int count2() { static int b = 0; b+=2; return b; }
```

Accepting such code would put a burden on C linkers; not accepting it would break C++ code. I think the most realistic choice is to ban such code, realizing that some implementations would accept it as an extension. The reason that I can envision banning such code is that I consider it rare and relatively unimportant. Naturally, we'd have to look at a lot of code before accepting that evaluation.

There is a more important use of static data in C++ that cannot be banned: static class members. However, since static class members have no equivalent in C, this is not a compatibility problem.

11 Static

C++ deprecates the use of ***static*** for declaring something ''local to this translation unit'' in favor of the more general notion of namspaces. The possible resolutions are

[1] withdraw that deprecation in C++
[2] deprecate or ban that use of ***static*** in C and introduce namespaces.

Only [1] is realistic.

12 Variable-Sized Data Structures

Classic C arrays are too low level for many uses: They have a fixed size, specified as a constant, and an array doesn't carry its size with it when passed as a function argument. Both C++ and C99 added features to deal with that issue:

[1] C++ added standard library containers. In particular, it added ***std::vector***. A ***vector*** can be specified by a size that is a variable, a ***vector*** can be resized, and a ***vector*** knows its size (that is, a ***vector*** can be passed as an object with it's size included, there is a member function for examining that size, and the size can be changed).
[2] C99 added Variable Length Arrays (VLAs). A VLA can be specified by a size that is a variable, but a VLA cannot be resized, and a VLA doesn't know its size.

The syntax of the two constructs differ and either could be argued to be more convenient and readable than

that the other:

```
void f(int m)
{
    int a[m];              // variable length array
    vector<int> v(m);      // standard library vector
    // ...
}
```

A VLA behaves much like a ***vector*** without the ability to resize. On the other hand, VLAs are designed with a heavier emphasis on run-time performance. In particular, elements of a VLA can be, but are not required to be, allocated on the stack.

For C/C++ compatibility, there are two obvious alternatives;

[1] Accept VLAs as defined in C99

[2] Ban VLAs.

Choosing [1] seems obvious. After all, VLAs are arguable the C99 committee's greatest contribution to C and the most significant language feature added to C since prototypes and ***const*** were imported from C with Classes. They are easy to implement, efficient, reasonably easy to use, and backwards compatible.

Unfortunately, from a C++ point of view, VLAs have several serious problems†:

[a] They are a very low-level mechanism, requiring programmers to remember sizes and pass them along. This is error-prone. This same lack of size information means that operations, such as copying, and range checking cannot be simply provided.

[b] A VLA can allocate an arbitrary amount of memory, specified at run time. However, there is no standard mechanism for detecting or handling memory exhaustion. This is particularly bothersome because a VLA looks so much like an ordinary array, for which the memory requirements can be calculated at compile time. For example:

```
#define M1 99

int f(int m2)
{
    int a[M1];      // array, space requirement known
    int b[m2];      // VLA, space requirement unknown
    // ...
}
```

This can lead to undefined behavior and obscure bugs.

[c] There is no guarantee that memory allocated for elements of a VLA are freed if a function containing it is exited abnormally (such as by an exception or a ***longjmp***). Thus, use of VLAs can lead to memory leaks.

[d] By using the array syntax, many programmers will see VLAs as ''favored by the language'' or ''recommended'' over alternatives, such as ***std::vector***, and as more efficient (even if only potentially so).

[e] VLAs are part of C, and ***std::vector*** is not, so if VLAs were accepted for the sake of C/C++ compatibility, people would accept the problems with VLAs and use them in the interest of maximal portability.

The net effect is that by accepting VLAs, the result would be a language that encouraged something that, from a C++ point of view, is unnecessarily low-level, unsafe, and can leak memory.

It follows that a third alternative is needed. Consider:

[3] Ban VLAs and replace it with ***vector*** (possibly provided as a built-in type).

[4] Define a VLA to be equivalent and interchangeable with a suitably designed container, ***array***.

Naturally, [3] is unacceptable because VLAs exist in the C standard, but it would have been a close-to-ideal

† It has been suggested that considering VLAs from a C++ point of view is unfair and disrespectful to the C committee because C is not C++ and C/C++ compatibility isn't part of the C standard committee's charter. I mean no disrespect to the C committee, its members, or to the ISO process that the C committee is part of. However, given that a large C/C++ community exist and that VLAs will inevitably be used together with C++ code (either through linkage or through permissive compiler switches), an analysis is unavoidable and needed.

solution. However, we can use the idea as a springboard to a more acceptable resolution. How would ***array*** have to be designed to bridge the gap between VLAs and C++ standard library containers? Consider possible implementations of VLAs. For C-only, a VLA and its size are needed together only at the point of allocation. If extended to support C++, destructors must be called for VLA elements, so the size must (conceptually, at least) be stored with a pointer to the elements. Therefore, a naive implementation of ***array*** would be something like this:

```
template<class T> class array {
    int s;
    T* elements;
public:
    array(int n);          // allocate "n" elements and let "elements" refer to them
    array(T* p, int n);    // make this array refer to p[0..n-1]
    operator T*() { return elements; }
    int size() const { return s; }

    // the usual container operations, such as = and [], much like vector
};
```

Apart from the two-argument constructor, this would simply be an ordinary container which could be designed to allocate from the stack, just like some VLA implementations. The key to compatibility is its integration with VLAs:

```
void h(array<double> a);        // C++

void g(int m, double vla[m]);       // C99

void f(int m, double vla1[m], array<double> a1)
{
    array<double>a2(vla1,m);        // a2 refers to vla1
    double* p = a1;                 // p refers to a1's elements

    h(a1);
    h(array(vla1,m));   // a bit verbose
    h(m,vla1);          // ???

    g(m,vla1);
    g(a1.size(),a1);    // a bit verbose
    g(a1);              // ???
}
```

The calls marked with `???` cannot be written in C++. Had they gotten past the type checking, the result would have executed correctly because of structural equivalence. If we somehow accept these calls, by a general mechanism or by a special rule for ***array*** and VLAs, ***array***s and VLAs would be completely interchangeable and a programmer could choose whichever style best suited taste and application.

Clearly, the ***array*** idea is not a complete proposal, but it shows a possible direction for coping with a particularly nasty problem of divergent language evolution.

13 Afterword

There are many more compatibility issues that must be dealt with by a thorough description of C/C++ incompatibilities and their possible resolution. However, the examples here should give a concrete basis for a debate both on principles and practical resolutions. Again, please note that the suggested resolutions don't make much sense in isolation, I see them as part of a comprehensive review to eliminate C/C++ incompatibilities.

I suspect that the main practical problem in eliminating the C/C++ incompatibilities, would not be one of the compatibility problems listed above. The main problem would be that starting from Dennis Ritchie's original text, the two standards have evolved independently using related but subtly different vocabularies, phrases, and styles. Reconciling those would take painstaking work of several experienced people for several months. The C++ standard is 720 pages, the C99 standard is 550 pages. I think the work would be worth it for the C/C++ community. The result would be a better language for all C and C++ programmers.

14 References

[C89] ISO/IEC 9899:1990, Programming Languages – C.

[C99] ISO/IEIC 9899:1999, Programming Languages – C.

[C++98] ISO/IEC 14882, Standard for the C++ Language.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. July 1989.

[Stroustrup,2002] Bjarne Stroustrup: *Sibling Rivalry: C and C++*. AT&T Labs - Research Technical Report TD-54MQZY, January 2002. http://www.research.att.com/~bs/sibling_rivalry.pdf.

[Stroustrup,2002a] Bjarne Stroustrup: *C and C++: Siblings*. The C/C++ Journal.

[Stroustrup,2002b] Bjarne Stroustrup: *C and C++: A Case for Compatibility*. The C/C++ Journal.

Exception Safety: Concepts and Techniques

Bjarne Stroustrup

AT&T Labs – Research
Florham Park, NJ 07932, USA
http://www.research.att.com/~bs

Abstract. This paper presents a set of concepts and design techniques that has proven successful in implementing and using C++ libraries intended for applications that simultaneously require high reliability and high performance. The notion of exception safety is based on the *basic guarantee* that maintains basic invariants and avoids resource leaks and the *strong guarantee* that ensures that a failed operation has no effect.

1 Introduction

This paper, based on *Appendix E: Standard-Library Exception Safety* of *The C++ Programming Language (Special Edition)* [1], presents

(1) a few fundamental concepts useful for discussion of exception safety
(2) effective techniques for crafting exception-safe and efficient containers
(3) some general rules for exception-safe programming.

The discussion of exception safety focuses on the containers provided as part of the ISO C++ standard library [2] [1]. Here, the standard library is used to provide examples of the kinds of concerns that must be addressed in demanding applications. The techniques used to provide exception safety for the standard library can be applied to a wide range of problems.

The discussion assumes an understanding of C++ and a basic understanding of C++'s exception handling mechanisms. These mechanism, the fundamental ways of using them, and the support they receive in the standard library are described in [1]. The reasoning behind the design of C++'s exception handling mechanisms and references to previous work influencing that design can be found in [3].

2 Exception Safety

An operation on an object is said to be *exception safe* if that operation leaves the object in a valid state when the operation is terminated by throwing an exception. This valid state could be an error state requiring cleanup, but it must be well defined so that reasonable error-handling code can be written for the object. For example, an exception handler might destroy the object, repair the object, repeat a variant of the operation, just carry on, etc.

In other words, the object will have an invariant, its constructors establish that invariant, all further operations maintain that invariant, and its destructor does a final cleanup. An operation should take care that the invariant is maintained before throwing an exception, so that the object is in a valid state.

However, it is quite possible for that valid state to be one that doesn't suit the application. For example, a string may have been left as the empty string or a container may have been left unsorted. Thus, ''repair'' means giving an object a value that is more appropriate/desirable for the application than the one it was left with after an operation failed. In the context of the standard library, the most interesting objects are containers.

Here, we consider under which conditions an operation on a standard-library container can be considered exception safe. There can be only two conceptually really simple strategies:

(1) ''*No guarantees*:'' If an exception is thrown, any container being manipulated is possibly corrupted.

(2) ''*Strong guarantee*:'' If an exception is thrown, any container being manipulated remains in the state in which it was before the standard-library operation started.

Unfortunately, both answers are too simple for real use. Alternative (1) is unacceptable because it implies that after an exception is thrown from a container operation, the container cannot be accessed; it can't even be destroyed without fear of run-time errors. Alternative (2) is unacceptable because it imposes the cost of roll-back semantics on every individual standard-library operation.

To resolve this dilemma, the C++ standard library provides a set of exception-safety guarantees that share the burden of producing correct programs between implementers of the standard library and users of the standard library:

(3a) ''*Basic guarantee* for all operations:'' The basic invariants of the standard library are maintained, and no resources, such as memory, are leaked.

(3b) ''*Strong guarantee* for key operations:'' In addition to providing the basic guarantee, either the operation succeeds, or has no effects. This guarantee is provided for key library operations, such as ***push_back***`()`, single-element ***insert***`()` on a ***list***, and ***uninitialized_copy***`()`.

(3c) ''*Nothrow guarantee* for some operations:'' In addition to providing the basic guarantee, some operations are guaranteed not to throw an exception This guarantee is provided for a few simple operations, such as ***swap***`()` and ***pop_back***`()`.

Both the basic guarantee and the strong guarantee are provided on the condition that user-supplied operations (such as assignments and ***swap***`()` functions) do not leave container elements in invalid states, that user-supplied operations do not leak resources, and that destructors do not throw exceptions.

Violating a standard library requirement, such as having a destructor exit by throwing an exception, is logically equivalent to violating a fundamental language rule, such a dereferencing a null pointer. The practical effects are also equivalent and often disastrous.

In addition to achieving pure exception safety, both the basic guarantee and the strong guarantee ensure the absence of resource leaks. That is, a standard library operation that throws an exception not only leaves its operands in well-defined states but also ensures that every resource that it acquired is (eventually) released. For example, at the point where an exception is thrown, all memory allocated must be either

deallocated or owned by some object, which in turn must ensure that the memory is properly deallocated. Remember that memory isn't the only kind of resource that can leak. Files, locks, network connections, and threads are examples of system resources that a function may have to release or hand over to an object before throwing an exception.

Note that the C++ language rules for partial construction and destruction ensure that exceptions thrown while constructing sub-objects and members will be handled correctly without special attention from standard-library code. This rule is an essential underpinning for all techniques dealing with exceptions.

3 Exception-Safe Implementation Techniques

The C++ standard library provides examples of problems that occur in many other contexts and of solutions that apply widely. The basic tools available for writing exception-safe code are

(1) the *try-block*, and

(2) the support for the ''resource acquisition is initialization'' technique.

The key idea behind the ''resource acquisition is initialization'' technique/pattern (sometimes abbreviated to RAII) is that ownership of a resource is given to a scoped object. Typically, that object acquires (opens, allocates, etc.) the resource in its constructor. That way, the objects destructor can release the resource at the end of its life independently of whether that destruction is caused by normal exit from its scope or from an exception. For details, see Sect. 14.4 of [1]. Also, the use of ***vector_base*** from Sect 3.2 of this paper is an example of ''resource acquisition is initialization.''

The general principles to follow are to

(1) don't destroy a piece of information before we can store its replacement

(2) always leave objects in valid states when throwing or re-throwing an exception

(3) avoid resource leaks.

That way, we can always back out of an error situation. The practical difficulty in following these principles is that innocent-looking operations (such as <, =, and ***sort***( )) might throw exceptions. Knowing what to look for in an application takes experience.

When you are writing a library, the ideal is to aim at the strong exception-safety guarantee and always to provide the basic guarantee. When writing a specific program, there may be less concern for exception safety. For example, if I write a simple data analysis program for my own use, I'm usually quite willing to have the program terminate in the unlikely event of virtual memory exhaustion. However, correctness and exception safety are closely related.

The techniques for providing basic exception safety, such as defining and checking invariants, are similar to the techniques that are useful to get a program small and correct. It follows that the overhead of providing basic exception safety (the basic guarantee) – or even the strong guarantee – can be minimal or even insignificant.

Here, I will consider an implementation of the standard container ***vector*** to see what it takes to achieve that ideal and where we might prefer to settle for more conditional safety.

3.1 A Simple Vector

A typical implementation of ***vector*** will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (or the equivalent information represented as a pointer plus offsets):

Here is a declaration of ***vector*** simplified to present only what is needed to discuss exception safety and avoidance of resource leaks:

```
template<class T, class A = allocator<T> > class vector {
public:
    T* v;        // start of allocation
    T* space;    // end of elements, start of space allocated for possible expansion
    T* last;     // end of allocated space
    A alloc;     // allocator

    explicit vector(size_type n, const T& val = T(), const A& = A());
    vector(const vector& a);                    // copy constructor
    vector& operator=(const vector& a);         // copy assignment
    ~vector();                                  // destructor

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);

    // ...
};
```

Consider first a naive implementation of a constructor:

```
template<class T, class A>              // warning: naive implementation
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :alloc(a)                           // copy the allocator
{
    v = alloc.allocate(n);              // get memory for elements
    space = last = v+n;
    for (T* p=v; p!=last; ++p) a.construct(p,val); // construct copy of val in *p
}
```

There are three sources of exceptions here:

(1) ***allocate***() throws an exception indicating that no memory is available;
(2) the allocator's copy constructor throws an exception;
(3) the copy constructor for the element type ***T*** throws an exception because it can't copy ***val***.

In all cases, no object is created. However, unless we are careful, resources can leak.

When ***allocate***() fails, the ***throw*** will exit before any resources are acquired, so all is well. When ***T***'s copy constructor fails, we have acquired some memory that must be

freed to avoid memory leaks. A more difficult problem is that the copy constructor for ***T*** might throw an exception after correctly constructing a few elements but before constructing them all. To handle this problem, we could keep track of which elements have been constructed and destroy those (and only those) in case of an error:

```
template<class T, class A>                // elaborate implementation
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :alloc(a)                             // copy the allocator
{
    v = alloc.allocate(n);                // get memory for elements

    iterator p;

    try {
        iterator end = v+n;
        for (p=v; p!=end; ++p) alloc.construct(p,val); // construct element
        last = space = p;
    }
    catch (...) {   // destroy constructed elements, free memory, and re-throw:
        for (iterator q = v; q!=p; ++q) alloc.destroy(q);
        alloc.deallocate(v,n);
        throw;
    }
}
```

The overhead here is the overhead of the *try-block*. In a good C++ implementation, this overhead is negligible compared to the cost of allocating memory and initializing elements. For implementations where entering a *try-block* incurs a cost, it may be worthwhile to test ***if***(***n***) before the ***try*** and handle the empty vector case separately.

The main part of this constructor is an exception-safe implementation of the standard library's ***uninitialized_fill***():

```
template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p=beg; p!=end; ++p)
            new(static_cast<void*>(&*p)) T(x); // construct copy of x in *p
    }
    catch (...) {   // destroy constructed elements and rethrow:
        for (For q = beg; q!=p; ++q) (&*q)->~T();
        throw;
    }
}
```

The curious construct &****p*** takes care of iterators that are not pointers. In that case, we need to take the address of the element obtained by dereference to get a pointer. The explicit cast to ***void***** ensures that the standard library placement function is used, and not some user-defined ***operator new***() for ***T*****s. This code is operating at a rather low level where writing truly general code can be difficult.

Fortunately, we don't have to reimplement ***uninitialized_fill***(), because the

standard library provides the desired strong guarantee for it. It is often essential for initialization to either complete successfully, having initialized every element, or fail leaving no constructed elements behind. Consequently, the standard-library algorithms ***uninitialized_fill*()**, ***uninitialized_fill_n*()**, and ***uninitialized_copy*()** are guaranteed to have this strong exception-safety property.

Note that ***uninitialized_fill*()** does not protect against exceptions thrown by element destructors or iterator operations. Doing so would be prohibitively expensive.

The ***uninitialized_fill*()** algorithm can be applied to many kinds of sequences. Consequently, it takes a forward iterator and cannot guarantee to destroy elements in the reverse order of their construction.

Using ***uninitialized_fill*()**, we can write:

```
template<class T, class A>              // messy implementation
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :alloc(a)                           // copy the allocator
{
    v = alloc.allocate(n);              // get memory for elements
    try {
        uninitialized_fill(v,v+n,val);  // copy elements
        space = last = v+n;
    }
    catch (...) {
        alloc.deallocate(v,n);          // free memory
        throw;                          // re-throw
    }
}
```

However, I wouldn't call that pretty code. The next section will demonstrate how it can be made much simpler.

Note that the constructor re-throws a caught exception. The intent is to make ***vector*** transparent to exceptions so that the user can determine the exact cause of a problem. All standard-library containers have this property. Exception transparency is often the best policy for templates and other ''thin'' layers of software. This is in contrast to major parts of a system (''modules'') that generally need to take responsibility for all exceptions thrown. That is, the implementer of such a module must be able to list every exception that the module can throw. Achieving this may involve grouping exceptions, mapping exceptions from lower-level routines into the module's own exceptions, or exception specification.

3.2 Representing Memory Explicitly

Experience revealed that writing correct exception-safe code using explicit *try-block*s is more difficult than most people expect. In fact, it is unnecessarily difficult because there is an alternative: The ''resource acquisition is initialization'' technique can be used to reduce the amount of code written and to make the code more stylized. In this case, the key resource required by the ***vector*** is memory to hold its elements. By providing an auxiliary class to represent the notion of memory used by a ***vector***, we can simplify the code and decrease the chance of accidentally forgetting to release it:

```
template<class T, class A = allocator<T> >
struct vector_base {
    A alloc;    // allocator
    T* v;       // start of allocation
    T* space;   // end of elements, start of space allocated for possible expansion
    T* last;    // end of allocated space

    vector_base(const A& a, typename A::size_type n)
        : alloc(a), v(a.allocate(n)), space(v+n), last(v+n) { }
    ~vector_base() { alloc.deallocate(v,last-v); }
};
```

As long as *v* and *last* are correct, *vector_base* can be destroyed. Class *vector_base* deals with memory for a type *T*, not objects of type *T*. Consequently, a user of *vector_base* must destroy all constructed objects in a *vector_base* before the *vector_base* itself is destroyed.

Naturally, *vector_base* itself is written so that if an exception is thrown (by the allocator's copy constructor or *allocate*() function) no *vector_base* object is created and no memory is leaked.

We want to be able to *swap*() *vector_base*s. However, the default *swap*() doesn't suit our needs because it copies and destroys a temporary. Because *vector_base* is a special-purpose class that wasn't given fool-proof copy semantics, that destruction would lead to undesirable side effects. Consequently we provide a specialization:

```
template<class T> void swap(vector_base<T>& a, vector_base<T>& b)
{
    swap(a.a,b.a);
    swap(a.v,b.v);
    swap(a.space,b.space);
    swap(a.last,b.last);
}
```

Given *vector_base*, *vector* can be defined like this:

```
template<class T, class A = allocator<T> >
class vector : private vector_base<T,A> {
    void destroy_elements() { for (T* p = v; p!=space; ++p) p->~T(); }
public:
    explicit vector(size_type n, const T& val = T(), const A& = A());
    vector(const vector& a);                // copy constructor
    vector& operator=(const vector& a);     // copy assignment
    ~vector() { destroy_elements(); }

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);
    // ...
};
```

The *vector* destructor explicitly invokes the *T* destructor for every element. This implies that if an element destructor throws an exception, the *vector* destruction fails.

This can be a disaster if it happens during stack unwinding caused by an exception and *terminate*() is called. In the case of normal destruction, throwing an exception from a destructor typically leads to resource leaks and unpredictable behavior of code relying on reasonable behavior of objects. There is no really good way to protect against exceptions thrown from destructors, so the library makes no guarantees if an element destructor throws.

Now the constructor can be simply defined:

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
        :vector_base<T,A>(a,n)                  // allocate space for n elements
{
        uninitialized_fill(v,v+n,val);      // copy elements
}
```

The copy constructor differs by using *uninitialized_copy*() instead of *uninitialized_fill*():

```
template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
        :vector_base<T,A>(a.alloc,a.size())
{
        uninitialized_copy(a.begin(),a.end(),v);
}
```

Note that this style of constructor relies on the fundamental language rule that when an exception is thrown from a constructor, sub-objects (such as bases) that have already been completely constructed will be properly destroyed. The *uninitialized_fill*() algorithm and its cousins provide the equivalent guarantee for partially constructed sequences.

3.3 Assignment

As usual, assignment differs from construction in that an old value must be taken care of. Consider a straightforward implementation:

```
template<class T, class A>              // offers the strong guarantee
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
        vector_base<T,A> b(alloc,a.size());             // get memory
        uninitialized_copy(a.begin(),a.end(),b.v);      // copy elements
        destroy_elements();                             // destroy old elements
        alloc.deallocate(v,last-v);                     // free old memory
        vector_base::operator=(b);                      // install new representation
        b.v = 0;                                        // prevent deallocation
        return *this;
}
```

This assignment is nice and exception safe. However, it repeats a lot of code from constructors and destructors. Also, the ‘‘installation’’ of the new *vector_base* is a bit obscure. To avoid this, we can write:

```
template<class T, class A>              // offers the strong guarantee
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
    vector temp(a);                                 // copy a
    swap< vector_base<T,A> >(*this,temp);           // swap representations
    return *this;
}
```

The old elements are destroyed by ***temp***'s destructor, and the memory used to hold them is deallocated by ***temp***'s ***vector_base***'s destructor.

The performance of the two versions ought to be equivalent. Essentially, they are just two different ways of specifying the same set of operations. However, the second implementation is shorter and doesn't replicate code from related ***vector*** functions, so writing the assignment that way ought to be less error prone and lead to simpler maintenance.

Note the absence of the traditional test for self-assignment:

```
if (this == &a) return *this;
```

These assignment implementations work by first constructing a copy and then swapping representations. This obviously handles self-assignment correctly. I decided that the efficiency gained from the test in the rare case of self-assignment was more than offset by its cost in the common case where a different ***vector*** is assigned.

In either case, two potentially significant optimizations are missing:

(1) If the capacity of the vector assigned to is large enough to hold the assigned vector, we don't need to allocate new memory.

(2) An element assignment may be more efficient than an element destruction followed by an element construction.

Implementing these optimizations, we get:

```
template<class T, class A>              // optimized, basic guarantee
vector<T,A>& vector<T,A>::operator=(const vector& a)
{
    if (capacity() < a.size()) {        // allocate new vector representation:
        vector temp(a);                                 // copy a
        swap< vector_base<T,A> >(*this,temp);           // swap representations
        return *this;
    }

    if (this == &a) return *this;       // protect against self assignment

                                        // assign to old elements:
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator();                  // copy the allocator

    if (asz<=sz) {   // copy over old elements and destroy surplus elements:
        copy(a.begin(),a.begin()+asz,v);
        for (T* p = v+asz; p!=space; ++p) p->~T();
    }
```

```
        else {      // copy over old elements and construct additional elements:
            copy(a.begin(),a.begin()+sz,v);
            uninitialized_copy(a.begin()+sz,a.end(),space);
        }
        space = v+asz;
        return *this;
    }
```

These optimizations are not free. The *copy*() algorithm does *not* offer the strong exception-safety guarantee. It does not guarantee that it will leave its target unchanged if an exception is thrown during copying. Thus, if *T*::*operator*=() throws an exception during *copy*(), the *vector* being assigned to need not be a copy of the vector being assigned, and it need not be unchanged. For example, the first five elements might be copies of elements of the assigned vector and the rest unchanged. It is also plausible that an element – the element that was being copied when *T*::*operator*=() threw an exception – ends up with a value that is neither the old value nor a copy of the corresponding element in the vector being assigned. However, if *T*::*operator*=() leaves its operands in a valid state if it throws an exception, the *vector* is still in a valid state – even if it wasn't the state we would have preferred.

Here, I have copied the allocator using an assignment. It is actually not required that every allocator support assignment.

The standard-library *vector* assignment offers the weaker exception-safety property of this last implementation – and its potential performance advantages. That is, *vector* assignment provides the basic guarantee, so it meets most people's idea of exception safety. However, it does not provide the strong guarantee. If you need an assignment that leaves the *vector* unchanged if an exception is thrown, you must either use a library implementation that provides the strong guarantee or provide your own assignment operation. For example:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // "obvious" a = b
{
    vector<T,A> temp(a.get_allocator());
    temp.reserve(b.size());
    for (typename vector<T,A>::iterator p = b.begin(); p!=b.end(); ++p)
        temp.push_back(*p);
    swap(a,temp);
}
```

If there is insufficient memory for *temp* to be created with room for *b*.*size*() elements, *std*::*bad_alloc* is thrown before any changes are made to *a*. Similarly, if *push_back*() fails for any reason, *a* will remain untouched because we apply *push_back*() to *temp* rather than to *a*. In that case, any elements of *temp* created by *push_back*() will be destroyed before the exception that caused the failure is rethrown.

Swap does not copy *vector* elements. It simply swaps the data members of a *vector*; that is, it swaps *vector_base*s (Sect. 3.2). Consequently, it does not throw exceptions even if operations on the elements might. Consequently, *safe_assign*() does not do spurious copies of elements and is reasonably efficient.

As is often the case, there are alternatives to the obvious implementation. We can let the library perform the copy into the temporary for us:

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // simple a = b
{
    vector<T,A> temp(b);      // copy the elements of b into a temporary
    swap(a,temp);
}
```

Indeed, we could simply use call-by-value:

```
template<class T, class A>            // simple a = b (note: b is passed by value)
void safe_assign(vector<T,A>& a, vector<T,A> b)
{
    swap(a,b);
}
```

The last two variants of ***safe_assign***() don't copy the ***vector***'s allocator. This is a permitted optimization.

3.4 *push_back()*

From an exception-safety point of view, ***push_back***() is similar to assignment in that we must take care that the ***vector*** remains unchanged if we fail to add a new element:

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
    if (space == last) {   // no more free space; relocate:
        vector_base b(alloc,size()?2*size():2); // double the allocation
        uninitialized_copy(v,space,b.v);
        new(b.space) T(x);                      // place a copy of x in *b.space
        ++b.space;
        destroy_elements();
        swap<vector_base<T,A> >(b,*this);       // swap representations
        return;
    }
    new(space) T(x);                            // place a copy of x in *space
    ++space;
}
```

Naturally, the copy constructor initializing ***space** might throw an exception. If that happens, the value of the ***vector*** remains unchanged, with ***space*** left unincremented. In that case, the ***vector*** elements are not reallocated so that iterators referring to them are not invalidated. Thus, this implementation implements the strong guarantee that an exception thrown by an allocator or even a user-supplied copy constructor leaves the ***vector*** unchanged. The standard library offers the strong guarantee for ***push_back***().

Note the absence of a *try-block* (except for the one hidden in ***uninitialized_copy***()). The update was done by carefully ordering the operations so that if an exception is thrown, the ***vector*** remains unchanged.

The approach of gaining exception safety through ordering and the ''resource

acquisition is initialization'' technique tends to be more elegant and more efficient than explicitly handling errors using *try-block*s. More problems with exception safety arise from a programmer ordering code in unfortunate ways than from lack of specific exception-handling code. The basic rule of ordering is not to destroy information before its replacement has been constructed and can be assigned without the possibility of an exception.

Exceptions introduce possibilities for surprises in the form of unexpected control flows. For a piece of code with a simple local control flow, such as the ***operator***=(), ***safe_assign***(), and ***push_back***() examples, the opportunities for surprises are limited. It is relatively simple to look at such code and ask oneself ''can this line of code throw an exception, and what happens if it does?'' For large functions with complicated control structures, such as complicated conditional statements and nested loops, this can be hard. Adding *try-block*s increases this local control structure complexity and can therefore be a source of confusion and errors. I conjecture that the effectiveness of the ordering approach and the ''resource acquisition is initialization'' approach compared to more extensive use of *try-block*s stems from the simplification of the local control flow. Simple, stylized code is easier to understand and easier to get right.

Note that the ***vector*** implementation is presented as an example of the problems that exceptions can pose and of techniques for addressing those problems. The standard does not require an implementation to be exactly like the one presented here. What the standard does guarantee is described in Sect. E.4 of [1].

3.5 Constructors and Invariants

From the point of view of exception safety, other ***vector*** operations are either equivalent to the ones already examined (because they acquire and release resources in similar ways) or trivial (because they don't perform operations that require cleverness to maintain valid states). However, for most classes, such ''trivial'' functions constitute the majority of code. The difficulty of writing such functions depends critically on the environment that a constructor established for them to operate in. Said differently, the complexity of ''ordinary member functions'' depends critically on choosing a good class invariant. By examining the ''trivial'' ***vector*** functions, it is possible to gain insight into the interesting question of what makes a good invariant for a class and how constructors should be written to establish such invariants.

Operations such as ***vector*** subscripting are easy to write because they can rely on the invariant established by the constructors and maintained by all functions that acquire or release resources. In particular, a subscript operator can rely on ***v*** referring to an array of elements:

```
template< class T, class A> T& vector<T,A>::operator[](size_type i)
{
    return v[i];
}
```

It is important and fundamental to have constructors acquire resources and establish a simple invariant. To see why, consider an alternative definition of ***vector_base***:

```
template<class T, class A = allocator<T> >        // clumsy use of constructor
class vector_base {
public:
    A alloc;    // allocator
    T* v;       // start of allocation
    T* space;   // end of elements, start of space allocated for possible expansion
    T* last;    // end of allocated space

    vector_base(const A& a, typename A::size_type n)
        : alloc(a), v(0), space(0), last(0)
    {
        v = alloc.allocate(n);
        space = last = v+n;
    }

    ~vector_base() { if (v) alloc.deallocate(v,last-v); }
};
```

Here, I construct a *vector_base* in two stages: First, I establish a ''safe state'' where *v*, *space*, and *last* are set to *0*. Only after that has been done do I try to allocate memory. This is done out of misplaced fear that if an exception happens during element allocation, a partially constructed object could be left behind. This fear is misplaced because a partially constructed object cannot be ''left behind'' and later accessed. The rules for static objects, automatic objects, member objects, and elements of the standard-library containers prevent that. However, it could/can happen in pre-standard libraries that used/use placement new to construct objects in containers designed without concern for exception safety. Old habits can be hard to break.

Note that this attempt to write safer code complicates the invariant for the class: It is no longer guaranteed that *v* points to allocated memory. Now *v* might be *0*. This has one immediate cost. The standard-library requirements for allocators do not guarantee that we can safely deallocate a pointer with the value *0*. In this, allocators differ from *delete*. Consequently, I had to add a test in the destructor.

This two-stage construct is not an uncommon style. Sometimes, it is even made explicit by having the constructor do only some ''simple and safe'' initialization to put the object into a destructible state. The real construction is left to an *init*() function that the user must explicitly call. For example:

```
template<class T>     // archaic (pre-standard, pre-exception) style
class Vector {
    T* v;       // start of allocation
    T* space;   // end of elements, start of space allocated for possible expansion
    T* last;    // end of allocated space

public:
    Vector() : v(0), space(0), last(0) { }
    ~Vector() { free(v); }

    bool init(size_t n);   // return true if initialization succeeded

    // ... Vector operations ...
};
```

```
template<class T>
bool Vector<T>::init(size_t n) // return true if initialization succeeded
{
    if (v = (T*)malloc(sizeof(T)*n)) {
        uninitialized_fill(v, v+n, T());
        space = last = v+n;
        return true;
    }
    return false;
}
```

The perceived value of this style is

(1) The constructor can't throw an exception, and the success of an initialization using ***init***() can be tested by ''usual'' (that is, non-exception) means.

(2) There exists a trivial valid state. In case of a serious problem, an operation can give an object that state.

(3) The acquisition of resources is delayed until a fully initialized object is needed.

However, this two-stage construction technique doesn't deliver its expected benefits and can itself be a source of problems.

The first point (using an ***init***() function in preference to a constructor) is bogus. Using constructors and exception handling is a more general and systematic way of dealing with resource acquisition and initialization errors. This style is a relic of pre-exception C++. Having a separate ***init***() function is an opportunity to

(1) forget to call ***init***(),

(2) call ***init***() more than once,

(3) forget to test on the success of ***init***(),

(4) forget that ***init***() might throw an exception, and

(5) use the object before calling ***init***().

Constructors and exceptions were introduced into C++ to prevent such problems [3].

The second point (having an easy-to-construct ''safe'' valid state) is in principle a good one. If we can't put an object into a valid state without fear of throwing an exception before completing that operation, we do indeed have a problem. However, this ''safe state'' should be one that is a natural part of the semantics of the class rather than an implementation artifact that complicates the class invariant.

If the ''safe'' state is not a natural part of the semantics of the class, the invariant is complicated and a burden is imposed on every member function. For example, the simple subscript operation becomes something like:

```
template< class T> T& Vector<T>::operator[](size_type i)
{
    if (v) return v[i];
    // error handling
}
```

If part of the reason for using a two-stage initialization was to avoid exceptions, the error handling part of that ***operator***[]() could easily become complicated.

Like the second point, the third (delaying acquisition of a resource until is needed) misapplies a good idea in a way that imposes cost without yielding benefits. In many cases, notably in containers such as ***vector***, the best way of delaying resource

acquisition is for the programmer to delay the creation of objects until they are needed.

To sum up: the two-phase construction approach leads to more complicated invariants and typically to less elegant, more error-prone, and harder-to-maintain code. Consequently, the language-supported ''constructor approach'' should be preferred to the ''*init*()-function approach'' whenever feasible. That is, resources should be acquired in constructors whenever delayed resource acquisition isn't mandated by the inherent semantics of a class.

The negative effects of two-phase construction become more marked when we consider application classes that acquire significant resources, such as network connections and files. Such classes are rarely part of a framework that guides their use and their implementation in the way the standard-library requirements guide the definition and use of *vector*. The problems also tend to increase as the mapping between the application concepts and the resources required to implement them becomes more complex. Few classes map as directly onto system resources as does *vector*.

4 Implications for Library Users

One way to look at exception safety in the context of the standard library is that we have no problems unless we create them for ourselves: The library will function correctly as long as user-supplied operations meet the standard library's basic requirements. In particular, no exception thrown by a standard container operation will cause memory leaks from containers or leave a container in an invalid state. Thus, the problem for the library user becomes: How can I define my types so that they don't cause undefined behavior or leak resources?

The basic rules are:

(1) When updating an object, don't destroy its old representation before a new representation is completely constructed and can replace the old one without risk of exceptions. For example, see the implementations of *safe_assign*() and *vector*::*push_back*() in Sect. 3.

(1a) If you must override an old representation in the process of creating the new, be sure to leave a valid object behind if an exception is thrown. For example, see the ''optimized'' implementation of *vector*::*operator*=().

(2) Before throwing an exception, release every resource acquired that is not owned by some (other) object.

(2a) The ''resource acquisition is initialization'' technique and the language rule that partially constructed objects are destroyed to the extent that they were constructed can be most helpful here.

(2b) The *uninitialized_copy*() algorithm and its cousins provide automatic release of resources in case of failure to complete construction of a set of objects.

(3) Before throwing an exception, make sure that every operand is in a valid state. That is, leave each object in a state that allows it to be accessed and destroyed without causing undefined behavior or an exception to be thrown from a destructor. For example, see *vector*'s assignment in Sect. 3.2.

(3a) Note that constructors are special in that when an exception is thrown from a constructor, no object is left behind to be destroyed later. This implies

that we don't have to establish an invariant and that we must be sure to release all resources acquired during a failed construction before throwing an exception.

(3b) Note that destructors are special in that an exception thrown from a destructor almost certainly leads to violation of invariants and/or calls to ***terminate*** ( ) .

In practice, it can be surprisingly difficult to follow these rules. The primary reason is that exceptions can be thrown from places where people don't expect them. A good example is ***std*::*bad_alloc***. Every function that directly or indirectly uses ***new*** or an ***allocator*** to acquire memory can throw ***bad_alloc***. In some programs, we can solve this particular problem by not running out of memory. However, for programs that are meant to run for a long time or to accept arbitrary amounts of input, we must expect to handle various failures to acquire resources. Thus, we must assume every function capable of throwing an exception until we have proved otherwise.

One simple way to try to avoid surprises is to use containers of elements that do not throw exceptions (such as containers of pointers and containers of simple concrete types) or linked containers (such as ***list***) that provide the strong guarantee. Another, complementary, approach is to rely primarily on operations, such as ***push_back*** ( ) , that offer the strong guarantee that an operation either succeeds or has no effect. However, these approaches are by themselves insufficient to avoid resource leaks and can lead to an ad hoc, overly restrictive, and pessimistic approach to error handling and recovery. For example, a ***vector<T*>*** is trivially exception safe if operations on ***T*** don't throw exceptions. However, unless the objects pointed to are deleted somewhere, an exception from the ***vector*** will lead to a resource leak. Thus, introducing a ***Handle*** class to deal with deallocation and using ***vector***<Handle<T> > rather than the plain ***vector<T*>*** will probably improve the resilience of the code.

When writing new code, it is possible to take a more systematic approach and make sure that every resource is represented by a class with an invariant that provides the basic guarantee. Given that, it becomes feasible to identify the critical objects in an application and provide roll-back semantics (that is, the strong guarantee – possibly under some specific conditions) for operations on such objects.

As mentioned in Sect. 3, the basic techniques for dealing with exceptions, focusing on resources and invariants, also help getting code correct and efficient. In general, keeping code stylish and simple by using classes to represent resources and concepts makes the code easier to understand, easier to maintain, and easier to reason about. Constructors, destructors, and the support for correct partial construction and destruction are the language-level keys to this. ''Resource acquisition is initialization'' is the key programming technique to utilize these language features.

Most applications contain data structures and code that are not written with exception safety in mind. Where necessary, such code can be fitted into an exception-safe framework by either verifying that it doesn't throw exception (as was the case for the C standard library) or through the use of interface classes for which the exception behavior and resource management can be precisely specified.

When designing types intended for use in an exception-safe environment, we must pay special attention to the operations used by the standard library: constructors,

destructors, assignments, comparisons, swap functions, functions used as predicates, and operations on iterators. This is best done by defining a class invariant that can be simply established by all constructors. Sometimes, we must design our class invariants so that we can put an object into a state where it can be destroyed even when an operation suffers a failure at an ''inconvenient'' point. Ideally, that state isn't an artifact defined simply to aid exception handling, but a state that follows naturally from the semantics of the type.

When considering exception safety, the emphasis should be on defining valid states for objects (invariants) and on proper release of resources. It is therefore important to represent resources directly as classes. The ***vector_base*** (Sect. 3.2) is a simple example of this. The constructors for such resource classes acquire lower-level resources (such as the raw memory for ***vector_base***) and establish invariants (such as the proper initialization of the pointers of a ***vector_base***). The destructors of such classes implicitly free lower-level resources. The rules for partial construction and the ''resource acquisition is initialization'' technique support this way of handling resources.

A well-written constructor establishes the class invariant for an object. That is, the constructor gives the object a value that allows subsequent operations to be written simply and to complete successfully. This implies that a constructor often needs to acquire resources. If that cannot be done, the constructor can throw an exception so that we can deal with that problem before an object is created. This approach is directly supported by the language and the standard library.

The requirement to release resources and to place operands in valid states before throwing an exception means that the burden of exception handling is shared among the function throwing, the functions on the call chain to the handler, and the handler. Throwing an exception does not make handling an error ''somebody else's problem.'' It is the obligation of functions throwing or passing along an exception to release resources that they own and to put operands in consistent states. Unless they do that, an exception handler can do little more than try to terminate gracefully.

5 Acknowledgements

The concepts and techniques described here are the work of many individuals. In particular, Dave Abrahams, Matt Austern, and Greg Colvin made major contributions to the notions of exception safety embodied in the C++ standard library.

6 References

[1] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[2] Andrew Koenig (editor): *Standard – The C++ Language*. ISO/IEC 14882:1998(E). Information Technology Council (NCITS). Washington, DC, USA. http://www.ncits.org/cplusplus.htm.

[3] Bjarne Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. 2000. ISBN 0-201-70073-5.

Evolving a language in and for the real world: C++ 1991-2006

Bjarne Stroustrup
Texas A&M University
www.research.att.com/~bs

Abstract

This paper outlines the history of the C++ programming language from the early days of its ISO standardization (1991), through the 1998 ISO standard, to the later stages of the C++0x revision of that standard (2006). The emphasis is on the ideals, constraints, programming techniques, and people that shaped the language, rather than the minutiae of language features. Among the major themes are the emergence of generic programming and the STL (the C++ standard library's algorithms and containers). Specific topics include separate compilation of templates, exception handling, and support for embedded systems programming. During most of the period covered here, C++ was a mature language with millions of users. Consequently, this paper discusses various uses of C++ and the technical and commercial pressures that provided the background for its continuing evolution.

Categories and Subject Descriptors K.2 [*History of Computing*]: systems

General Terms Design, Programming Language, History

Keywords C++, language use, evolution, libraries, standardization, ISO, STL, multi-paradigm programming

1. Introduction

In October 1991, the estimated number of C++ users was 400,000 [121]. The corresponding number in October 2004 was 3,270,000 [61]. Somewhere in the early '90s C++ left its initial decade of exponential growth and settled into a decade of steady growth. The key efforts over that time were to

1. use the language (obviously)
2. provide better compilers, tools, and libraries
3. keep the language from fragmenting into dialects
4. keep the language and its community from stagnating

[Copyright notice will appear here once 'preprint' option is removed.]

Obviously, the C++ community spent the most time and money on the first of those items. The ISO C++ standards committee tends to focus on the second with some concern for the third. My main effort was on the third and the fourth. The ISO committee is the focus for people who aim to improve the C++ language and standard library. Through such change they (we) hope to improve the state of the art in real-world C++ programming. The work of the C++ standards committee is the primary focus of the evolution of C++ and of this paper.

Thinking of C++ as a platform for applications, many have wondered why — after its initial success — C++ didn't shed its C heritage to "move up the food chain" and become a "truly object-oriented" applications programming language with a "complete" standard library. Any tendencies in that direction were squelched by a dedication to systems programming, C compatibility, and compatibility with early versions of C++ in most of the community associated with the standards committee. Also, there were never sufficient resources for massive projects in that community. Another important factor was the vigorous commercial opportunism by the major software and hardware vendors who each saw support for application building as *their* opportunity to distinguish themselves from their competition and to lock in their users. Minor players saw things in much the same light as they aimed to grow and prosper. However, there was no lack of support for the committee's activities within its chosen domain. Vendors rely on C++ for their systems and for highly demanding applications. Therefore, they have provided steady and most valuable support for the standards effort in the form of hosting and — more importantly — of key technical people attending.

For good and bad, ISO C++ [66] remains a general-purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

An explanation of the first three items from a historical perspective can be found in [120, 121]; the explanation of

"supports generic programming" is a major theme of this paper. Bringing aspects of generic programming into the mainstream is most likely C++'s greatest contribution to the software development community during this period.

The paper is organized in loose chronological order:

§1 Introduction

§2 Background: C++ 1979-1991 — early history, design criteria, language features.

§3 The C++ world in 1991 — the C++ standards process, chronology.

§4 Standard library facilities 1991-1998 — the C++ standard library with a heavy emphasis on its most important and innovative component: the STL (containers, algorithms, and iterators).

§5 Language features 1991-1998 — focusing on separate compilation of templates, exception safety, run-time type information and namespaces.

§6 Standards maintenance 1997-2003 — for stability, no additions were made to the C++ standard. However, the committee wasn't idle.

§7 C++ in real-world use — application areas; applications programming vs. systems programming; programming styles; libraries, Application Binary Interfaces (ABIs), and environments; tools and research; Java, C#, and C; dialects.

§8 C++0x — aims, constraints, language features, and library facilities.

§9 Retrospective — influences and impact; beyond C++.

The emphasis is on the early and later years: The early years shaped current C++ (C++98). The later ones reflect the response to experience with C++98 as represented by C++0x. It is impossible to discuss how to express ideas in code without code examples. Consequently, code examples are used to illustrate key ideas, techniques, and language facilities. The examples are explained to make them accessible to non-C++ programmers. However, the focus of the presentation is the people, ideas, ideals, techniques, and constraints that shape C++ rather than on language-technical details. For a description of what ISO C++ is today, see [66, 126]. The emphasis is on straightforward questions: What happened? When did it happen? Who were there? What were their reasons? What were the implications of what was done (or not done)?

C++ is a living language. The main aim of this paper is to describe its evolution. However, it is also a language with a great emphasis on backwards compatibility. The code that I describe in this paper still compiles and runs today. Consequently, I tend to use the present tense to describe it. Given the emphasis on compatibility, the code will probably also run as described 15 years from now. Thus, my use of the present tense emphasizes an important point about the evolution of C++.

2. Background: C++ 1979-1991

The early history of C++ (up until 1991) is covered by my HOPL-II paper [120]. The standard reference for the first 15 years of C++ is my book *The Design and Evolution of C++*, usually referred to as D&E [121]. It tells the story from the pre-history of C++ until 1994. However, to set the scene for the next years of C++, here is a brief summary of C++'s early history.

C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. It was intended to deliver that to real projects within half a year of the idea. It succeeded.

At the time, I realized neither the modesty nor the preposterousness of that goal. The goal was modest in that it did not involve innovation, and preposterous in both its time scale and its Draconian demands on efficiency and flexibility. While a modest amount of innovation did emerge over the early years, efficiency and flexibility have been maintained without compromise. While the goals for C++ have been refined, elaborated, and made more explicit over the years, C++ as used today directly reflects its original aims.

Starting in 1979, while in the Computer Science Research Center of Bell Labs, I first designed a dialect of C called "C with Classes". The work and experience with C with Classes from 1979 to 1983 determined the shape of C++. In turn, C with Classes was based on my experiences using BCPL and Simula as part of my PhD studies (in distributed systems) in the University of Cambridge, England. For challenging systems programming tasks I felt the need for a "tool" with the following properties:

- A good tool would have Simula's support for program organization – that is, classes, some form of class hierarchies, some form of support for concurrency, and compile-time checking of a type system based on classes. This I saw as support for the process of inventing programs; that is, as support for design rather than just support for implementation.
- A good tool would produce programs that ran as fast as BCPL programs and share BCPL's ability to combine separately compiled units into a program. A simple linkage convention is essential for combining units written in languages such as C, Algol68, Fortran, BCPL, assembler, etc., into a single program and thus not to suffer the handicap of inherent limitations in a single language.
- A good tool should allow highly portable implementations. My experience was that the "good" implementation I needed would typically not be available until "next year" and only on a machine I couldn't afford. This implied that a tool must have multiple sources of implementations (no monopoly would be sufficiently responsive to users of "unusual" machines and to poor graduate students), that there should be no complicated run-time support system to port, and that there should be only very

limited integration between the tool and its host operating system.

During my early years at Bell Labs, these ideals grew into a set of "rules of thumb" for the design of C++.

- General rules:
  - C++'s evolution must be driven by real problems.
  - Don't get involved in a sterile quest for perfection.
  - C++ must be useful *now*.
  - Every feature must have a reasonably obvious implementation.
  - Always provide a transition path.
  - C++ is a language, not a complete system.
  - Provide comprehensive support for each supported style.
  - Don't try to force people to use a specific programming style.
- Design support rules:
  - Support sound design notions.
  - Provide facilities for program organization.
  - Say what you mean.
  - All features must be affordable.
  - It is more important to allow a useful feature than to prevent every misuse.
  - Support composition of software from separately developed parts.
- Language-technical rules:
  - No implicit violations of the static type system.
  - Provide as good support for user-defined types as for built-in types.
  - Locality is good.
  - Avoid order dependencies.
  - If in doubt, pick the variant of a feature that is easiest to teach.
  - Syntax matters (often in perverse ways).
  - Preprocessor usage should be eliminated.
- Low-level programming support rules:
  - Use traditional (dumb) linkers.
  - No gratuitous incompatibilities with C.
  - Leave no room for a lower-level language below C++ (except assembler).
  - What you don't use, you don't pay for (zero-overhead rule).
  - If in doubt, provide means for manual control.

These criteria are explored in detail in Chapter 4 of D&E [121]. C++ as defined at the time of release 2.0 in 1989 strictly fulfilled these criteria; the fundamental tensions in the effort to design templates and exception-handling mechanisms for C++ arose from the need to depart from some aspects of these criteria. I think the most important property of these criteria is that they are only loosely connected with specific programming language features. Rather, they specify constraints on solutions to design problems.

Reviewing this list in 2006, I'm struck by two design criteria (ideals) that are not explicitly stated:

- There is a direct mapping of C++ language constructs to hardware
- The standard library is specified and implemented in C++

Coming from a C background and being deep in the development of the C++98 standard (§3.1), these points (at the time, and earlier) seemed so obvious that I often failed to emphasize them. Other languages, such as Lisp, Smalltalk, Python, Ruby, Java, and C#, do not share these ideals. Most languages that provide abstraction mechanisms still have to provide the most useful data structures, such as strings, lists, trees, associative arrays, vectors, matrices, and sets, as built-in facilities, relying on other languages (such as assembler, C, and C++) for their implementation. Of major languages, only C++ provides general, flexible, extensible, and efficient containers implemented in the language itself. To a large extent, these ideals came from C. C has those two properties, but not the abstraction mechanisms needed to define nontrivial new types.

The C++ ideal – from day one of C with Classes (§2.1) – was that the language should allow optimal implementation of arbitrary data structures and operations on them. This constrained the design of abstraction mechanisms in many useful ways [121] and led to new and interesting implementation techniques (for example, see §4.1). In turn, this ideal would have been unattainable without the "direct mapping of C++ language constructs to hardware" criterion. The key idea (from C) was that the operators directly reflected hardware operations (arithmetic and logical) and that access to memory was what the hardware directly offered (pointers and arrays). The combination of these two ideals is also what makes C++ effective for embedded systems programming [131] (§6.1).

2.1 The Birth of C with Classes

The work on what eventually became C++ started with an attempt to analyze the UNIX kernel to determine to what extent it could be distributed over a network of computers connected by a local area network. This work started in April of 1979 in the Computing Science Research Center of Bell Laboratories in Murray Hill, New Jersey. Two subproblems soon emerged: how to analyze the network traffic that would result from the kernel distribution and how to modularize the

kernel. Both required a way to express the module structure of a complex system and the communication pattern of the modules. This was exactly the kind of problem that I had become determined never to attack again without proper tools. Consequently, I set about developing a proper tool according to the criteria I had formed in Cambridge.

In October of 1979, I had the initial version of a pre-processor, called Cpre, that added Simula-like classes to C. By March of 1980, this pre-processor had been refined to the point where it supported one real project and several experiments. My records show the pre-processor in use on 16 systems by then. The first key C++ library, called "the task system", supported a coroutine style of programming [108, 115]. It was crucial to the usefulness of "C with Classes," as the language accepted by the pre-processor was called, in most early projects.

During the April to October period the transition from thinking about a "tool" to thinking about a "language" had occurred, but C with Classes was still thought of primarily as an extension to C for expressing modularity and concurrency. A crucial decision had been made, though. Even though support of concurrency and Simula-style simulations was a primary aim of C with Classes, the language contained no primitives for expressing concurrency; rather, a combination of inheritance (class hierarchies) and the ability to define class member functions with special meanings recognized by the pre-processor was used to write the library that supported the desired styles of concurrency. Please note that "styles" is plural. I considered it crucial — as I still do — that more than one notion of concurrency should be expressible in the language.

Thus, the language provided general mechanisms for organizing programs rather than support for specific application areas. This was what made C with Classes and later C++ a general-purpose language rather than a C variant with extensions to support specialized applications. Later, the choice between providing support for specialized applications or general abstraction mechanisms has come up repeatedly. Each time, the decision has been to improve the abstraction mechanisms.

2.2 Feature Overview

The earliest features included

- classes
- derived classes
- constructors and destructors
- public/private access control
- type checking and implicit conversion of function arguments.

In 1981, a few more features were added based on perceived need:

- inline functions
- default arguments
- overloading of the assignment operator.

Since a pre-processor was used for the implementation of C with Classes, only new features, that is features not present in C, needed to be described and the full power of C was directly available to users. Both of these aspects were appreciated at the time. In particular, having C as a subset dramatically reduced the support and documentation work needed. C with Classes was still seen as a dialect of C. Furthermore, classes were referred to as "An Abstract Data Type Facility for the C Language" [108]. Support for object-oriented programming was not claimed until the provision of virtual functions in C++ in 1983 [110].

A common question about "C with Classes" and later about C++ was "Why use C? Why didn't you build on, say, Pascal?" One version of my answer can be found in [114]:

> C is clearly not the cleanest language ever designed nor the easiest to use, so why do so many people use it?
>
> - C is *flexible*: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.
> - C is *efficient*: The semantics of C are 'low level'; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.
> - C is *available*: Given a computer, whether the tiniest micro or the largest super-computer, the chance is that there is an acceptable-quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so that a programmer rarely needs to design a new system from scratch.
> - C is *portable*: A C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible.
>
> Compared with these 'first-order' advantages, the 'second-order' drawbacks like the curious C declarator syntax and the lack of safety of some language constructs become less important.

Pascal was considered a toy language [78], so it seemed easier and safer to add type checking to C than to add

the features considered necessary for systems programming to Pascal. At the time, I had a positive dread of making mistakes of the sort where the designer, out of misguided paternalism or plain ignorance, makes the language unusable for real work in important areas. The ten years that followed clearly showed that choosing C as a base left me in the mainstream of systems programming where I intended to be. The cost in language complexity has been considerable, but (just barely) manageable. The problem of maintaining compatibility with an evolving C language and standard library is a serious one to this day (see §7.6).

In addition to C and Simula, I considered Modula-2, Ada, Smalltalk, Mesa, and Clu as sources for ideas for C++ [111], so there was no shortage of inspiration.

2.3 Work Environment

C with Classes was designed and implemented by me as a research project in the Computing Science Research Center of Bell Labs. This center provided a possibly unique environment for such work. When I joined in 1979, I was basically told to "do something interesting," given suitable computer resources, encouraged to talk to interesting and competent people, and given a year before having to formally present my work for evaluation.

There was a cultural bias against "grand projects" requiring many people, against "grand plans" like untested paper designs for others to implement, and against a class distinction between designers and implementers. If you liked such things, Bell Labs and other organizations had many places where you could indulge such preferences. However, in the Computing Science Research Center it was almost a requirement that you — if you were not into theory — personally implemented something embodying your ideas and found users who could benefit from what you built. The environment was very supportive for such work and the Labs provided a large pool of people with ideas and problems to challenge and test anything built. Thus I could write in [114]: "There never was a C++ paper design; design, documentation, and implementation went on simultaneously. Naturally, the C++ front-end is written in C++. There never was a "C++ project" either, or a "C++ design committee". Throughout, C++ evolved, and continues to evolve, to cope with problems encountered by users, and through discussions between the author and his friends and colleagues".

2.4 From C with Classes to C++

During 1982, it became clear to me that C with Classes was a "medium success" and would remain so until it died. The success of C with Classes was, I think, a simple consequence of meeting its design aim: C with Classes helped organize a large class of programs significantly better than C. Crucially, this was achieved without the loss of run-time efficiency and without requiring unacceptable cultural changes in development organizations. The factors limiting its success were partly the limited set of new facilities offered over C and partly the pre-processor technology used to implement C with Classes. C with Classes simply didn't provide support for people who were willing to invest significant effort to reap matching benefits: C with Classes was an important step in the right direction, but only one small step. As a result of this analysis, I began designing a cleaned-up and extended successor to C with Classes and implementing it using traditional compiler technology.

In the move from C with Classes to C++, the type checking was generally improved in ways that are possible only using a proper compiler front-end with full understanding of all syntax and semantics. This addressed a major problem with C with Classes. In addition,

- virtual functions
- function name and operator overloading
- references
- constants (`const`)
- many minor facilities

were added. To many, virtual functions were the major addition, as they enable object-oriented programming. I had been unable to convince my colleagues of their utility, but saw them as essential for the support of a key programming style ("paradigm").

After a couple of years of use, release 2.0 was a major release providing a significantly expanded set of features, such as

- type-safe linkage,
- abstract classes
- multiple inheritance.

Most of these extensions and refinements represented experience gained with C++ and could not have been added earlier without more foresight than I possessed.

2.5 Chronology

The chronology of the early years can be summarized:

1979 Work on C with Classes starts; first C with Classes use

1983 1st C++ implementation in use

1984 C++ named

1985 Cfront Release 1.0 (first commercial release); *The C++ Programming Language* (TC++PL) [112]

1986 1st commercial Cfront PC port (Cfront 1.1, Glockenspiel)

1987 1st GNU C++ release

1988 1st Oregon Software C++ release; 1st Zortech C++ release;

1989 Cfront Release 2.0; *The Annotated C++ Reference Manual* [35]; ANSI C++ committee (J16) founded (Washington, D.C.)

1990 1st ANSI X3J16 technical meeting (Somerset, New Jersey); templates accepted (Seattle, WA); exceptions accepted (Palo Alto, CA); 1st Borland C++ release

1991 1st ISO WG21 meeting (Lund, Sweden); Cfront Release 3.0 (including templates); *The C++ Programming Language (2nd edition)* [118]

On average, the number of C++ users doubled every 7.5 months from 1 in October 1979 to 400,000 in October of 1991 [121]. It is of course very hard to count users, but during the early years I had contacts with everyone who shipped compilers, libraries, books, etc., so I'm pretty confident of these numbers. They are also consistent with later numbers from IDC [61].

3. The C++ World in 1991

In 1991, the second edition of my *The C++ Programming Language* [118] was published to complement the 1989 language definition *The Annotated C++ Reference Manual* ("the ARM") [35]. Those two books set the standard for C++ implementation and to some extent for programming techniques for years to come. Thus, 1991 can be seen as the end of C++'s preparations for entry into the mainstream. From then on, consolidation was a major issue. That year, there were five C++ compilers to choose from (AT&T, Borland, GNU, Oregon, and Zortech) and three more were to appear in 1992 (IBM, DEC, and Microsoft). The October 1991 AT&T release 3.0 of Cfront (my original C++ compiler [121]) was the first to support templates. The DEC and IBM compilers both implemented templates and exceptions, but Microsoft's did not, thus seriously setting back efforts to encourage programmers to use a more modern style. The effort to standardize C++, which had begun in 1989, was officially converted into an international effort under the auspices of ISO. However, this made no practical difference as even the organizational meeting in 1989 had large non-US participation. The main difference is that we refer to ISO C++ rather than ANSI C++. The C++ programmer — and would-be C++ programmer — could now choose from among about 100 books of greatly varying aim, scope, and quality.

Basically, 1991 was an ordinary, successful year for the C++ community. It didn't particularly stand out from the years before or after. The reason to start our story here is that my HOPL-II paper [120] left off in 1991.

3.1 The ISO C++ Standards Process

For the C++ community, the ISO standards process is central: The C++ community has no other formal center, no other forum for driving language evolution, no other organization that cares for the language itself and for the community as a whole. C++ has no owner corporation that determines a path for the language, finances its development, and provides marketing. Therefore, the C++ standards committee became the place where language and standard library evolution is seriously considered. To a large extent, the 1991-2006 evolution of C++ was determined by what could be done by the volunteer individuals in the committee and how far the dreaded "design by committee" could be avoided.

The American National Standards Institute committee for C++ (ANSI J16) was founded in 1989; it now works under the auspices of INCITS (InterNational Committee for Information Technology Standards, a US organization). In 1991, it became part of an international effort under the auspices of ISO. Several other countries (such as France, Japan, and the UK) have their own national committees that hold their own meetings (in person or electronically) and send representatives to the ANSI/ISO meetings. Up until the final vote on the C++98 standard [63] in 1997, the committee met three times a year for week-long meetings. Now, it meets twice a year, but depends far more on electronic communications in between meetings. The committee tries to alternate meetings between the North American continent and elsewhere, mostly Europe.

The members of the J16 committee are volunteers who have to pay (about $800 a year) for the privilege of doing all the work. Consequently, most members represent companies that are willing to pay fees and travel expenses, but there is always a small number of people who pay their own way. Each company participating has one vote, just like each individual, so a company cannot stuff the committee with employees. People who represent their nations in the ISO (WG21) and participate in the national C++ panels pay or not according to their national standards organization rules. The J16 convener runs the technical sessions and does formal votes. The voting procedures are very democratic. First come one or more "straw votes" in a working group as part of the process to improve the proposal and gain consensus. Then come the more formal J16 votes in full committee where only accredited members vote. The final (ISO) votes are done on a per-nation basis. The aim is for consensus, defined as a massive majority, so that the resulting standard is good enough for everybody, if not necessarily exactly what any one member would have preferred. For the 1997/1998 final standards ballot, the ANSI vote was 43-0 and the ISO vote 22-0. We really did reach consensus, and even unanimity. I'm told that such clear votes are unusual.

The meetings tend to be quite collegial, though obviously there can be very tense moments when critical and controversial decisions must be made. For example, the debate leading up to the `export` vote strained the committee (§5.2). The collegiality has been essential for C++. The committee is the only really open forum where implementers and users from different — and often vigorously competing — organizations can meet and exchange views. Without a committee with a dense web of professional, personal, and organizational connections, C++ would have broken into a mess of feuding dialects. The number of people attending a meeting varies between 40 and 120. Obviously, the attendance increases when the meeting is in a location with many C++

programmers (e.g., Silicon Valley) or something major is being decided (e.g., should we adopt the STL?). At the Santa Cruz meeting in 1996, I counted 105 people in the room at the same time.

Most technical work is done by individuals between meetings or in working groups. These working groups are officially "ad hoc" and have no formal standing. However, some lasts for many years and serve as a focus for work and as the institutional memory of the committee. The main long-lived working groups are:

- *Core* — chairs: Andrew Koenig, Josée Lajoie, Bill Gibbons, Mike Miller, Steve Adamczyk.
- *Evolution* (formerly *extensions*) — chair: Bjarne Stroustrup.
- *Library* — chairs: Mike Vilot, Beman Dawes, Matt Austern, Howard Hinnant.

When work is particularly hectic, these groups split into sub-working-groups focussed on specific topics. The aim is to increase the degree of parallelism in the process and to better use the skills of the many people present.

The official "chair" of the whole committee whose primary job is the ensure that all formal rules are obeyed and to report back to SC22 is called the convener. The original convener was Steve Carter (BellCore). Later Sam Harbinson (Tartan Labs and Texas Instruments), Tom Plum (Plum Hall), and Herb Sutter (Microsoft) served.

The J16 chairman conducts the meeting when the whole committee is gathered. Dmitri Lenkov (Hewlett-Packard) was the original J16 chair and Steve Clamage (Sun) took over after him.

The draft standard text is maintained by the project editor. The original project editor was Jonathan Shopiro (AT&T). He was followed by Andrew Koenig (AT&T) whose served 1992-2004, after which Pete Becker (Dinkumware) took over "the pen".

The committee consists of individuals of diverse interests, concerns, and backgrounds. Some represent themselves, some represent giant corporations. Some use PCs, some use UNIX boxes, some use mainframes, etc. Some would like C++ to become more of an object-oriented language (according to a variety of definitions of "object-oriented"), others would have been more comfortable had ANSI C been the end point of C's evolution. Many have a background in C, some do not. Some have a background in standards work, many do not. Some have a computer science background, some do not. Most are programmers, some are not. Some are language lawyers, some are not. Some serve end-users, some are tools suppliers. Some are interested in large projects, some are not. Some are interested in C compatibility, some are not. It is hard to find a generalization that covers them all.

This diversity of backgrounds has been good for C++; only a very diverse group could represent the diverse interests of the C++ community. The end results — such as the 1998 standard and the Technical Reports (§6.1, §6.2) — are something that is good enough for everyone represented, rather than something that is ideal for any one subcommunity. However, the diversity and size of the membership do make constructive discussion difficult and slow at times. In particular, this very open process is vulnerable to disruption by individuals whose technical or personal level of maturity doesn't encourage them to understand or respect the views of others. Part of the consideration of a proposal is a process of education of the committee members. Some members have claimed — only partly in jest — that they attend to get that education. I also worry that the voice of C++ users (that is, programmers and designers of C++ applications) can be drowned by the voices of language lawyers, would-be language designers, standards bureaucrats, implementers, tool builders, etc.

To get an idea about what organizations are represented, here are some names from the 1991-2005 membership lists: Apple, AT&T, Bellcore, Borland, DEC, Dinkumware, Edison Design Group (EDG), Ericsson, Fujitsu, Hewlett-Packard, IBM, Indiana University, Los Alamos National Labs, Mentor Graphics, Microsoft, NEC, Object Design, Plum Hall, Siemens Nixdorf, Silicon Graphics, Sun Microsystems, Texas Instruments, and Zortech.

Changing the definition of a widely used language is very different from simple design from first principles. Whenever we have a "good idea", however major or minor, we must remember that

- there are hundreds of millions of lines of code "out there" — most will not be rewritten however much gain might result from a rewrite
- there are millions of programmers "out there" — most won't take out time to learn something new unless they consider it essential
- there are decade-old compilers still in use — many programmers can't use a language feature that doesn't compile on every platform they support
- there are many millions of outdated textbooks out there — many will still be in use in five years' time

The committee considers these factors and obviously that gives a somewhat conservative bias. Among other things, the members of the committee are indirectly responsible for well over 100 million lines of code (as representatives of their organizations). Many of the members are on the committee to promote change, but almost all do so with a great sense of caution and responsibility. Other members feel that their role is more along the lines of avoiding unnecessary and dangerous instability. Compatibility with previous versions of C++ (back to ARM C++ [35]), previous versions of C (back to K&R C [76]), and generations of corporate dialects is a serious issue for most members. We (the members of the committee) try to face future challenges, such as con-

currency, but we do so remembering that C++ is at the base of many tool chains. Break C++ and the major implementations of Java and C# would also break. Obviously, the committee couldn't "break C++" by incompatibilities even if it wanted to. The industry would simply ignore a seriously incompatible standard and probably also start migrating away from C++.

3.2 Chronology

Looking forward beyond 1991, we can get some idea of the process by listing some significant decisions (votes):

1993 Run-time type identification accepted (Portland, Oregon) §5.1.2; namespaces accepted (Munich, Germany) §5.1.1

1994 `string` (templatized by character type) (San Diego, California)

1994 The STL (San Diego, California) §4.1

1996 `export` (Stockholm, Sweden) §5.2

1997 Final committee vote on the complete standard (Morristown, New Jersey)

1998 ISO C++ standard [63] ratified

2003 Technical Corrigendum ("mid-term bug-fix release") [66]; work on C++0x starts

2004 Performance technical report [67] §6.1; Library Technical Report (hash tables, regular expressions, smart pointers, etc.) [68] §6.2

2005 1st votes on features for C++0x (Lillehammer, Norway); `auto`, `static_assert`, and rvalue references accepted in principle; §8.3.2

2006 1st full committee (official) votes on features for C++0x (Berlin, Germany)

The city names reflect the location of the meeting where the decision was taken. They give a flavor of the participation. When the committee meets in a city, there are usually a dozen or more extra participants from nearby cities and countries. It is also common for the host to take advantage of the influx of C++ experts — many internationally known — to arrange talks to increase the understanding of C++ and its standard in the community. The first such arrangement was in Lund in 1991 when the Swedish delegation collaborated with Lund University to put on a two-day C++ symposium.

This list gives a hint of the main problem about the process from the point of view of someone trying to produce a coherent language and library, rather than a set of unrelated "neat features". At any time, the work focused on a number of weakly related specific topics, such as the definition of "undefined", whether it is possible to resume from an exception (it isn't), what functions should be provided for `string`, etc. It is extremely hard to get the committee to agree on an overall direction.

3.3 Why Change?

Looking at the list of decisions and remembering the committee's built-in conservative bias, the obvious question is: "Why change anything?" There are people who take the position that "standardization is to document existing practice". They were never more than a tiny fraction of the committee membership and represent an even smaller proportion of the vocal members of the C++ committee. Even people who say that they want "no change" ask for "just one or two improvements". In this, the C++ committee strongly resembles other language standardization groups.

Basically, we (the members of the committee) desire change because we hold the optimistic view that better language features and better libraries lead to better code. Here, "better" means something like "more maintainable", "easier to read", "catches more errors", "faster", "smaller", "more portable", etc. People's criteria differ, sometimes drastically. This view is optimistic because there is ample evidence that people can — and do — write really poor code in every language. However, most groups of programmers — including the C++ committee — are dominated by optimists. In particular, the conviction of a large majority of the C++ committee has consistently been that the quality of C++ code can be improved over the long haul by providing better language features and standard-library facilities. Doing so takes time: in some cases we may have to wait for a new generation of programmers to get educated. However, the committee is primarily driven by optimism and idealism — moderated by vast experience — rather than the cynical view of just giving people what they ask for or providing what "might sell".

An alternative view is that the world changes and a living language will change — whatever any committee says. So, we can work on improvements — or let others do it for us. As the world changes, C++ must evolve to meet new challenges. The most obvious alternative would not be "death" but capture by a corporation, as happened with Pascal and Objective C, which became Borland and Apple corporate languages, respectively.

After the Lund (Sweden) meeting in 1991, the following cautionary tale became popular in the C++ community:

> We often remind ourselves of the good ship Vasa. It was to be the pride of the Swedish navy and was built to be the biggest and most beautiful battleship ever. Unfortunately, to accommodate enough statues and guns, it underwent major redesigns and extension during construction. The result was that it only made it halfway across Stockholm harbor before a gust of wind blew it over and it sank, killing about 50 people. It has been raised and you can now see it in a museum in Stockholm. It is a beauty to behold — far more beautiful at the time than its unextended first design and far more beautiful today than if it had suffered the usual fate of a 17th century battleship — but that is

no consolation to its designer, builders, and intended users.

This story is often recalled as a warning against adding features (that was the sense in which I told it to the committee). However, to complicate matters, there is another side to the story: Had the Vasa been completed as originally designed, it would have been sent to the bottom full of holes the first time it encountered a "modern two-deck" battleship. Ignoring changes in the world isn't an option (either).

4. The Standard Library: 1991-1998

After 1991, most major changes to the draft C++ standard were in the standard library. Compared to that, the language features were little changed even though we made an apparently infinite number of improvements to the text of the standard. To gain a perspective, note that the standard is 718 pages: 310 define the language and 366 define the standard library. The rest are appendices, etc. In addition, the C++ standard library includes the C standard library by reference; that's another 81 pages. To compare, the base document for the standardization, the reference manual of TC++PL2 [118], contained 154 pages on the language and just one on the standard library.

By far the most innovative component of the standard library is "the STL" — the containers, iterators, and algorithms part of the library. The STL influenced and continues to influence not only programming and design techniques but also the direction of new language features. Consequently, this is treated first here and in far greater detail than other standard library components.

4.1 The STL

The STL was the major innovation to become part of the standard and the starting point for much of the new thinking about programming techniques that have occurred since. Basically, the STL was a revolutionary departure from the way the C++ community had been thinking about containers and their use.

4.1.1 Pre-STL containers

From the earliest days of Simula, containers (such as lists) had been intrusive: An object could be put into a container if and only if its class had been (explicitly or implicitly) derived from a specific `Link` and/or `Object` class. This class contains the link information needed for management of objects in a container and provides a common type for elements. Basically, such a container is a container of references (pointers) to links/objects. We can graphically represent such a list like this:

The links come from a base class `Link`.

Similarly, an "object-oriented" vector is a basically an array of references to objects. We can graphically represent such a vector like this:

The references to objects in the vector data structure point to objects of an `Object` base class. This implies that objects of a fundamental type, such as `int` and `double`, can't be put directly into containers and that the array type, which directly supports fundamental types, must be different from other containers:

Furthermore, objects of really simple classes, such as `complex` and `Point`, can't remain optimal in time and space if we want to put them into a container. In other words, Simula-style containers are intrusive, relying on data fields inserted into their element types, and provide indirect access to objects through pointers (references). Furthermore, Simula-style containers are not statically type safe. For example, a `Circle` may be added to a `list`, but when it is extracted we know only that it is an `Object` and need to apply a cast (explicit type conversion) to regain the static type.

Thus, Simula containers provide dissimilar treatment of built-in and user-defined types (only some of the latter can be in containers). Similarly, arrays are treated differently from user-defined containers (only arrays can hold fundamental types). This is in direct contrast to two of the clearest language-technical ideals for C++:

- Provide the same support for built-in and user-defined types
- What you don't use, you don't pay for (zero-overhead rule).

Smalltalk has the same fundamental approach to containers as Simula, though it makes the base class universal and thus implicit. The problems also appear in later languages, such as Java and C# (though they – like Smalltalk – make use of a universal class and C# 2.0 applies C++-like specialization to optimize containers of integers). Many early C++ libraries (e.g. the NIHCL [50], early AT&T libraries [5]) also followed this model. It does have significant utility and many designers were familiar with it. However, I considered this double irregularity and the inefficiency (in time and space) that goes with it unacceptable for a truly general-purpose library (you can find a summary of my analysis in §16.2 of TC++PL3 [126]).

The lack of a solution to these logical and performance problems was the fundamental reason behind my "biggest mistake" of not providing a suitable standard library for C++ in 1985 (see D&E [121] §9.2.3): I didn't want to ship anything that couldn't directly handle built-in types as elements and wasn't statically type safe. Even the first "C with Classes" paper [108] struggled with that problem, unsuccessfully trying to solve it using macros. Furthermore, I specifically didn't want to provide something with covariant containers. Consider a `Vector` of some "universal" `Object` class:

```
void f(Vector& p)
{
     p[2] = new Pear;
}

void g()
{
   Vector apples(10);  // container of apples
   for(int i=0; i<10; ++i)
      apples[i] = new Apple;
   f(apples);
   // now apples contains a pear
}
```

The author of `g` pretends that `apples` is a `Vector` of `Apple`s. However, `Vector` is a perfectly ordinary object-oriented container, so it really contains (pointers to) `Object`s. Since `Pear` is also an `Object`, `f` can without any problems put a `Pear` into it. It takes a run-time check (implicit or explicit) to catch the error/misconception. I remember how shocked I was when this problem was first explained to me (sometime in the early 1980s). It was just too shoddy. I was determined that no container written by me should give users such nasty surprises and significant run-time checking costs. In modern (i.e. post-1988) C++ the problem is solved using a container parameterized by the element type. In particular, Standard C++ offers `vector<Apple>` as a solution. Note that a `vector<Apple>` does not convert to a `vector<Fruit>` even when `Apple` implicitly converts to `Fruit`.

4.1.2 The STL emerges

In late 1993, I became aware of a new approach to containers and their use that had been developed by Alex Stepanov. The library he was building was called "The STL". Alex then worked at HP Labs but he had earlier worked for a couple of years at Bell Labs, where he had been close to Andrew Koenig and where I had discussed library design and template mechanisms with him. He had inspired me to work harder on generality and efficiency of some of the template mechanisms, but fortunately he failed to convince me to make templates more like Ada generics. Had he succeeded, he wouldn't have been able to design and implement the STL!

Alex showed the latest development in his decades-long research into generic programming techniques aiming for "the most general and most efficient code" based on a rigorous mathematical foundation. It was a framework of containers and algorithms. He first explained his ideas to Andrew, who after playing with the STL for a couple of days showed it to me. My first reaction was puzzlement. I found the STL style of containers and container use very odd, even ugly and verbose. For example, you sort a `vector` of `double`s, `vd`, according to their absolute value like this:

```
sort(vd.begin(), vd.end(), Absolute<double>());
```

Here, `vd.begin()` and `vd.end()` specify the beginning and end of the `vector`'s sequence of elements and `Absolute<double>()` compares absolute values.

The STL separates algorithms from storage in a way that's classical (as in math) but not object oriented. Furthermore, it separates policy decisions of algorithms, such as sorting criteria and search criteria, from the algorithm, the container, and the element class. The result is unsurpassed flexibility and — surprisingly — performance.

Like many programmers acquainted with object-oriented programming, I thought I knew roughly how code using containers had to look. I imagined something like Simula-style containers augmented by templates for static type safety and maybe abstract classes for iterator interfaces. The STL code looked very different. However, over the years I had developed a checklist of properties that I considered important for containers:

1. Individual containers are simple and efficient.
2. A container supplies its "natural" operations (e.g., list provides `put` and `get` and vector provides subscripting).
3. Simple operators, such as member access operations, do not require a function call for their implementation.
4. Common functionality can be provided (maybe through iterators or in a base class)
5. Containers are by default statically type-safe and homogeneous (that is, all elements in a container are of the same type).
6. A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
7. Containers are non-intrusive (i.e., an object need not have a special base class or link field to be a member of a container).
8. A container can contain elements of built-in types
9. A container can contain `struct`s with externally imposed layouts.
10. A container can be fitted into a general framework (of containers and operations on containers).
11. A container can be sorted without it having a `sort` member function.

12. “Common services” (such as persistence) can be provided in a single place for a set of containers (in a base class?).

A slightly differently phrased version can be found in [124].

To my amazement the STL met all but one of the criteria on that list! The missing criterion was the last. I had been thinking of using a common base class to provide services (such as persistence) for all derived classes (e.g., all objects or all containers). However, I didn’t (and don’t) consider such services intrinsic to the notion of a container. Interestingly, some “common services” can be expressed using “concepts” (§8.3.3) that specifically address the issue of what can be required of a set of types, so C++0x (§8) is likely to bring the STL containers even closer to the ideals expressed in that list.

It took me some time — weeks — to get comfortable with the STL. After that, I worried that it was too late to introduce a completely new style of library into the C++ community. Considering the odds to get the standards committee to accept something new and revolutionary at such a late stage of the standards process, I decided (correctly) that those odds were very low. Even at best, the standard would be delayed by a year — and the C++ community urgently needed that standard. Also, the committee is fundamentally a conservative body and the STL was revolutionary.

So, the odds were poor, but I plodded on hoping. After all, I really did feel very bad about C++ not having a sufficiently large and sufficiently good standard library [120] (D&E [121] §9.2.3). Andrew Koenig did his best to build up my courage and Alex Stepanov lobbied Andy and me as best he knew how to. Fortunately, Alex didn’t quite appreciate the difficulties of getting something major through the committee, so he was less daunted and worked on the technical aspects and on teaching Andrew and me. I began to explain the ideas behind the STL to others; for example, the examples in D&E §15.6.3.1 came from the STL and I quoted Alex Stepanov: “C++ is a powerful enough language — the first such language in our experience — to allow the construction of generic programming components that combine mathematical precision, beauty, and abstractness with the efficiency of non-generic hand-crafted code”. That quote is about generic programming in general (§7.2.1) and the STL in particular.

Andrew Koenig and I invited Alex to give an evening presentation at the October 1993 standards committee meeting in San Jose, California: “It was entitled *The Science of C++ Programming* and dealt mostly with axioms of regular types — connecting construction, assignment and equality. I also described axioms of what is now called Forward Iterators. I did not at all mention any containers and only one algorithm: `find`”. [105]. That talk was an audacious piece of rabble rousing that to my amazement and great pleasure basically swung the committee away from the attitude of “it’s impossible to do something major at this stage” to “well, let’s have a look”.

That was the break we needed! Over the next four months, we (Alex, his colleague Meng Lee, Andrew, and I) experimented, argued, lobbied, taught, programmed, redesigned, and documented so that Alex was able to present a complete description of the STL to the committee at the March 1994 meeting in San Diego, California. Alex arranged a meeting for C++ library implementers at HP later in 1994. The participants were Tom Keffer (Rogue Wave), Meng Lee (HP), Nathan Myers (Rogue Wave), Larry Podmolik (Anderson Consulting), Mike Vilot, Alex, and I. We agreed on many principles and details, but the size of the STL emerged as the major obstacle. There was no consensus about the need for large parts of the STL, there was a (realistic) worry that the committee wouldn’t have the time to properly review and more formally specify something that large, and people were simply daunted by the sheer number of things to understand, implement, document, teach, etc. Finally, at Alex’s urging, I took a pen and literally crossed out something like two thirds of all the text. For each facility, I challenged Alex and the other library experts to explain — very briefly — why it couldn’t be cut and why it would benefit most C++ programmers. It was a horrendous exercise. Alex later claimed that it broke his heart. However, what emerged from that slashing is what is now known as the STL [103] and it made it into the ISO C++ standard at the October 1994 meeting in Waterloo, Canada — something that the original and complete STL would never have done. Even the necessary revisions of the “reduced STL” delayed the standard by more than a year. The worries about size and complexity (even after my cuts) were particularly acute among library implementers concerned about the cost of providing a quality implementation. For example, I remember Roland Hartinger (representing Siemens and Germany) worrying that acceptance of the STL would cost his department one million marks. In retrospect, I think that I did less damage than we had any right to hope for.

Among all the discussions about the possible adoption of the STL one memory stands out: Beman Dawes calmly explaining to the committee that he had thought the STL too complex for ordinary programmers, but as an exercise he had implemented about 10% of it himself so he no longer considered it beyond the standard. Beman was (and is) one of the all too rare application builders in the committee. Unfortunately, the committee tends to be dominated by compiler, library, and tools builders.

I credit Alex Stepanov with the STL. He worked with the fundamental ideals and techniques for well over a decade before the STL, unsuccessfully using languages such as Scheme and Ada [101]. However, Alex is always the first to insist that others took part in that quest. David Musser (a professor at Rensselaer Polytechnic Institute) has been working with Alex on generic programming for almost two

decades and Meng Lee worked closely with him at HP helping to program the original STL. Email discussions between Alex and Andrew Koenig also helped. Apart from the slashing exercise, my technical contributions were minor. I suggested that various information related to memory be collected into a single object — what became the allocators. I also drew up the initial requirement tables on Alex's blackboard, thus creating the form in which the standard specifies the requirements that STL templates place on their arguments. These requirements tables are actually an indicator that the language is insufficiently expressive — such requirements should be part of the code; see §8.3.3.

Alex named his containers, iterators, and algorithm library "the STL". Usually, that's considered an acronym for "Standard Template Library". However, the library existed — with that name — long before it was any kind of standard and most parts of the standard library rely on templates. Wits have suggested "STepanov and Lee" as an alternative explanation, but let's give Alex the final word: " 'STL' stands for 'STL' ". As in many other cases, an acronym has taken on a life of its own.

4.1.3 STL ideals and concepts

So what is the STL? It comes from an attempt to apply the mathematical ideal of generality to the problem of data and algorithms. Consider the problem of storing objects in containers and writing algorithms to manipulate such objects. Consider this problem in the light of the ideals of direct, independent, and composable representation of concepts:

- express concepts directly in code
- express relations among concepts directly in code
- express independent concepts in independent code
- compose code representing concepts freely wherever the composition makes sense

We want to be able to

- store objects of a variety of types (e.g. `ints`, `Points`, and pointers to `Shapes`)
- store those objects in a variety of containers (e.g. `list`, `vector`, and `map`),
- apply a variety of algorithms (e.g. `sort`, `find`, and `accumulate`) to the objects in the containers, and
- use a variety of criteria (comparisons, predicates, etc.) for those algorithms

Furthermore, we want the use of these objects, containers, and algorithms to be statically type safe, as fast as possible, as compact as possible, not verbose, and readable. Achieving all of this simultaneously is not easy. In fact, I spent more than ten years unsuccessfully looking for a solution to this puzzle (§4.1.2).

The STL solution is based on parameterizing containers with their element types and on completely separating the algorithms from the containers. Each type of container provides an iterator type and all access to the elements of the container can be done using only iterators of that type. The iterator defines the interface between the algorithm and the data on which it operates. That way, an algorithm can be written to use iterators without having to know about the container that supplied them. Each type of iterator is completely independent of all others except for supplying the same semantics to required operations, such as * and ++.

Algorithms use iterators and container implementers implement iterators for their containers:

Let's consider a fairly well-known example, the one that Alex Stepanov initially showed to the committee (San Jose, California, 1993). We want to find elements of various types in various containers. First, here are a couple of containers:

```
vector<int> vi;    // vector of ints
list<string> ls;   // list of strings
```

The `vector` and `list` are the standard library versions of the notions of vector and list implemented as templates. An STL container is a non-intrusive data structure into which you can copy elements of any type. We can graphically represent a (doubly linked) `list<string>` like this:

Note that the link information is *not* part of the element type. An STL container (here, `list`) manages the memory for its elements (here, `strings`) and supplies the link information.

Similarly, we can represent a `vector<int>` like this:

Note that the elements are stored in memory managed by the `vector` and `list`. This differs from the Simula-style containers in §4.1.1 in that it minimizes allocation operations, minimizes per-object memory, and saves an indirection on each access to an element. The corresponding cost is a copy operation when an object is first entered into a container; if

a copy is expensive, programmers tend to use pointers as elements.

Assume that the containers `vi` and `ls` have been suitably initialized with values of their respective element types. It then makes sense to try to find the first element with the value `777` in `vi` and the first element with the value `"Stepanov"` in `ls`:

```
vector<int>::iterator p
     = find(vi.begin(),vi.end(),777);
list<string>::iterator q
     = find(ls.begin(),ls.end(),"Stepanov");
```

The basic idea is that you can consider the elements of any container as a sequence of elements. A container "knows" where its first element is and where its last element is. We call an object that points to an element "an iterator". We can then represent the elements of a container by a pair of iterators, `begin()` and `end()`, where `begin()` points to the first element and `end()` to one-beyond-the-last element. We can represent this general model graphically:

The `end()` iterator points to one-past-the-last element rather than to the last element to allow the empty sequence not to be a special case:

What can you do with an iterator? You can get the value of the element pointed to (using `*` just as with a pointer), make the iterator point to the next element (using `++` just as with a pointer) and compare two iterators to see if they point to the same element (using `==` or `!=` of course). Surprisingly, this is sufficient for implementing `find()`:

```
template<class Iter, class T>
Iter find(Iter first, Iter last, const T& val)
{
   while (first!=last && *first!=val)
      ++first;
   return first;
}
```

This is a simple — very simple, really — function template. People familiar with C and C++ pointers should find the code easy the read: `first!=last` checks whether we reached the end and `*first!=val` checks whether we found the value that we were looking for (`val`). If not, we increment the iterator `first` to make it point to the next element and try again. Thus, when `find()` returns, its value will point to either the first element with the value `val` or one-past-the-last element (`end()`). So we can write:

```
vector<int>::iterator p =
   find(vi.begin(),vi.end(),7);

if (p != vi.end()) {  // we found 7
   // ...
}
else {                // no 7 in vi
   // ...
}
```

This is very, very simple. It is simple like the first couple of pages in a math book and simple enough to be really fast. However, I know that I wasn't the only person to take significant time figuring out what really is going on here and longer to figure out why this is actually a good idea. Like simple math, the first STL rules and principles generalize beyond belief.

Consider first the implementation: In the call `find(vi.begin(),vi.end(),7)`, the iterators `vi.begin()` and `vi.end()` become `first` and `last`, respectively, inside. To `find()`, `first` is simply "something that points to an `int`". The obvious implementation of `vector<int>::iterator` is therefore a pointer to `int`, an `int*`. With that implementation, `*` becomes pointer dereference, `++` becomes pointer increment, and `!=` becomes pointer comparison. That is, the implementation of `find()` is obvious and optimal.

Please note that the STL does not use function calls to access the operations (such as `*` and `!=`) that are effectively arguments to the algorithm because they depend on a template argument. In this, templates differ radically from most mechanisms for "generics", relying on indirect function calls (like virtual functions), as provided by Java and C#. Given a good optimizer, `vector<int>::iterator` can without overhead be a class with `*` and `++` provided as inline functions. Such optimizers are now not uncommon and using a iterator class rather than a pointer improves type checking by catching unwarranted assumptions, such as that the `iterator` for a `vector` is a pointer:

```
int* p = find(vi.begin(),vi.end(),7); // error

// verbose, but correct:
vector<int>::iterator q =
   find(vi.begin(),vi.end(),7);
```

C++0x will provide ways of dealing with the verbosity; see §8.3.2.

In addition, *not* defining the interface between an algorithm and its type arguments as a set of functions with unique types provides a degree of flexibility that proved very important [130] (§8.3.3). For example, the standard library algorithm `copy` can copy between different container types:

```
void f(list<int>& lst, vector<int>& v)
{
     copy(lst.begin(), lst.end(), v.begin());
     // ...
     copy(v.begin(), v.end(), lst.end());
}
```

So why didn't we just dispense with all that "iterator stuff" and use pointers? One reason is that `vector<int>::iterator` could have been a class providing range checked access. For a less subtle explanation, have a look at another call of `find()`:

```
list<string>::iterator q =
   find(ls.begin(),ls.end(),"McIlroy");

if (q != ls.end()) {  // we found "McIlroy"
   // ...
}
else {               // no "McIlroy" in ls
   // ...
}
```

Here, `list<string>::iterator` isn't going to be a `string*`. In fact, assuming the most common implementation of a linked list, `list<string>::iterator` is going to be a `Link<string>*` where `Link` is a link node type, such as:

```
template<class T> struct Link {
   T value;
   Link* suc;
   Link* pre;
};
```

That implies that * means `p->value` ("return the value field"), ++ means `p->suc` ("return a pointer to the next link"), and != pointer comparison (comparing `Link*`s). Again the implementation is obvious and optimal. However, it is completely different from what we saw for `vector<int>::iterator`.

We used a combination of templates and overload resolution to pick radically different, yet optimal, implementations of operations used in the definition of `find()` for each use of `find()`. Note that there is no run-time dispatch, no virtual function calls. In fact, there are only calls of trivially inlined functions and fundamental operations, such as * and ++ for a pointer. In terms of execution time and code size, we have hit the absolute minimum!

Why not use "sequence" or "container" as the fundamental notion rather than "pair of iterators"? Part of the reason is that "pair of iterators" is simply a more general concept than "container". For example, given iterators, we can sort the first half of a container only: `sort(vi.begin(), vi.begin()+vi.size()/2)`. Another reason is that the STL follows the C++ design rules that we must provide transition paths and support built-in and user-defined types uniformly. What if someone kept data in an ordinary array? We can still use the STL algorithms. For example:

```
int buf[max];
// ... fill buf ...
int*  p = find(buf,buf+max,7);

if (p != buf+max) {  // we found 7
   // ...
}
else {               // no 7 in buf
   // ...
}
```

Here, the `*`, `++`, and `!=` in `find()` really are pointer operations! Like C++ itself, the STL is compatible with older notions such as C arrays. Thus, the STL meets the C++ ideal of always providing a transition path (§2). It also meets the ideal of providing uniform treatment to user-defined types (such as `vector`) and built-in types (in this case, array) (§2).

Another reason for basing algorithms on iterators, rather than on containers or an explicit sequence abstraction, was the desire for optimal performance: using iterators directly rather than retrieving a pointer or an index from another abstraction eliminates a level of indirection.

As adopted as the containers and algorithms framework of the ISO C++ standard library, the STL consists of a dozen containers (such as `vector`, `list`, and `map`) and data structures (such as arrays) that can be used as sequences. In addition, there are about 60 algorithms (such as `find`, `sort`, `accumulate`, and `merge`). It would not be reasonable to present all of those here. For details, see [6, 126].

So, we can use simple arithmetic to see how the STL technique of separating algorithms from containers reduces the amount of source code we have to write and maintain. There are 60*12 (that is, 720) combinations of algorithm and container in the standard but just 60+12 (that is, 72) definitions. The separation reduces the combinatorial explosion to a simple addition. If we consider element types and policy parameters (function objects, see §4.1.4) for algorithms we see an even more impressive gain: Assume that we have N algorithms with M alternative criteria (policies) and X containers with Y element types. Then, the STL approach gives us N+M+X+Y definitions whereas "hand-crafted code" requires N*M*X*Y definitions. In real designs, the difference isn't quite that dramatic because typically designers attack that huge N*M*X*Y figure with a combination of conversions (one container to another, one data type to another), class derivations, function parameters, etc., but the STL ap-

proach is far cleaner and more systematic than earlier alternatives.

The key to both the elegance and the performance of the STL is that it — like C++ itself — is based directly on the hardware model of memory and computation. The STL notion of a sequence is basically that of the hardware's view of memory as a set of sequences of objects. The basic semantics of the STL map directly into hardware instructions allowing algorithms to be implemented optimally. The compile-time resolution of templates and the perfect inlining they support is then key to the efficient mapping of the high-level expression using the STL to the hardware level.

4.1.4 Function objects

The STL and generic programming in general owes a — freely and often acknowledged (e.g., [124]) — debt to functional programming. So where are the lambdas and higher-order functions? C++ doesn't directly support anything like that (though there are always proposals for nested functions, closures, lambdas, etc.; see §8.2). Instead, classes that define the application operator, called function objects (or even "functors"), take that role and have become the main mechanism of parameterization in modern C++. Function objects build on general C++ mechanisms to provide unprecedented flexibility and performance.

The STL framework, as described so far, is somewhat rigid. Each algorithm does exactly one thing in exactly the way the standard specifies it to. For example, using `find()`, we find an element that is equal to the value we give as the argument. It is actually more common to look for an element that has some desired property, such as matching strings without case sensitivity or matching floating-point values allowing for very slight differences.

As an example, instead of finding a value 7, let's look for a value that meets some predicate, say, being less than 7:

```
vector<int>::iterator p =
   find_if(v.begin(),v.end(),Less_than<int>(7));

if (p != vi.end()) { // element < 7
   // ...
}
else {                // no such element
   // ...
}
```

What is `Less_than<int>(7)`? It is a function object; that is, it is an object of a class that has the application operator, `()`, defined to perform an action:

```
template<class T> struct Less_than {
   T value;
   Less_than(const T& v) :value(v) { }
   bool operator()(const T& v) const
      { return v<value; }
};
```

For example:

```
Less_than<double> f(3.14); // f holds 3.14
bool b1 = f(3);  // true: 3<3.14 is true
bool b2 = f(4);  // false: 4<3.14 is false
```

From the vantage point of 2005, it seems odd that function objects are not mentioned in D&E or TC++PL1. They deserve a whole section. Even the use of a user-defined application operator, `()`, isn't mentioned even though it has had a long and distinguished career. For example, it was among the initial set of operators (after =; see D&E §3.6) that I allowed to be overloaded and was among many other things used to mimic Fortran subscript notation [112].

We used the STL algorithm `find_if` to apply `Less_than<int>(7)` to the elements of the `vector`. The definition of `find_if` differs from `find()`'s definition in using a user-supplied predicate rather than equality:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
   while (first!=last && !pred(*first))
      ++first;
   return first;
}
```

We simply replaced `*first!=val` with `!pred(*first)`. The function template `find_if()` will accept any object that can be called given an element value as its argument. In particular, we could call `find_if()` with an ordinary function as its third argument:

```
bool less_than_7(int a)
{
    return a<7;
}

vector<int>::iterator p =
   find_if(v.begin(),v.end(),less_than_7);
```

However, this example shows why we often prefer a function object over a function: The function object can be initialized with one (or more) arguments and carry information along for later use. A function object can carry a state. That makes for more general and more elegant code. If needed, we can also examine that state later. For example:

```
template<class T>
struct Accumulator {  // keep the sum of n values
   T value;
   int count;
   Accumulator() :value(), count(0) { }
   Accumulator(const T& v) :value(v), count(0) { }
   void operator()(const T& v)
      { ++count; value+=v; }
};
```

An `Accumulator` object can be passed to an algorithm that calls it repeatedly. The partial result is carried along in the object. For example:

```
int main()
{
```

```
    vector<double> v;
    double d;
    while (cin>>d) v.push_back(d);

    Accumulator<double> ad;
    ad = for_each(v.begin(),v.end(), ad);
    cout << "sum==" << ad.value
         << ", mean==" << ad.value/ad.count
         << '\n';
      return 0;
}
```

The standard library algorithm `for_each` simply applies its third argument to each element of its sequence and returns that argument as its return value. The alternative to using a function object would be a messy use of global variables to hold `value` and `count`. In a multithreaded system, such use of global variables is not just messy, but gives incorrect results.

Interestingly, simple function objects tend to perform better than their function equivalents. The reason is that they tend to be simple classes without virtual functions, so that when we call a member function the compiler knows exactly which function we are calling. That way, even a simple-minded compiler has all the information needed to inline. On the other hand, a function used as a parameter is passed as a pointer, and optimizers have traditionally been incapable of performing optimizations involving pointers. This can be very significant (e.g. a factor of 50 in speed) when we pass an object or function that performs a really simple operation, such as the comparison criteria for a sort. In particular, inlining of function objects is the reason that the STL (C++ standard library) `sort()` outperforms the conventional `qsort()` by several factors when sorting arrays of types with simple comparison operators (such as `int` and `double`) [125]. Spurred on by the success of function objects, some compilers now can do inlining for pointers to functions as long as a constant is used in the call. For example, today, some compilers can inline calls to `compare` from within `qsort`:

```
bool compare(double* a, double* b) { /* ... */ }
// ...
qsort(p,max,sizeof(double),compare);
```

In 1994, no production C or C++ compiler could do that.

Function objects are the C++ mechanism for higher-order constructs. It is not the most elegant expression of high-order ideas, but it is surprisingly expressive and inherently efficient in the context of a general purpose language. To get the same efficiency of code (in time and space) from a general implementation of conventional functional programming facilities requires significant maturity from an optimizer. As an example of expressiveness, Jaakko Järvi and Gary Powell showed how to provide and use a lambda class that made the following example legal with its obvious meaning [72]:

```
list<int> lst;
// ...
Lambda x;
list<int>::iterator p =
   find_if(lst.begin(),lst.end(),x<7);
```

Note how overload resolution enables us to make the element type, `int`, implicit (deduced). If you want just `<` to work, rather than building a general library, you can add definitions for `Lambda` and `<` in less than a dozen lines of code. Using `Less_than` from the example above, we can simply write:

```
class Lambda {};

template<class T>
Less_than<T> operator<(Lambda,const T& v)
{
   return Less_than<T>(v);
}
```

So, the argument `x<7` in the call of `find_if` becomes a call of `operator<(Lambda,const int&)`, which generates a `Less_than<int>` object. That's exactly what we used explicitly in the first example in this section. The difference here is just that we have achieved a much simpler and more intuitive syntax. This is a good example of the expressive power of C++ and of how the interface to a library can be simpler than its implementation. Naturally, there is no run-time or space overhead compared to a laboriously written loop to look for an element with a value less than `7`.

The closest that C++ comes to higher-order functions is a function template that returns a function object, such as `operator<` returning a `Less_than` of the appropriate type and value. Several libraries have expanded that idea into quite comprehensive support for functional programming (e.g., Boost's Function objects and higher-order programming libraries [16] and FC++ [99]).

4.1.5 Traits

C++ doesn't offer a general compile-time way of asking for properties of a type. In the STL, and in many other libraries using templates to provide generic type-safe facilities for diverse types, this became a problem. Initially, the STL used overloading to deal with this (e.g., note the way the type `int` is deduced and used in `x<7`; §4.1.4). However, that use of overloading was unsystematic and therefore unduly difficult and error prone. The basic solution was discovered by Nathan Myers during the effort to templatize `iostream` and `string` [88]. The basic idea is to provide an auxiliary template, "a trait", to contain the desired information about a set of types. Consider how to find the type of the elements pointed to by an iterator. For a `list_iterator<T>` it is `list_iterator<T>::value_type` and for an ordinary pointer `T*` it is `T`. We can express that like this:

```
template<class Iter>
struct iterator_trait {
   typedef Iter::value_type value_type;
};
```

```
template<class T>
struct iterator_trait<T*> {
   typedef T value_type;
};
```

That is, the `value_type` of an iterator is its member type `value_type`. However, pointers are a common form of iterators and they don't have any member types. So, for pointers we use the type pointed to as `value_type`. The language construct involved is called partial specialization (added to C++ in 1995; §5). Traits can lead to somewhat bloated source code (though they have no object code or run-time cost) and even though the technique is very extensible it often requires explicit programmer attention when a new type is added. Traits are now ubiquitous in template-based C++ libraries. However, the "concepts" mechanism (§8.3.3) promises to make many traits redundant by providing direct language support for the idea of expressing properties of types.

4.1.6 Iterator categories

The result of the STL model as described so far could easily have become a mess with each algorithm depending on the peculiarities of the iterators offered by specific containers. To achieve interoperability, iterator interfaces have to be standardized. It would have been simplest to define a single set of operators for every iterator. However, to do so would have been doing violence to reality: From an algorithmic point of view lists, vectors, and output streams really do have different essential properties. For example, you can efficiently subscript a vector, you can add an element to a list without disturbing neighboring elements, and you can read from an input stream but not from an output stream. Consequently the STL provides a classification of iterators:

- input iterator (ideal for homogeneous stream input)
- output iterator (ideal for homogeneous stream output)
- forward iterator (we can read and write an element repeatedly, ideal for singly-linked lists)
- bidirectional iterator (ideal for doubly-linked lists)
- random access iterator (ideal for vectors and arrays)

This classification acts as a guide to programmers who care about interoperability of algorithms and containers. It allows us to minimize the coupling of algorithms and containers. Where different algorithms exist for different iterator categories, the most suitable algorithm is automatically chosen through overloading (at compile time).

4.1.7 Complexity requirements

The STL included complexity measures (using the big-O notation) for every standard library operation and algorithm. This was novel in the context of a foundation library for a language in major industrial use. The hope was and still is that this would set a precedent for better specification of libraries. Another — less innovative — aspect of this is a fairly systematic use of preconditions and postconditions in the specification of the library.

4.1.8 Stepanov's view

The description of the STL here is (naturally) focused on language and library issues in the context of C++. To get a complementary view, I asked Alexander Stepanov for his perspective [106]:

> In October of 1976 I observed that a certain algorithm — parallel reduction — was associated with monoids: collections of elements with an associative operation. That observation led me to believe that it is possible to associate every useful algorithm with a mathematical theory and that such association allows for both widest possible use and meaningful taxonomy. As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures. One seldom needs to know the exact type of data on which an algorithm works since most algorithms work on many similar types. In order to write an algorithm one needs only to know the properties of operations on data. I call a collection of types with similar properties on which an algorithm makes sense the *underlying concept* of the algorithm. Also, in order to pick an efficient algorithm one needs to know the complexity of these operations. In other words, complexity is an essential part of the interface to a concept.
>
> In the late '70s I became aware of John Backus's work on FP [7]. While his idea of programming with functional forms struck me as essential, I realized that his attempt to permanently fix the number of functional forms was fundamentally wrong. The number of functional forms — or, as I call them now, generic algorithms — is always growing as we discover new algorithms. In 1980 together with Dave Musser and Deepak Kapur I started working on a language Tecton to describe algorithms defined on algebraic theories. The language was functional since I did not realize at the time that memory and pointers were a fundamental part of programming. I also spent time studying Aristotle and his successors which led me to better understanding of fundamental operations on objects like equality and copying and the relation between whole and part.
>
> In 1984 I started collaborating with Aaron Kershenbaum who was an expert on graph algorithms. He was able to convince me to take arrays seriously. I viewed sequences as recursively definable since it was commonly perceived to be the "theoretically sound" approach. Aaron showed me that many fundamental algorithms depended on random access. We produced a

large set of components in Scheme and were able to implement generically some complicated graph algorithms.

The Scheme work led to a grant to produce a generic library in Ada. Dave Musser and I produced a generic library that dealt with linked structures. My attempts to implement algorithms that work on any sequential structure (both lists and arrays) failed because of the state of Ada compilers at the time. I had equivalences to many STL algorithms, but could not compile them. Based on this work, Dave Musser and I published a paper where we introduced the notion of generic programming insisting on deriving abstraction from useful efficient algorithms. The most important thing I learned from Ada was the value of static typing as a design tool. Bjarne Stroustrup had learned the same lesson from Simula.

In 1987 at Bell Labs Andy Koenig taught me semantics of C. The abstract machine behind C was a revelation. I also read lots of UNIX and Plan 9 code: Ken Thompson's and Rob Pike's code certainly influenced STL. In any case, in 1987 C++ was not ready for STL and I had to move on.

At that time I discovered the works of Euler and my perception of the nature of mathematics underwent a dramatic transformation. I was de-Bourbakized, stopped believing in sets, and was expelled from the Cantorian paradise. I still believe in abstraction, but now I know that one ends with abstraction, not starts with it. I learned that one has to adapt abstractions to reality and not the other way around. Mathematics stopped being a science of theories but reappeared to me as a science of numbers and shapes.

In 1993, after five years working on unrelated projects, I returned to generic programming. Andy Koenig suggested that I write a proposal for including my library into the C++ standard, Bjarne Stroustrup enthusiastically endorsed the proposal and in less than a year STL was accepted into the standard. STL is the result of 20 years of thinking but of less than two years of funding.

STL is only a limited success. While it became a widely used library, its central intuition did not get across. People confuse generic programming with using (and abusing) C++ templates. Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.

You can find references to the work leading to STL at `www.stepanovpapers.com`.

I am more optimistic about the long-term impact of Alex's ideas than he is. However, we agree that the STL is just the first step of a long journey.

4.1.9 The impact of the STL

The impact of the STL on the thinking on C++ has been immense. Before the STL, I consistently listed three fundamental programming styles ("paradigms") as being supported by C++ [113]:

- Procedural programming
- Data abstraction
- Object-oriented programming

I saw templates as support for data abstraction. After playing with the STL for a while, I factored out a fourth style:

- Generic programming

The techniques based on the use of templates and largely inspired by techniques from functional programming are qualitatively different from traditional data abstraction. People simply think differently about types, objects, and resources. New C++ libraries are written — using templates — to be statically type safe and efficient. Templates are the key to embedded systems programming and high-performance numeric programming where resource management and correctness are key [67]. The STL itself is not always ideal in those areas. For example, it doesn't provide direct support for linear algebra and it can be tricky to use in hard-real-time systems where free store use is banned. However, the STL demonstrates what can be done with templates and gives examples of effective techniques. For example, the use of iterators (and allocators) to separate logical memory access from actual memory access is key to many high-performance numeric techniques [86, 96] and the use of small, easily inlined, objects is key to examples of optimal use of hardware in embedded systems programming. Some of these techniques are documented in the standards committee's technical report on performance (§6.1). The emphasis on the STL and on generic programming in the C++ community in the late 1990s and early 2000s is to a large extent a reaction — and a constructive alternative — to a trend in the larger software-development community towards overuse of "object-oriented" techniques relying excessively on class hierarchies and virtual functions.

Obviously, the STL isn't perfect. There is no one "thing" to be perfect relative to. However, it broke new ground and has had impact even beyond the huge C++ community (§9.3). It also inspired many to use templates both in more disciplined and more adventurous ways. People talk about "template meta-programming" (§7.2.2) and generative programming [31] and try to push the techniques pioneered by the STL beyond the STL. Another line of attack is to consider how C++ could better support effective uses of templates (concepts, `auto`, etc.; see §8).

Inevitably, the success of STL brought its own problems. People wanted to write all kinds of code in the STL style. However, like any other style or technique, the STL style or even generic programming in general isn't ideal for every kind of problem. For example, generic programming relying on templates and overloading completely resolves all name bindings at compile time. It does not provide a mechanism for bindings that are resolved at run time; that's what class hierarchies and their associated object-oriented design techniques are for. Like all successful language mechanisms and programming techniques, templates and generic programming became fashionable and severely overused. Programmers built truly baroque and brittle constructs based on the fact that template instantiation and deduction is Turing complete. As I earlier observed for C++'s object-oriented facilities and techniques: "Just because you can do it, doesn't mean that you have to". Developing a comprehensive and simple framework for using the different programming styles supported by C++ is a major challenge for the next few years. As a style of programming, "multi-paradigm programming" [121] is underdeveloped. It often provides solutions that are more elegant and outperforms alternatives [28], but we don't (yet) have a simple and systematic way of combining the programming styles. Even its name gives away its fundamental weakness.

Another problem with the STL is that its containers are non-intrusive. From the point of view of code clarity and independence of concepts, being non-intrusive is a huge advantage. However, it does mean that we need to copy elements into containers or insert objects with default values into containers and later give them the desired values. Sometimes that's inefficient or inconvenient. For example, people tend not to insert large objects into a `vector` for that reason; instead, they insert pointers to such large objects. Similarly, the implicit memory management that the standard containers provide for their elements is a major convenience, but there are applications (e.g., in some embedded and high-performance systems) where such implicit memory management must be avoided. The standard containers provide features for ensuring that (e.g., `reserve`), but they have to be understood and used to avoid problems.

4.2 Other Parts of the Standard Library

From 1994 onwards, the STL dominated the work on the standard library and provided its major area of innovation. However, it was not the only area of work. In fact, the standard library provides several components:

- basic language run-time support (memory management, run-time type information (RTTI), exceptions, etc.)
- the C standard library
- the STL (containers, algorithms, iterators, function objects)
- `iostreams` (templatized on character type and implicitly on locale)
- `locales` (objects characterizing cultural preferences in I/O)
- `string` (templatized on character type)
- `bitset` (a set of bits with logical operations)
- `complex` (templatized on scalar type)
- `valarray` (templatized on scalar type)
- `auto_ptr` (a resource handle for objects templatized on type)

For a variety of reasons, the stories of the other library components are less interesting and less edifying than the story of the STL. Most of the time, work on each of these components progressed in isolation from work on the others. There was no overall design or design philosophy. For example, `bitset` is range checked whereas `string` isn't. Furthermore, the design of several components (such as `string`, `complex`, and `iostream`) was constrained by compatibility concerns. Several (notably `iostream` and `locale`) suffered from the "second-system effect" as their designers tried to cope with all kinds of demands, constraints, and existing practice. Basically, the committee failed to contain "design by committee" so whereas the STL reflects a clear philosophy and coherent style, most of the other components suffered. Each represents its own style and philosophy, and some (such as `string`) manage simultaneously to present several. I think `complex` is the exception here. It is basically my original design [91] templatized to allow for a variety of scalar types:

```
complex<double> z;     // double-precision
complex<float> x;      // single-precision
complex<short> point;  // integer grid
```

It is hard to seriously mess up math.

The committee did have some serious discussions about the scope of the standard library. The context for this discussion was the small and universally accepted C standard library (which the C++ standard adopted with only the tiniest of modification) and the huge corporate foundation libraries. During the early years of the standards process, I articulated a set of guidelines for the scope of the C++ standard library:

> First of all, the key libraries now in almost universal use must be standardized. This means that the exact interface between C++ and the C standard libraries must be specified and the iostreams library must be specified. In addition, the basic language support must be specified. ...
>
> Next, the committee must see if it can respond to the common demand for "more useful and standard classes," such as `string`, without getting into a mess of design by committee and without competing with the C++ library industry. Any libraries beyond the

> C libraries and iostreams accepted by the committee must be in the nature of building blocks rather than more ambitious frameworks. The key role of a standard library is to ease communication between separately developed, more ambitious libraries.

The last sentence delineated the scope of the committee's efforts. The elaboration of the requirement for the standard library to consist of building blocks for more ambitious libraries and frameworks emphasized absolute efficiency and extreme generality. One example I frequently used to illustrate the seriousness of those demands was that a container where element access involved a virtual function call could not be sufficiently efficient and that a container that couldn't hold an arbitrary type could not be sufficiently general (see also §4.1.1). The committee felt that the role of the standard library was to support rather than supplant other libraries.

Towards the end of the work on the 1998 standard, there was a general feeling in the committee that we hadn't done enough about libraries, and also that there had been insufficient experimentation and too little attention to performance measurement for the libraries we did approve. The question was how to address those issues in the future. One — classical standards process — approach was to work on technical reports (see §6.2). Another was initiated by Beman Dawes in 1998, called Boost [16]. To quote from boost.org (August 2006):

> "Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use. We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) as a step toward becoming part of a future C++ Standard. More Boost libraries are proposed for the upcoming TR2".

Boost thrived and became a significant source of libraries and ideas for the standards committee and the C++ community in general.

5. Language Features: 1991-1998

By 1991, the most significant C++ language features for C++98 had been accepted: templates and exceptions as specified in the ARM were officially part of the language. However, work on their detailed specification went on for another several years. In addition, the committee worked on many new features, such as

1992 Covariant return types — the first extension beyond the features described in the ARM

1993 Run-time type identification (RTTI: `dynamic_cast`, `typeid`, and `type_info`); §5.1.2

1993 Declarations in conditions; §5.1.3

1993 Overloading based on enumerations

1993 Namespaces; §5.1.1

1993 `mutable`

1993 New casts (`static_cast`, `reinterpret_cast`, and `const_cast`)

1993 A Boolean type (`bool`); §5.1.4

1993 Explicit template instantiation

1993 Explicit template argument specification in function template calls

1994 Member templates ("nested templates")

1994 Class templates as template arguments

1996 In-class member initializers

1996 Separate compilation of templates (`export`); §5.2

1996 Template partial specialization

1996 Partial ordering of overloaded function templates

I won't go into detail here; the history of these features can be found in D&E [121] and TC++PL3 [126] describes their use. Obviously, most of these features were proposed and discussed long before they were voted into the standard.

Did the committee have overall criteria for acceptance of new features? Not really. The introduction of classes, class hierarchies, templates, and exceptions each (and in combination) represented a deliberate attempt to change the way people think about programming and write code. Such a major change was part of my aims for C++. However, as far as a committee can be said to think, that doesn't seem to be the way it does it. Individuals bring forward proposals, and the ones that make progress through the committee and reach a vote tend to be of limited scope. The committee members are busy and primarily practical people with little patience for abstract goals and a liking of concrete details that are amenable to exhaustive examination.

It is my opinion that the sum of the facilities added gives a more complete and effective support of the programming styles supported by C++, so we could say that the overall aim of these proposals is to "provide better support for procedural, object-oriented, and generic programming and for data abstraction". That's true, but it is not a concrete criterion that can be used to select proposals to work on from a long list. To the extent that the process has been successful in selecting new "minor features", it has been the result of decisions by individuals on a proposal-by-proposal basis. That's not my ideal, but the result could have been much worse. ISO C++ (C++98) is a better approximation to my ideals than the previous versions of C++ were. C++98 is a far more flexible (powerful) programming language than "ARM C++" (§3).

The main reason for that is the cumulative effect of the refinements, such as member templates.

Not every feature accepted is in my opinion an improvement, though. For example, "in-class initialization of static const members of integral type with a constant expression" (proposed by John "Max" Skaller representing Australia and New Zealand) and the rule that `void f(T)` and `void f(const T)` denote the same function (proposed by Tom Plum for C compatibility reasons) share the dubious distinction of having been voted into C++ "over my dead body".

5.1 Some "Minor Features"

The "minor features" didn't feel minor when the committee worked on them, and may very well not look minor to a programmer using them. For example, I refer to namespaces and RTTI as "major" in D&E [121]. However, they don't significantly change the way we think about programs, so I will only briefly discuss a few of the features that many would deem "not minor".

5.1.1 Namespaces

C provides a single global namespace for all names that don't conveniently fit into a single function, a single `struct`, or a single translation unit. This causes problems with name clashes. I first grappled with this problem in the original design of C++ by defaulting all names to be local to a translation unit and requiring an explicit `extern` declaration to make them visible to other translation units. This idea was neither sufficient to solve the problem nor sufficiently compatible to be acceptable, so it failed.

When I devised the type-safe linkage mechanism [121], I reconsidered the problem. I observed that a slight change to the

```
extern "C" { /* ... */ }
```

syntax, semantics, and implementation technique would allow us to have

```
extern XXX { /* ... */ }
```

mean that names declared in `XXX` were in a separate scope `XXX` and accessible from other scopes only when qualified by `XXX::` in exactly the same way static class members are accessed from outside their class.

For various reasons, mostly related to lack of time, this idea lay dormant until it resurfaced in the ANSI/ISO committee discussions early in 1991. First, Keith Rowe from Microsoft presented a proposal that suggested the notation

```
bundle XXX { /* ... */ };
```

as a mechanism for defining a named scope and an operator `use` for bringing all names from a `bundle` into another scope. This led to a — not very vigorous — discussion among a few members of the extensions group including Steve Dovich, Dag Brück, Martin O'Riordan, and me. Eventually, Volker Bauche, Roland Hartinger, and Erwin Unruh from Siemens refined the ideas discussed into a proposal that didn't use new keywords:

```
:: XXX :: { /* ... */ };
```

This led to a serious discussion in the extensions group. In particular, Martin O'Riordan demonstrated that this `::` notation led to ambiguities with `::` used for class members and for global names.

By early 1993, I had — with the help of multi-megabyte email exchanges and discussions at the standards meetings — synthesized a coherent proposal. I recall technical contributions on namespaces from Dag Brück, John Bruns, Steve Dovich, Bill Gibbons, Philippe Gautron, Tony Hansen, Peter Juhl, Andrew Koenig, Eric Krohn, Doug McIlroy, Richard Minner, Martin O'Riordan, John "Max" Skaller, Jerry Schwarz, Mark Terribile, Mike Vilot, and me. In addition, Mike Vilot argued for immediate development of the ideas into a definite proposal so that the facilities would be available for addressing the inevitable naming problems in the ISO C++ library. In addition to various common C and C++ techniques for limiting the damage of name clashes, the facilities offered by Modula-2 and Ada were discussed. Namespaces were voted into C++ at the Munich meeting in July 1993. So, we can write:

```
namespace XXX {
   // ...
   int f(int);
}

int f(int);
int x = f(1);        // call global f
int y = XXX::f(1);   // call XXX's f
```

At the San Jose meeting in November 1993, it was decided to use namespaces to control names in the standard C and C++ libraries.

The original `namespace` design included a few more facilities, such as namespace aliases to allow abbreviations for long names, `using` declarations to bring individual names into a namespace, `using` directives to make all names from a namespace available with a single directive. Three years later, argument-dependent lookup (ADL or "Koenig lookup") was added to make namespaces of argument type names implicit.

The result was a facility that is useful and used but rarely loved. Namespaces do what they are supposed to do, sometimes elegantly, sometimes clumsily, and sometimes they do more than some people would prefer (especially argument-dependent lookup during template instantiation). The fact that the C++ standard library uses only a single namespace for all of its major facilities is an indication of a failure to establish namespaces as a primary tool of C++ programmers. Using sub-namespaces for the standard library would have implied a standardization of parts of the library implementation (to say which facilities were in which namespaces

and which parts depended on other parts). Some library vendors strongly objected to such constraints on their traditional freedom as implementers — traditionally the internal organization of C and C++ libraries have been essentially unconstrained. Using sub-namespaces would also have been a source of verbosity. Argument-dependent lookup would have helped, but it was only introduced later in the standardization process. Also, ADL suffers from a bad interaction with templates that in some cases make it prefer a surprising template over an obvious non-template. Here "surprising" and "obvious" are polite renderings of user comments.

This has led to proposals for C++0x to strengthen namespaces, to restrict their use, and most interestingly a proposal from David Vandevoorde from EDG to make some namespaces into modules [146] — that is, to provide separately compiled namespaces that load as modules. Obviously, that facility looks a bit like the equivalent features of Java and C#.

5.1.2 Run-time type information

When designing C++, I had left out facilities for determining the type of an object (Simula's `QUA` and `INSPECT` similar to Smalltalk's `isKindOf` and `isA`). The reason was that I had observed frequent and serious misuse to the great detriment of program organization: people were using these facilities to implement (slow and ugly) versions of a switch statement.

The original impetus for adding facilities for determining the type of an object at run time to C++ came from Dmitry Lenkov from Hewlett-Packard. Dmitry in turn built on experience from major C++ libraries such as Interviews [81], the NIH library [50], and ET++ [152]. The RTTI mechanisms provided by these libraries (and others) were mutually incompatible, so they became a barrier to the use of more than one library. Also, all require considerable foresight from base class designers. Consequently, a language-supported mechanism was needed.

I got involved in the detailed design for such mechanisms as the coauthor with Dmitry of the original proposal to the committee and as the main person responsible for the refinement of the proposal in the committee [119]. The proposal was first presented to the committee at the London meeting in July 1991 and accepted at the Portland, Oregon meeting in March 1993.

The run-time type information mechanism consists of three parts:

- An operator, `dynamic_cast`, for obtaining a pointer to an object of a derived class given a pointer to a base class of that object. The operator `dynamic_cast` delivers that pointer only if the object pointed to really is of the specified derived class; otherwise it returns `0`.
- An operator, `typeid`, for identifying the exact type of an object given a pointer to a base class.
- A structure, `type_info`, acting as a hook for further run-time information associated with a type.

Assume that a library supplies class `dialog_box` and that its interfaces are expressed in terms of `dialog_box`es. I, however, use both `dialog_box`es and my own `Sbox`s:

```
class dialog_box : public window {
   // library class known to ``the system''
public:
   virtual int ask();
   // ...
};

class Sbox : public dialog_box {
   // can be used to communicate a string
public:
   int ask();
   virtual char* get_string();
   // ...
};
```

So, when the system/library hands me a pointer to a `dialog_box`, how can I know whether it is one of my `Sbox`s? Note that I can't modify the library to know my `Sbox` class. Even if I could, I wouldn't, because then I would have to modify every new version of the library forever after. So, when the system passes an object to my code, I sometimes need to ask it if was "one of mine". This question can be asked directly using the `dynamic_cast` operator:

```
void my_fct(dialog_box* bp)
{
   if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
      // use sbp
   }
   else {
      // treat *pb as a ``plain'' dialog box
   }
}
```

The `dynamic_cast<T*>(p)` converts `p` to the desired type `T*` if `*p` really is a `T` or a class derived from `T`; otherwise, the value of `dynamic_cast<T*>(p)` is `0`. This use of `dynamic_cast` is the essential operation of a GUI callback. Thus, C++'s RTTI can be seen as the minimal facility for supporting a GUI.

If you don't want to test explicitly, you can use references instead of pointers:

```
void my_fct(dialog_box& br)
{
   Sbox& sbr = dynamic_cast<Sbox&>(br);
     // use sbr
}
```

Now, if the `dialog_box` isn't of the expected type, an exception is thrown. Error handling can then be elsewhere (§5.3).

Obviously, this run-time type information is minimal. This has led to requests for the maximal facility: a full

meta-data facility (reflection). So far, this has been deemed unsuitable for a programming language that among other things is supposed to leave its applications with a minimal memory footprint.

5.1.3 Declarations in conditions

Note the way the cast, the declaration, and the test were combined in the "box example":

```
if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
   // use sbp
}
```

This makes a neat logical entity that minimizes the chance of forgetting to test, minimizes the chance of forgetting to initialize, and limits the scope of the variable to its minimum. For example, the scope of `dbp` is the `if`-statement.

The facility is called "declaration in condition" and mirrors the "declaration as `for`-statement initializer", and "declaration as statement". The whole idea of allowing declarations everywhere was inspired by the elegant statements-as-expressions definition of Algol68 [154]. I was therefore most amazed when Charles Lindsey explained to me at the HOPL-II conference that Algol68 for technical reasons had not allowed declarations in conditions.

5.1.4 Booleans

Some extensions really are minor, but the discussions about them in the C++ community are not. Consider one of the most common enumerations:

```
enum bool { false, true };
```

Every major program has that one or one of its cousins:

```
#define bool char
#define Bool int
typedef unsigned int BOOL;
typedef enum { F, T } Boolean;
const true = 1;
#define TRUE 1
#define False (!True)
```

The variations are apparently endless. Worse, most variations imply slight variations in semantics, and most clash with other variations when used together.

Naturally, this problem has been well known for years. Dag Brück (representing Ericsson and Sweden) and Andrew Koenig (AT&T) decided to do something about it: "The idea of a Boolean data type in C++ is a religious issue. Some people, particularly those coming from Pascal or Algol, consider it absurd that C should lack such a type, let alone C++. Others, particularly those coming from C, consider it absurd that anyone would bother to add such a type to C++"

Naturally, the first idea was to define an `enum`. However, Dag Brück and Sean Corfield (UK) examined hundreds of thousands of lines of C++ and found that most Boolean types were used in ways that required implicit conversion of `bool` to and from `int`. C++ does not provide implicit conversion of `int`s to enumerations, so defining a standard `bool` as an enumeration would break too much existing code. So why bother with a Boolean type?

- The Boolean data type is a fact of life whether it is a part of a C++ standard or not.
- The many clashing definitions make it hard to use *any* Boolean type conveniently and safely.
- Many people want to overload based on a Boolean type.

Somewhat to my surprise, the committee accepted this argument, so `bool` is now a distinct integral type in C++ with literals `true` and `false`. Non-zero values can be implicitly converted to `true`, and `true` can be implicitly converted to `1`. Zero can be implicitly converted to `false`, and `false` can be implicitly converted to `0`. This ensures a high degree of compatibility.

Over the years, `bool` proved popular. Unexpectedly, I found it useful in teaching C++ to people without previous programming experience. After `bool`'s success in C++, the C standards committee decided to also add it to C. Unfortunately, they decided to do so in a different and incompatible way, so in C99 [64], `bool` is a `macro` for the keyword `_Bool` defined in the header `<stdbool.h>` together with macros `true` and `false`.

5.2 The Export Controversy

From the earliest designs, templates were intended to allow a template to be used after specifying just a declaration in a translation unit [117, 35]. For example:

```
template<class In, class T>
In find(In, In, const T&);   // no function body

vector<int>::iterator p =
   find(vi.begin(), vi.end(),42);
```

It is then the job of the compiler and linker to find and use the definition of the `find` template (D&E §15.10.4). That's the way it is for other language constructs, such as functions, but for templates that's easily said but extremely hard to do.

The first implementation of templates, Cfront 3.0 (October 1991), implemented this, but in a way that was very expensive in both compile time and link time. However, when Taumetric and Borland implemented templates, they introduced the "include everything" model: Just place all template definitions in header files and the compiler plus linker will eliminate the multiple definitions you get when you include a file multiple times in separately compiled translation units. The First Borland compiler with "rudimentary template support" shipped November 20, 1991, quickly followed by version 3.1 and the much more robust version 4.0 in November 1993 [27]. Microsoft, Sun, and others followed along with (mutually incompatible) variations of the "include everything" approach. Obviously, this approach violates the usual separation between an implementation (using definitions) and an interface (presenting only declarations)

and makes definitions vulnerable to macros, unintentional overload resolution, etc. Consider a slightly contrived example:

```
// printer.h:
   template<class Destination>
   class Printer {
      locale loc;
      Destination des;
   public:
      template<class T> void out(const T& x)
         { print(des,x,loc); }
      // ...
   };
```

We might use `Printer` in two translation units like this:

```
//user1.c:
  typedef int locale; // represent locale by int
  #define print(a,b,c) a(c)<<x
  #include "printer.h"
  // ...
```

and this

```
//user2.c:
  #include<locale>  // use standard locale
  using namespace std;
  #include "printer.h"
  // ...
```

This is obviously illegal because differences in the contexts in which `printer.h` are seen in `user1.c` and `user2.c` lead to inconsistent definitions of `Printer`. However, such errors are hard for a compiler to detect. Unfortunately, the probability of this kind of error is far higher in C++ than in C and even higher in C++ using templates than in C++ that doesn't use templates. The reason is that when we use templates there is so much more text in header files for typedefs, overloads, and macros to interfere with. This leads to defensive programming practices (such as naming all local names in template definitions in a cryptic style unlikely to clash, e.g., `_L2`) and a desire to reduce the amount of code in the header files through separate compilation.

In 1996, a vigorous debate erupted in the committee over whether we should not just accept the "include everything" model for template definitions, but actually outlaw the original model of separation of template declarations and definitions into separate translation units. The arguments of the two sides were basically

- Separate translation of templates is too hard (if not impossible) and such a burden should not be imposed on implementers
- Separate translation of templates is necessary for proper code organization (according to data-hiding principles)

Many subsidiary arguments supported both sides. Mike Ball (Sun) and John Spicer (EDG) led the "ban separate compilation of templates" group and Dag Bruck (Ericcson and Sweden) usually spoke for the "preserve separate compilation of templates" group. I was on the side that insisted on separate compilation of templates. As ever in really nasty discussions, both sides were mostly correct on their key points. In the end, people from SGI — notably John Wilkinson — proposed a new model that was accepted as a compromise. The compromise was named after the keyword used to indicate that a template could be separately translated: `export`.

The separate compilation of templates issue festers to this day: The "export" feature remains disabled even in some compilers that do support it because enabling it would break ABIs. As late as 2003, Herb Sutter (representing Microsoft) and Tom Plum (of Plum Hall) proposed a change to the standard so that an implementation that didn't implement separate compilation of templates would still be conforming; that is, `export` would be an optional language feature. The reason given was again implementation complexity plus the fact that even five years after the standard was ratified only one implementation existed. That motion was defeated by an 80% majority, partly because an implementation of `export` now existed. Independently of the technical arguments, many committee members considered it unfair to deem a feature optional after some, but not all, implementers had spent significant time and effort implementing it.

The real heroes of this sad tale are the implementers of the EDG compiler: Steve Adamczyk, John Spicer, and David Vandevoorde. They strongly opposed separate compilation of templates, finally voted for the standard as the best compromise attainable, and then proceeded to spend more than a year implementing what they had opposed. That's professionalism! The implementation was every bit as difficult as its opponents had predicted, but it worked and provided some (though not all) of the benefits that its proponents had promised. Unfortunately, some of the restrictions on separately compiled templates that proved essential for the compromise ended up not providing their expected benefits and complicated the implementation. As ever, political compromises on technical issues led to "warts".

I suspect that one major component of a better solution to the separate compilation of templates is concepts (§8.3.3) and another is David Vandevoorde's modules [146].

5.3 Exception Safety

During the effort to specify the STL we encountered a curious phenomenon: We didn't quite know how to talk about the interaction between templates and exceptions. Quite a few people were placing blame for this problem on templates and others began to consider exceptions fundamentally flawed (e.g., [20]) or at least fundamentally flawed in the absence of automatic garbage collection. However, when a group of "library people" (notably Nathan Myers, Greg Colvin, and Dave Abrahams) looked into this problem, they found that we basically had a language feature — exceptions — that we didn't know how to use well. The problem was in the interaction between resources and exceptions (or more generally, the interaction between invariants and

exceptions). If throwing an exception renders resources inaccessible there is no hope of recovering gracefully. I had of course considered this when I designed the exception-handling mechanisms and come up with the rules for exceptions thrown from constructors (correctly handling partially constructed composite objects) and the "resource acquisition is initialization" technique (§5.3.1). However, that was only a good start and an essential foundation. What we needed was a conceptual framework — a more systematic way of thinking about resource management. Together with many other people, notably Matt Austern, such a framework was developed.

The key was the three guarantees[1] that Dave Abrahams had formulated in early 1997:

- The *basic guarantee*: that the invariants of the component are preserved, and no resources are leaked.
- The *strong guarantee*: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The *no-throw guarantee*: that the operation will not throw an exception.

Note that the strong guarantee basically is the database "commit or rollback" rule. Using these fundamental concepts, the library working group described the standard library and implementers produced efficient and robust implementations. The standard library provides the basic guarantee for all operations with the caveat that we may not exit a destructor by throwing an exception. In addition, the library provides the strong guarantee and the no-throw guarantee for key operations. I found this result important enough to add an appendix to TC++PL [124], yielding [126]. For details of the standard library exception guarantees and programming techniques for using exceptions, see Appendix E of TC++PL.

The first implementations of the STL using these concepts to achieve exception safety were Matt Austern's SGI STL and Boris Fomitchev's STLPort [42] augmented with Dave Abrahams' exception-safe implementations of standard containers and algorithms. They appeared in the spring of 1997.

I think the key lesson here is that it is not sufficient just to know how a language feature behaves. To write good software, we must have a clearly articulated design strategy for problems that require the use of the feature.

5.3.1 Resource management

Exceptions are typically — and correctly — seen as a control structure: a `throw` transfers control to some `catch`-clause. However, sequencing of operations is only part of the picture: error handling using exceptions is mostly about resource management and invariants. This view is actually built into the C++ class and exception primitives in a way that provides a necessary foundation for the guarantees and the standard library design.

When an exception is thrown, every constructed object in the path from the `throw` to the `catch` is destroyed. The destructors for partially constructed objects (and unconstructed objects) are not invoked. Without those two rules, exception handling would be unmanageable (in the absence of other support). I (clumsily) named the basic technique "resource acquisition is initialization" — commonly abbreviated to "RAII". The classical example [118] is

```
// naive and unsafe code:
void use_file(const char* fn)
{
   FILE* f = fopen(fn,"w");  // open file fn
   // use f
   fclose(f);  // close file fn
}
```

This looks plausible. However, if something goes wrong after the call of `fopen` and before the call of `fclose`, an exception may cause `use_file` to be exited without calling `fclose`. Please note that exactly the same problem can occur in languages that do not support exception handling. For example, a call of the standard C library function `longjmp` would have the same bad effects. Even a misguided `return` among the code using `f` would cause the program to leak a file handle. If we want to support writing fault-tolerant systems, we must solve this problem.

The general solution is to represent a resource (here the file handle) as an object of some class. The class' constructor acquires the resource and the class' destructor gives it back. For example, we can define a class `File_ptr` that acts like a `FILE*`:

```
class File_ptr {
   FILE* p;
public:
   File_ptr(const char* n, const char* a)
   {
      p = fopen(n,a);
      if (p==0) throw Failed_to_open(n);
   }
   ~File_ptr() { fclose(p); }
   // copy, etc.
   operator FILE*() { return p; }
};
```

We can construct a `File_ptr` given the arguments required for `fopen`. The `File_ptr` will be destroyed at the end of its scope and its destructor closes the file. Our program now shrinks to this minimum

```
void use_file(const char* fn)
{
   File_ptr f(fn,"r");  // open file fn
   // use f
} // file fn implicitly closed
```

The destructor will be called independently of whether the function is exited normally or because an exception is thrown.

The general form of the problem looks like this:

```
void use()
{
   // acquire resource 1
   // ...
   // acquire resource n

   // use resources

   // release resource n
   // ...
   // release resource 1
}
```

This applies to any "resource" where a resource is anything you acquire and have to release (hand back) for a system to work correctly. Examples include files, locks, memory, iostream states, and network connections. Please note that automatic garbage collection is *not* a substitute for this: the point in time of the release of a resource is often important logically and/or performance-wise.

This is a systematic approach to resource management with the important property that correct code is shorter and less complex than faulty and primitive approaches. The programmer does not have to remember to do anything to release a resource. This contrasts with the older and ever-popular `finally` approach where a programmer provides a `try`-block with code to release the resource. The C++ variant of that solution looks like this:

```
void use_file(const char* fn)
{
   FILE* f = fopen(fn,"r");  // open file fn
   try {
      // use f
   }
   catch (...) {   // catch all
      fclose(f);   // close file fn
      throw;       // re-throw
   }
   fclose(f);      // close file fn
}
```

The problem with this solution is that it is verbose, tedious, and potentially expensive. It can be made less verbose and tedious by providing a `finally` clause in languages such as Java, C#, and earlier languages [92]. However, shortening such code doesn't address the fundamental problem that the programmer has to remember to write release code for each acquisition of a resource rather just once for each resource, as in the RAII approach. The introduction of exceptions into the ARM and their presentation as a conference paper [79] was delayed for about half a year until I found "resource acquisition is initialization" as a systematic and less error-prone alternative to the `finally` approach.

Note that the RAII technique — like other powerful C++ idioms, such as smart pointers — rests on the use of destructors. Destructors were among the very first features introduced into C with Classes in 1980 (§2.1). Their introduction was explicitly motivated by concerns about resource management: constructors acquire resources and initialize; destructors are the inverse operations: they release resources and clean up. Because release and cleanup must be guaranteed, destructors are implicitly called for objects allocated on the stack (and for statically allocated variables). Thus, the roots of modern C++ resource management lie in the first version of C with Classes and ultimately in my earlier work on operating systems [121].

5.4 Automatic Garbage Collection

Sometime in 1995, it dawned on me that a majority of the committee was of the opinion that plugging a garbage collector into a C++ program was not standard-conforming because the collector would inevitably perform some action that violated a standard rule. Worse, they were obviously right about the broken rules. For example:

```
void f()
{
   int* p = new int[100];
   // fill *p with valuable data
   file << p;  // write the pointer to a file
   p = 0;      // remove the pointer to the ints
   // work on something else for a week
   file >> p;
   if (p[37] == 5) {        // now use the ints
     // ...
   }
}
```

My opinion — as expressed orally and in print — was roughly: "such programs deserve to be broken" and "it is perfectly good C++ to use a conservative garbage collector". However, that wasn't what the draft standard said. A garbage collector would undoubtedly have recycled that memory before we read the pointer back from the file and started using the integer array again. However, in standard C and standard C++, there is absolutely nothing that allows a piece of memory to be recycled without some explicit programmer action.

To fix this problem, I made a proposal to explicitly allow "optional automatic garbage collection" [123]. This would bring C++ back to what I had thought I had defined it to be and make the garbage collectors already in actual use [11, 47] standard conforming. Explicitly mentioning this in the standard would also encourage use of GC where appropriate. Unfortunately, I seriously underestimated the dislike of garbage collection in a large section of the committee and also mishandled the proposal.

My fatal mistake with the GC proposal was to get thoroughly confused about the meaning of "optional". Did "optional" mean that an implementation didn't have to provide a garbage collector? Did it mean that the programmer could

decide whether the garbage collector was turned on or not? Was the choice made at compile time or run time? What should happen if I required the garbage collector to be activated and the implementation didn't supply one? Can I ask if the garbage collector is running? How? How can I make sure that the garbage collector isn't running during a critical operation? By the time a confused discussion of such questions had broken out and different people had found conflicting answers attractive, the proposal was effectively dead.

Realistically, garbage collection wouldn't have passed in 1995, even if I hadn't gotten confused. Parts of the committee

- strongly distrusted GC for performance reasons
- disliked GC because it was seen as a C incompatibility
- didn't feel they understood the implications of accepting GC (we didn't)
- didn't want to build a garbage collector
- didn't want to pay for a garbage collector (in terms of money, space, or time)
- wanted alternative styles of GC
- didn't want to spend precious committee time on GC

Basically, it was too late in the standards process to introduce something that major. To get anything involving garbage collection accepted, I should have started a year earlier.

My proposal for garbage collection reflected the then major use of garbage collection in C++ — that is, conservative collectors that don't make assumptions about which memory locations contain pointers and never move objects around in memory [11]. Alternative approaches included creating a type-safe subset of C++ so that it is possible to know exactly where every pointer is, using smart pointers [34] and providing a separate operator (`gcnew` or `new(gc)`) for allocating objects on a "garbage-collected heap". All three approaches are feasible, provide distinct benefits, and have proponents. This further complicates any effort to standardize garbage collection for C++.

A common question over the years has been: Why don't you add GC to C++? Often, the implication (or follow-up comment) is that the C++ committee must be a bunch of ignorant dinosaurs not to have done so already. First, I observe that in my considered opinion, C++ would have been stillborn had it relied on garbage collection when it was first designed. The overheads of garbage collection at the time, on the hardware available, precluded the use of garbage collection in the hardware-near and performance-critical areas that were C++'s bread and butter. There were garbage-collected languages then, such as Lisp and Smalltalk, and people were reasonably happy with those for the applications for which they were suitable. It was not my aim to replace those languages in their established application areas. The aim of C++ was to make object-oriented and data-abstraction techniques affordable in areas where these techniques at the time were "known" to be impractical. The core areas of C++ usage involved tasks, such as device drivers, high-performance computation, and hard-real-time tasks, where garbage collection was (and is) either infeasible or not of much use.

Once C++ was established without garbage collection and with a set of language features that made garbage collection difficult (pointers, casts, unions, etc.), it was hard to retrofit it without doing major damage. Also, C++ provides features that make garbage collection unnecessary in many areas (scoped objects, destructors, facilities for defining containers and smart pointers, etc.). That makes the case for garbage collection less compelling.

So, why would I like to see garbage collection supported in C++? The practical reason is that many people write software that uses the free store in an undisciplined manner. In a program with hundreds of thousands of lines of code with `news` and `deletes` all over the place, I see no hope for avoiding memory leaks and access through invalid pointers. My main advice to people who are starting a project is simply: "don't do that!". It is fairly easy to write correct and efficient C++ code that avoids those problems through the use of containers (STL or others; §4.1), resource handles (§5.3.1, and (if needed) smart pointers (§6.2). However, many of us have to deal with older code that does deal with memory in an undisciplined way, and for such code plugging in a conservative garbage collector is often the best option. I expect C++0x to require every C++ implementation to be shipped with a garbage collector that, somehow, can be either active or not.

The other reason that I suspect a garbage collector will eventually become necessary is that I don't see how to achieve perfect type safety without one — at least not without pervasive testing of pointer validity or damaging compatibility (e.g. by using two-word non-local pointers). And improving type safety (e.g., "eliminate every implicit type violation" [121]) has always been a fundamental long-term aim of C++. Obviously, this is a very different argument from the usual "when you have a garbage collector, programming is easy because you don't have to think about deallocation". To contrast, my view can be summarized as "C++ is such a good garbage-collected language because it creates so little garbage that needs to be collected". Much of the thinking about C++ has been focused on resources in general (such as locks, file handles, thread handles, and free store memory). As noted in §5.3, this focus has left traces in the language itself, in the standard library, and in programming techniques. For large systems, for embedded systems, and for safety-critical systems a systematic treatment of resources seems to me much more promising than a focus on garbage collection.

5.5 What Wasn't Done

Choosing what to work on is probably more significant than how that work is done. If someone — in this case the C++ standards committee — decides to work on the wrong prob-

lem, the quality of the work done is largely irrelevant. Given the limited time and resources, the committee could cope with only a few topics and choose to work hard on those few rather than take on more topics. By and large, I think that the committee chose well and the proof of that is that C++98 is a significantly better language than ARM C++. Naturally, we could have done better still, but even in retrospect it is hard to know how. The decisions made at the time were taken with as much information (about problems, resources, possible solutions, commercial realities, etc.) as we had available then and who are we to second guess today?

Some questions are obvious, though:

- Why didn't we add support for concurrency?
- Why didn't we provide a much more useful library?
- Why didn't we provide a GUI?

All three questions were seriously considered and in the first two cases settled by explicit vote. The votes were close to unanimous. Given that we had decided not to pursue concurrency or to provide a significantly larger library, the question about a GUI library was moot.

Many of us — probably most of the committee members — would have liked some sort of concurrency support. Concurrency is fundamental and important. Among other suggestions, we even had a pretty solid proposal for concurrency support in the form of micro-C++ from the University of Toronto [18]. Some of us, notably Dag Brück relying on data from Ericssson, looked at the issue and presented the case for *not* dealing with concurrency in the committee:

- We found the set of alternative ways of supporting concurrency large and bewildering.
- Different application areas apparently had different needs and definitely had different traditions.
- The experience with Ada's direct language support for concurrency was discouraging.
- Much (though of course not all) could be done with libraries.
- The committee lacked sufficient experience to do a solid design.
- We didn't want a set of primitives that favored one approach to concurrency over others.
- We estimated that the work — if feasible at all — would take years given our scarce resources.
- We wouldn't be able to decide what other ideas for improvements to drop to make room for the concurrency work.

My estimate at the time was that "concurrency is as big a topic as all of the other extensions we are considering put together". I do not recall hearing what I in retrospect think would have been "the killer argument": Any sufficient concurrency support will involve the operating system; since C++ is a systems programming language, we need to be able to map the C++ concurrency facilities to the primitives offered. But for each platform the owners insist that C++ programmers can use every facility offered. In addition, as a fundamental part of a multi-language environment, C++ cannot rely on a concurrency model that is dramatically different from what other languages support. The resulting design problem is so constrained that it has no solution.

The reason the lack of concurrency support didn't hurt the C++ community more than it did is that much of what people actually do with concurrency is pretty mundane and can be done through library support and/or minor (non-standard) language extensions. The various threads libraries (§8.6) and MPI [94] [49] offer examples.

Today, the tradeoffs appear to be different: The continuing increase in gate counts paired with the lack of increase of hardware clock speeds is a strong incentive to exploit low-level concurrency. In addition, the growth of multiprocessors and clusters requires other styles of concurrency support and the growth of wide-area networking and the web makes yet other styles of concurrent systems essential. The challenge of supporting concurrency is more urgent than ever and the major players in the C++ world seem far more open to the need for changes to accommodate it. The work on C++0x reflects that (§8.2).

The answer to "Why didn't we provide a much more useful library?" is simpler: We didn't have the resources (time and people) to do significantly more than we did. Even the STL caused a year's delay and gaining consensus on other components, such as `iostreams`, strained the committee. The obvious shortcut — adopting a commercial foundation library — was considered. In 1992, Texas Instruments offered their very nice library for consideration and within an hour five representatives of major corporations made it perfectly clear that if this offer was seriously considered they would propose their own corporate foundation libraries. This wasn't a way that the committee could go. Another committee with a less consensus-based culture might have made progress by choosing one commercial library over half-a-dozen others in major use, but not this C++ committee.

It should also be remembered that in 1994, many already considered the C++ standard library monstrously large. Java had not yet changed programmers' perspective of what they could expect "for free" from a language. Instead, many in the C++ community used the tiny C standard library as the measure of size. Some national bodies (notably the Netherlands and France) repeatedly expressed worry that C++ standard library was seriously bloated. Like many in the committee, I also hoped that the standard would help rather than try to supplant the C++ libraries industry.

Given those general concerns about libraries, the answer to "Why didn't we provide a GUI?" is obvious: The committee couldn't do it. Even before tackling the GUI-specific design issues, the committee would have had to tackle con-

currency and settle on a container design. In addition, many of the members were simply blindsided. GUI was seen as just another large and complex library that people — given `dynamic_cast` (§5.1.2) — could write themselves (in particular, that was my view). They did. The problem today is not that there is no C++ GUI library, but that there are on the order of 25 such libraries in use (e.g., Gtkmm [161], FLTK [156], SmartWin++ [159], MFC [158], WTL [160], wxWidgets (formerly wxWindows) [162], Qt [10]). The committee could have worked on a GUI library, worked on library facilities that could be used as a basis for GUI libraries, or worked on standard library interfaces to common GUI functionality. The latter two approaches might have yielded important results, but those paths weren't taken and I don't think that the committee then had the talents necessary for success in that direction. Instead, the committee worked hard on more elaborate interfaces to stream I/O. That was probably a dead end because the facilities for multiple character sets and locale dependencies were not primarily useful in the context of traditional data streams.

6. Standards Maintenance: 1997-2005

After a standard is passed, the ISO process can go into a "maintenance mode" for up to five years. The C++ committee decided to do that because:

- the members were tired (after ten years' work for some members) and wanted to do something else,
- the community was way behind in understanding the new features,
- many implemeters were way behind with the new features, libraries, and support tools,
- no great new ideas were creating a feeling of urgency among the members or in the community,
- for many members, resources (time and money) for standardization were low, and
- many members (including me) thought that the ISO process required a "cooling-off period".

In "maintenance mode," the committee primarily responded to defect reports. Most defects were resolved by clarifying the text or resolving contradictions. Only very rarely were new rules introduced and real innovation was avoided. Stability was the aim. In 2003, all these minor corrections were published under the name "Technical Corrigendum 1". At the same time, members of the British national committee took the opportunity to remedy a long-standing problem: they convinced Wiley to publish a printed version of the (revised) standard [66]. The initiative and much of the hard work came from Francis Glassborow and Lois Goldthwaite with technical support from the committee's project editor, Andrew Koenig, who produced the actual text.

Until the publication of the revised standard in 2003, the only copies of the standard available to the public were a very expensive (about $200) paper copy from ISO or a cheap ($18) pdf version from INCITS (formerly ANSI X3). The pdf version was a complete novelty at the time. Standards bodies are partially financed through the sales of standards, so they are most reluctant to make them available free of charge or cheaply. In addition, they don't have retail sales channels, so you can't find a national or international standard in your local book store — except the C++ standard, of course. Following the C++ initiative, the C standard is now also available.

Much of the best standards work is invisible to the average programmer and appears quite esoteric and often boring when presented. The reason is that a lot of effort is expended in finding ways of expressing clearly and completely "what everyone already knows, but just happens not to be spelled out in the manual" and in resolving obscure issues that — at least in theory — don't affect most programmers. The maintenance is mostly such "boring and esoteric" issues. Furthermore, the committee necessarily focuses on issues where the standard contradicts itself — or appears to do so. However, these issues are essential to implementers trying to ensure that a given language use is correctly handled. In turn, these issues become essential to programmers because even the most carefully written large program will deliberately or accidentally depend on some feature that would appear obscure or esoteric to some. Unless implementers agree, the programmer has a hard time achieving portability and easily becomes the hostage of a single compiler purveyor — and that would be contrary to my view of what C++ is supposed to be.

To give an idea of the magnitude of this maintenance task (which carries on indefinitely): Since the 1998 standard until 2006, the core and library working groups has each handled on the order of 600 "defect reports". Fortunately, not all were real defects, but even determining that there really is no problem, or that the problem is just lack of clarity in the standard's text, takes time and care.

Maintenance wasn't all that the committee did from 1997 to 2003. There was a modest amount of planning for the future (thinking about C++0x), but the main activities were writing a technical report on performance issues [67] and one on libraries [68].

6.1 The Performance TR

The performance technical report ("TR") [67] was prompted by a suggestion to standardize a subset of C++ for embedded systems programming. The proposal, called Embedded C++ [40] or simply EC++, originated from a consortium of Japanese embedded systems tool developers (including Toshiba, Hitachi, Fujitsu, and NEC) and had two main concerns: removal of language features that potentially hurt performance and removal of language features perceived to be too complicated for programmers (and thus seen as potential productivity or correctness hazards). A less clearly stated

aim was to define something that in the short term was easier to implement than full standard C++.

The features banned in this (almost) subset included: multiple inheritance, templates, exceptions, run-time type information (§5.1.2), new-style casts, and name spaces. From the standard library, the STL and locales were banned and an alternative version of `iostreams` provided. I considered the proposal misguided and backwards looking. In particular, the performance costs were largely imaginary, or worse. For example, the use of templates has repeatedly been shown to be key to both performance (time and space) and correctness of embedded systems. However, there wasn't much hard data in this area in 1996 when EC++ was first proposed. Ironically, it appears that most of the few people who use EC++ today use it in the form of "Extended EC++" [36], which is EC++ plus templates. Similarly, namespaces (§5.1.1) and new style casts (§5) are features that are primarily there to clarify code and can be used to ease maintenance and verification of correctness. The best documented (and most frequently quoted) overhead of "full C++" as compared to EC++ was `iostreams`. The primary reason for that is that the C++98 `iostreams` support locales whereas that older `iostreams` do not. This is somewhat ironic because the locales were added to support languages different from English (most notably Japanese) and can be optimized away in environments where they are not used (see [67]).

After serious consideration and discussion, the ISO committee decided to stick to the long-standing tradition of not endorsing dialects — even dialects that are (almost) subsets. Every dialect leads to a split in the user community, and so does even a formally defined subset when its users start to develop a separate culture of techniques, libraries, and tools. Inevitably, myths about failings of the full language relative to the "subset" will start to emerge. Thus, I recommend against the use of EC++ in favor of using what is appropriate from (full) ISO Standard C++.

Obviously, the people who proposed EC++ were right in wanting an efficient, well-implemented, and relatively easy-to-use language. It was up to the committee to demonstrate that ISO Standard C++ was that language. In particular, it seemed a proper task for the committee to document the utility of the features rejected by EC++ in the context of performance-critical, resource-constrained, or safety-critical tasks. It was therefore decided to write a technical report on "performance". Its executive summary reads:

> 'The aim of this report is:
>
> - to give the reader a model of time and space overheads implied by use of various C++ language and library features,
> - to debunk widespread myths about performance problems,
> - to present techniques for use of C++ in applications where performance matters, and
> - to present techniques for implementing C++ Standard language and library facilities to yield efficient code.
>
> As far as run-time and space performance is concerned, if you can afford to use C for an application, you can afford to use C++ in a style that uses C++'s facilities appropriately for that application.

Not every feature of C++ is efficient and predictable to the extent that we need for some high-performance and embedded applications. A feature is predictable if we can in advance easily and precisely determine the time needed for each use. In the context of an embedded system, we must consider if we can use

- free store (`new` and `delete`)
- run-time type identification (`dynamic_cast` and `typeid`)
- exceptions (throw and catch)

The time needed to perform one of these operations can depend on the context of the code (e.g. how much stack unwinding a `throw` must perform to reach its matching `catch`) or on the state of the program (e.g. the sequence of `new`s and `delete`s before a `new`).

Implementations aimed at embedded or high performance applications all have compiler options for disabling run-time type identification and exceptions. Free store usage is easily avoided. All other C++ language features are predictable and can be implemented optimally (according to the zero-overhead principle; §2). Even exceptions can be (and tend to be) efficient compared to alternatives [93] and should be considered for all but the most stringent hard-real-time systems. The TR discusses these issues and also defines an interface to the lowest accessible levels of hardware (such as registers). The performance TR was written by a working group primarily consisting of people who cared about embedded systems, including members of the EC++ technical committee. I was active in the performance working group and drafted significant portions of the TR, but the chairman and editor was first Martin O'Riordan and later Lois Goldthwaite. The acknowledgments list 28 people, including Jan Kristofferson, Dietmar Kühl, Tom Plum, and Detlef Vollmann. In 2004, that TR was approved by unanimous vote.

In 2004, after the TR had been finalized, Mike Gibbs from Lockheed-Martin Aero found an algorithm that allows `dynamic_cast` to be implemented in constant time, and fast [48]. This offers hope that `dynamic_cast` will eventually be usable for hard-real-time programming.

The performance TR is one of the few places where the immense amount of C++ usage in embedded systems surfaces in publicly accessible writing. This usage ranges from high-end systems such as found in telecommunications systems to really low-level systems where complete and direct access to specific hardware features is essential (§7). To serve the latter, the performance TR contains a "hard-

ware addressing interface" together with guidelines for its usage. This interface is primarily the work of Jan Kristofferson (representing Ramtex International and Denmark) and Detlef Vollmann (representing Vollmann Engineering GmbH and Switzerland). To give a flavor, here is code copying a register buffer specified by a port called `PortA2_T`:

```
unsigned char mybuf[10];
register_buffer<PortA2_T, Platform> p2;
for (int i = 0; i != 10; ++i)
{
   mybuf[i] = p2[i];
}
```

Essentially the same operation can be done as a block read:

```
register_access<PortA3_T, Platform> p3;
UCharBuf myBlock;
myBlock = p3;
```

Note the use of templates and the use of integers as template arguments; it's essential for a library that needs to maintain optimal performance in time and space. This comes as a surprise to many who have been regaled with stories of memory bloat caused by templates. Templates are usually implemented by generating a copy of the code used for each specialization; that is, for each combination of template arguments. Thus, obviously, if your code generated by a template takes up a lot of memory, you can use a lot of memory. However, many modern templates are written based on the observation that an inline function may shrink to something that's as small as the function preamble or smaller still. That way you can simultaneously save both time and space. In addition to good inlining, there are two more bases for good performance from template code: dead code elimination and avoidance of spurious pointers.

The standard requires that no code is generated for a member function of a class template unless that member function is called for a specific set of template arguments. This automatically eliminates what could have become "dead code". For example:

```
template<class T> class X {
public:
      void f() { /* ... */ }
      void g() { /* ... */ }
      // ...
};

int main()
{
      X<int> xi;
      xi.f();
      X<double> xd;
      xd.g();
      return 0;
 }
```

For this program, the compiler must generate code for `X<int>::f()` and `X<double>::g()` but may not generate code for `X<int>::g()` and `X<double>::f()`. This rule was added to the standard in 1993, at my insistence, specifically to reduce code bloat, as observed in early uses of templates. I have seen this used in embedded systems in the form of the rule "all classes must be templates, even if only a single instantiation is used". That way, dead code is automatically eliminated.

The other simple rule to follow to get good memory performance from templates is: "don't use pointers if you don't need them". This rule preserves complete type information and allows the optimizer to perform really well (especially when inline functions are used). This implies that function templates that take simple objects (such as function objects) as arguments should do so by value rather than by reference. Note that pointers to functions and `virtual` functions break this rule, causing problems for optimizers.

It follows that to get massive code bloat, say megabytes, what you need is to

1. use large function templates (so that the code generated is large)
2. use lots of pointers to objects, virtual functions, and pointers to functions (to neuter the optimizer)
3. use "feature rich" hierarchies (to generate a lot of potentially dead code)
4. use a poor compiler and a poor optimizer

I designed templates specifically to make it easy for the programmer to avoid (1) and (2). Based on experience, the standard deals with (3), except when you violate (1) or (2). In the early 1990s, (4) became a problem. Alex Stepanov named it "the abstraction penalty" problem. He defined "the abstraction penalty" as the ratio of runtime between a templated operation (say, `find` on a `vector<int>` and the trivial non-templated equivalent (say a loop over an array of `int`). An implementation that does all of the easy and obvious optimizations gets a ratio of 1. Poor compilers had an abstraction penalty of 3, though even then good implementations did significantly better. In October 1995, to encourage implementers to do better, Alex wrote the "abstraction penalty benchmark", which simply measured the abstraction penalty [102]. Compiler and optimizer writers didn't like their implementations to be obviously poor, so today ratios of 1.02 or so are common.

The other — and equally important — aspect of C++'s support for embedded systems programming is simply that its model of computation and memory is that of real-world hardware: the built-in types map directly to memory and registers, the built-in operations map directly to machine operations, and the composition mechanisms for data structures do not impose spurious indirections or memory overhead [131]. In this, C++ equals C. See also §2.

The views of C++ as a close-to-machine language with abstraction facilities that can be used to express predicable type-safe low-level facilities has been turned into a

coding standard for safety-critical hard-real-time code by Lockheed-Martin Aero [157]. I helped draft that standard. Generally, C and assembly language programmers understand the direct mapping of language facilities to hardware, but often not the the need for abstraction mechanisms and strong type checking. Conversely, programmers brought up with higher-level "object-oriented" languages often fail to see the need for closeness to hardware and expect some unspecified technology to deliver their desired abstractions without unacceptable overheads.

6.2 The Library TR

When we finished the standard in 1997, we were fully aware that the set of standard libraries was simply the set that we had considered the most urgently needed and also ready to ship. Several much-wanted libraries, such as hash tables, regular expression matching, directory manipulation, and threads, were missing. Work on such libraries started immediately in the Libraries Working Group chaired by Matt Austern (originally working at SGI with Alex Stepanov, then at AT&T Labs with me, and currently at Google). In 2001, the committee started work on a technical report on libraries. In 2004 that TR [68] specifying libraries that people considered most urgently needed was approved by unanimous vote.

Despite the immense importance of the standard library and its extensions, I will only briefly list the new libraries here:

- Polymorphic function object wrapper
- Tuple types
- Mathematical special functions
- Type traits
- Regular expressions
- Enhanced member pointer adaptor
- General-purpose smart pointers
- Extensible random number facility
- Reference wrapper
- Uniform method for computing function-object return types
- Enhanced binder
- Hash tables

Prototypes or industrial-strength implementations of each of these existed at the time of the vote; they are expected to ship with every new C++ implementation from 2006 onwards. Many of these new library facilities are obviously "technical"; that is, they exist primarily to support library builders. In particular, they exist to support builders of standard library facilities in the tradition of the STL. Here, I will just emphasize three libraries that are of direct interest to large numbers of application builders:

- Regular expressions
- General-purpose smart pointers
- Hash tables

Regular expression matching is one of the backbones of scripting languages and of much text processing. Finally, C++ has a standard library for that. The central class is `regex`, providing regular expression matching of patterns compatible with ECMAscript (formerly JavaScript or Jscript) and with other popular notations.

The main "smart pointer" is a reference-counted pointer, `shared_ptr`, intended for code where shared ownership is needed. When the last `shared_ptr` to an object is destroyed, the object pointed to is deleted. Smart pointers are popular, but not universally so and concerns about their performance and likely overuse kept `smart_ptr`'s "ancestor", `counted_ptr`, out of C++98. Smart pointers are not the panacea they are sometimes presented to be. In particular, they can be far more expensive to use than ordinary pointers, destructors for objects "owned" by a set of `shared_ptr`s will run at unpredictable times, and if a lot of objects are deleted at once because the last `shared_ptr` to them is deleted you can incur "garbage-collection delays" exactly as if you were running a general collector. The costs primarily relate to free-store allocation of use-count objects and especially to locking during access to the use counts in threaded systems ("lock-free" implementations appear to help here). If it is garbage collection you want, you might be better off simply using one of the available garbage collectors [11, 47] or waiting for C++0x (§5.4).

No such worries affected hash tables; they would have been in C++98 had we had the time to do a proper detailed design and specification job. There was no doubt that a `hash_map` was needed as an alternative to `map` for large tables where the key was a character string and we could design a good hash function. In 1995, Javier Barreirro, Robert Fraley and David Musser tried to get a proposal ready in time for the standard and their work became the basis for many of the later `hash_map`s [8]. The committee didn't have the time, though, and consequently the Library TR's `unordered_map` (and `unordered_set`) are the result of about eight years of experiment and industrial use. A new name, "unordered_map", was chosen because now there are half a dozen incompatible `hash_map`s in use. The `unordered_map` is the result of a consensus among the `hash_map` implementers and their key users in the committee. An `unordered_map` is "unordered" in the sense that an iteration over its elements are not guaranteed to be in any particular order: a hash function doesn't define an ordering in the way a `map`'s < does.

The most common reaction to these extensions among developers is "that was about time; why did it take you so long?" and "I want much more right now". That's understandable (I too want much more right now — I just know that I can't get it), but such statements reflect a lack of understanding what an ISO committee is and can do. The com-

mittee is run by volunteers and requires both a consensus and an unusual degree of precision in our specifications (see D&E §6.2). The committee doesn't have the millions of dollars that commercial vendors can and do spend on "free", "standard" libraries for their customers.

7. C++ in Real-World Use

Discussions about programming languages typically focus on language features: which does the language have? how efficient are they? More enlightened discussions focus on more difficult questions: how is the language used? how can it be used? who can use it? For example, in an early OOPSLA keynote, Kristen Nygaard (of Simula and OOP fame) observed: "if we build languages that require a PhD from MIT to use, we have failed". In industrial contexts, the first — and often only — questions are: Who uses the language? What for? Who supports it? What are the alternatives? This section presents C++ and its history from the perspective of its use.

Where is C++ used? Given the number of users (§1), it is obviously used in a huge number of places, but since most of the use is commercial it is difficult to document. This is one of the many areas where the lack of a central organization for C++ hurts the C++ community — nobody systematically gathers information about its use and nobody has anywhere near complete information. To give an idea of the range of application areas, here are a few examples where C++ is used for crucial components of major systems and applications:

- Adobe — Acrobat, Photoshop, Illustrator, ...
- Amadeus — airline reservations
- Amazon — e-commerce
- Apple — iPod interface, applications, device drivers, finder, ...
- AT&T — 1-800 service, provisioning, recovery after network failure, ...
- Games — Doom3, StarCraft, Halo, ...
- Google — search engines, Google Earth, ...
- IBM — K42 (very high-end operating system), AS/400, ...
- Intel — chip design and manufacturing, ...
- KLA-Tencor — semiconductor manufacturing
- Lockheed-Martin Aero — airplane control (F16, JSF), ...
- Maeslant Barrier — Dutch surge barrier control
- MAN B&W — marine diesel engine monitoring and fuel injection control
- Maya — professional 3D animation
- Microsoft — Windows XP, Office, Internet explorer, Visual Studio, .Net, C# compiler, SQL, Money, ...
- Mozilla Firefox — browser
- NASA/JPL — Mars Rover scene analysis and autonomous driving, ...
- Southwest Airlines — customer web site, flight reservations, flight status, frequent flyer program, ...
- Sun — compilers, OpenOffice, HotSpot Java Virtual Machine, ...
- Symbian — OS for hand-held devices (especially cellular phones)
- Vodaphone — mobile phone infrastructure (including billing and provisioning)

For more examples, see [137]. Some of the most widely used and most profitable software products ever are on this list. Whatever C++'s theoretical importance and impact, it certainly met its most critical design aim: it became an immensely useful practical tool. It brought object-oriented programming and more recently also generic programming into the mainstream. In terms of numbers of applications and the range of application areas, C++ is used beyond any individual's range of expertise. That vindicates my emphasis on generality in D&E [121] (Chapter 4).

It is a most unfortunate fact that applications are not documented in a way that reaches the consciousness of researchers, teachers, students, and other application builders. There is a huge amount of experience "out there" that isn't documented or made accessible. This inevitably warps people's sense of reality — typically in the direction of what is new (or perceived as new) and described in trade press articles, academic papers, and textbooks. Much of the visible information is very open to fads and commercial manipulation. This leads to much "reinvention of the wheel", suboptimal practice, and myths.

One common myth is that "most C++ code is just C code compiled with a C++ compiler". There is nothing wrong with such code — after all, the C++ compiler will find more bugs than a C compiler — and such code is not uncommon. However, from seeing a lot of commercial C++ code and talking with innumerable developers and from talking with developers, I know that for many major applications, such as the ones mentioned here, the use of C++ is far more "adventurous". It is common for developers to mention use of major locally designed class hierarchies, STL use, and use of "ideas from STL" in local code. See also §7.2.

Can we classify the application areas in which C++ is used? Here is one way of looking at it:

- Applications with systems components
- Banking and financial (funds transfer, financial modeling, customer interaction, teller machines, ...)
- Classical systems programming (compilers, operating systems, editors, database systems, ...)

- Conventional small business applications (inventory systems, customer service, ...)
- Embedded systems (instruments, cameras, cell phones, disc controllers, airplanes, rice cookers, medical systems, ...)
- Games
- GUI — iPod, CDE desktop, KDE desktop, Windows, ...
- Graphics
- Hardware design and verification [87]
- Low-level system components (device drivers, network layers, ...)
- Scientific and numeric computation (physics, engineering, simulations, data analysis, ...)
- Servers (web servers, large application backbones, billing systems, ...)
- Symbolic manipulation (geometric modeling, vision, speech recognition, ...)
- Telecommunication systems (phones, networking, monitoring, billing, operations systems, ...)

Again, this is more a list than a classification. The world of programming resists useful classification. However, looking at these lists, one thing should be obvious: C++ cannot be ideal for all of that. In fact, from the earliest days of C++, I have maintained that a general-purpose language can at most be second best in a well-defined application area and that C++ is "a general-purpose programming language with a bias towards systems programming".

7.1 Applications Programming vs. Systems Programming

Consider an apparent paradox: C++ has a bias towards systems programming but "most programmers" write applications. Obviously millions of programmers are writing applications and many of those write their applications in C++. Why? Why don't they write them in an applications programming language? For some definition of "applications programming language", many do. We might define an "applications programming language" as one in which we can't directly access hardware and that provides direct and specialized support for an important applications concept (such as data base access, user interaction, or numerical computation). Then, we can deem most languages "applications languages": Excel, SQL, RPG, COBOL, Fortran, Java, C#, Perl, Python, etc. For good and bad, C++ is used as a general-purpose programming language in many areas where a more specialized (safer, easier to use, easier to optimize, etc.) language would seem applicable.

The reason is not just inertia or ignorance. I don't claim that C++ is anywhere near perfect (that would be absurd) nor that it can't be improved (we are working on C++0x, after all and see §9.4). However, C++ does have a niche — a very large niche — where other current languages fall short:

- applications with a significant systems programming component; often with resource constraints
- applications with components that fall into different application areas so that no single specialized applications language could support all

Application languages gain their advantages through specialization, through added conveniences, and through eliminating difficult to use or potentially dangerous features. Often, there is a run-time or space cost. Often, simplifications are based on strong assumptions about the execution environment. If you happen to need something fundamental that was deemed unnecessary in the design (such as direct access to memory or fully general abstraction mechanisms) or don't need the "added conveniences" (and can't afford the overhead they impose), C++ becomes a candidate. The basic conjecture on which C++ is built is that many applications have components for which that is the case: "Most of us do unusual things some of the time".

The positive way of stating this is that general mechanisms beat special-purpose features for the majority of applications that does not completely fit into a particular classification, have to collaborate with other applications, or significantly evolve from their original narrow niche. This is one reason that every language seems to grow general-purpose features, whatever its original aims were and whatever its stated philosophy is.

If it was easy and cheap to switch back and forth among applications languages and general-purpose languages, we'd have more of a choice. However, that is rarely the case, especially where performance or machine-level access is needed. In particular, using C++ you can (but don't have to) break some fundamental assumption on which an application language is built. The practical result is that if you need a systems programming or performance-critical facility of C++ somewhere in an application, it becomes convenient to use C++ for a large part of the application — and then C++'s higher-level (abstraction) facilities come to the rescue. C++ provides hardly any high-level features that are directly applicable in an application. What it offers are mechanisms for defining such facilities as libraries.

Please note that from a historical point of view this analysis need not be correct or the only possible explanation of the facts. Many prefer alternative ways of looking at the problem. Successful languages and companies have been built on alternative views. However, it is a fact that C++ was designed based on this view and that this view guided the evolution of C++; for example, see Chapter 9 of [121]. I consider it the reason that C++ initially succeeded in the mainstream and the reason that its use continued to grow steadily during the time period covered by this paper, despite the continuing

presence of well-designed and better-financed alternatives in the marketplace. See also §9.4.

7.2 Programming Styles

C++ supports several programming styles or, as they are sometimes somewhat pretentiously called, "programming paradigms". Despite that, C++ is often referred to as "an object-oriented programming language". This is only true for some really warped definition of "object-oriented" and I never say just "C++ is an object-oriented language" [122]. Instead, I prefer "C++ supports object-oriented programming and other programming styles" or "C++ is a multi-paradigm programming language". Programming style matters. Consequently, the way people refer to a language matters because it sets expectations and influences what people see as ideals.

C++ has C (C89) as an "almost subset" and supports the styles of programming commonly used for C [127]. Arguably, C++ supports those styles better than C does by providing more type checking and more notational support. Consequently, a lot of C++ code has been written in the style of C or — more commonly — in the style of C with a number of classes and class hierarchies thrown in without affecting the overall design. Such code is basically procedural, using classes to provide a richer set of types. That's sometimes referred to as "C with Classes style". That style can be significantly better (for understanding, debugging, maintenance, etc.) than pure C. However, it is less interesting from a historical perspective than code that also uses the C++ facilities to express more advanced programming techniques, and very often less effective than such alternatives. The fraction of C++ code written purely in "C style" appears to have been decreasing over the last 15 years (or more).

The abstract data type and object-oriented styles of C++ usage have been discussed often enough not to require explanation here (e.g., see [126]). They are the backbone of many C++ applications. However, there are limits to their utility. For example, object-oriented programming can lead to overly rigid hierarchies and overreliance on virtual functions. Also, a virtual function call is fundamentally efficient for cases where you need to select an action at run time, but it is still an indirect function call and thus expensive compared to an individual machine instruction. This has led to generic programming becoming the norm for C++ where high performance is essential (§6.1).

7.2.1 Generic programming

Sometimes, generic programming in C++ is defined as simply "using templates". That's at best an oversimplification. A better description from a programming language feature point of view is "parametric polymorphism" [107] plus overloading, which is selecting actions and constructing types based on parameters. A template is basically a compile-time mechanism for generating definitions (of classes and functions) based on type arguments, integer arguments, etc. [144].

Before templates, generic programming in C++ was done using macros [108], `void*`, and casts, or abstract classes. Naturally, some of that still persists in current use and occasionally these techniques have advantages (especially when combined with templatized interfaces). However, the current dominant form of generic programming relies on class templates for defining types and function templates for defining operations (algorithms).

Being based on parameterization, generic programming is inherently more regular than object-oriented programming. One major conclusion from the years of use of major generic libraries, such as the STL, is that the current support for generic programming in C++ is insufficient. C++0x is taking care of at least part of that problem (§8).

Following Stepanov (§4.1.8), we can define generic programming without mentioning language features: Lift algorithms and data structures from concrete examples to their most general and abstract form. This typically implies representing the algorithms and their access to data as templates, as shown in the descripton of the STL (§4.1).

7.2.2 Template metaprogramming

The C++ template instantiation mechanism is (when compiler limits are ignored, as they usually can be) Turing complete (e.g., see [150]). In the design of the template mechanism, I had aimed at full generality and flexibility. That generality was dramatically illustrated by Erwin Unruh in the early days of the standardization of templates. At an extensions working group meeting in 1994, he presented a program that calculated prime numbers at compile time (using error messages as the output mechanism) [143] and appeared surprised that I (and others) thought that marvelous rather than scary. Template instantiation is actually a small compile-time functional programming language. As early as 1995, Todd Veldhuizen showed how to define a compile-time if-statement using templates and how to use such if-statements (and switch-statements) to select among alternative data structures and algorithms [148]. Here is a compile-time if-statement with a simple use:

```
template<bool b, class X, class Y>
struct if_ {
   typedef X type;  // use X if b is true
};

template<class X, class Y>
struct if_<false,X,Y> {
     typedef Y type;  // use Y if b is false
};

void f()
{
   if_<sizeof(Foobar)<40, Foo, Bar>::type xy;
```

```
    // ...
}
```

If the size of type `Foobar` is less than `40`, the type of the variable `xy` is `Foo`; otherwise it is `Bar`. The second definition of `if_` is a partial specialization used when the template arguments match the `<false,X,Y>` pattern specified. It's really quite simple, but very ingenious and I remember being amazed when Jeremy Siek first showed it to me. In a variety of guises, it has proven useful for producing portable high-performance libraries (e.g., most of the Boost libraries [16] rely on it).

Todd Veldhuizen also contributed the technique of expression templates [147], initially as part of the implementation of his high-performance numeric library Blitz++ [149]. The key idea is to achieve compile-time resolution and delayed evaluation by having an operator return a function object containing the arguments and operation to be (eventually) evaluated. The < operator generating a `Less_than` object in §4.1.4 is a trivial example. David Vandevoorde independently discovered this technique.

These techniques and others that exploit the computational power of template instantiation provide the foundation for techniques based on the idea of generating source code that exactly matches the needs of a given situation. It can lead to horrors of obscurity and long compile times, but also to elegant and very efficient solutions to hard problems; see [3, 2, 31]. Basically, template instantiation relies on overloading and specialization to provide a reasonably complete functional compile-time programming language.

There is no universally agreed-upon definition of the distinction between generic programming and template metaprogramming. However, generic programming tends to emphasize that each template argument type must have an enumerated well-specified set of properties; that is, it must be able to define a concept for each argument (§4.1.8, §8.3.3). Template metaprogramming doesn't always do that. For example, as in the `if_` example, template definitions can be chosen based on very limited aspects of an argument type, such as its size. Thus, the two styles of programming are not completely distinct. Template metaprogramming blends into generic programming as more and more requirements are placed on arguments. Often, the two styles are used in combination. For example, template metaprogramming can be used to select template definitions used in a generic part of a program.

When the focus of template use is very strongly on composition and selection among alternatives, the style of programming is sometimes called "generative programming" [31].

7.2.3 Multi-paradigm programming

It is important for programmers that the various programming styles supported by C++ are part of a single language. Often, the best code requires the use of more than one of the four basic "paradigms". For example, we can write the classical "draw all shapes in a container" example from SIMULA BEGIN [9] like this:

```
void draw_all(vector<Shape*>& v)
{
   for_each(v.begin(), v.end(),     // sequence
            mem_fun(&Shape::draw)); // operation
}
```

Here, we use object-oriented programming to get the run-time polymorphism from the `Shape` class hierarchy. We use generic programming for the parameterized (standard library) container `vector` and the parameterized (standard library) algorithm `for_each`. We use ordinary procedural programming for the two functions `draw_all` and `mem_fun`. Finally, the result of the call of `mem_fun` is a function object, a class that is not part of a hierarchy and has no virtual functions, so that can be classified as abstract data type programming. Note that `vector`, `for_each`, `begin`, `end`, and `mem_fun` are templates, each of which will generate the most appropriate definition for its actual use.

We can generalize that to any sequence defined by a pair of `ForwardIterator`s, rather than just `vector`s and improve type checking using C++0x concepts (§8.3.3):

```
template<ForwardIterator For>
void draw_all(For first, For last)
   requires SameType<For::value_type,Shape*>
{
   for_each(first, last, mem_fun(&Shape::draw));
}
```

I consider it a challenge to properly characterize multi-paradigm programming so that it can be easy enough to use for most mainstream programmers. This will involve finding a more descriptive name for it. Maybe it could even benefit from added language support, but that would be a task for C++1x.

7.3 Libraries, Toolkits, and Frameworks

So, what do we do when we hit an area in which the C++ language is obviously inadequate? The standard answer is: Build a library that supports the application concepts. C++ isn't an application language; it is a language with facilities supporting the design and implementation of elegant and efficient libraries. Much of the talk about object-oriented programming and generic programming comes down to building and using libraries. In addition to the standard library (§4) and the components from the library TR (§6.2), examples of widely used C++ libraries include

- ACE [95] — distributed computing
- Anti-Grain Geometry — 2D graphics
- Borland Builder (GUI builder)
- Blitz++[149] — vectors "The library that thinks it is a compiler"

- Boost[16] — foundation libraries building on the STL, graph algorithms, regular expression matching, threading, ...
- CGAL[23] — computational geometry
- Maya — 3D animation
- MacApp and PowerPlant — Apple foundation frameworks
- MFC — Microsoft Windows foundation framework
- Money++ — banking
- RogueWave library (pre-STL foundation library)
- STAPL[4], POOMA[86] — parallel computation
- Qt [10], FLTK [156], gtkmm [161], wxWidgets [162] — GUI libraries and builders
- TAO [95], MICO, omniORB — CORBA ORBs
- VTK [155] — visualization

In this context, we use the word "toolkit" to describe a library supported by programming tools. In this sense, VTK is a toolkit because it contains tools for generating interfaces in Java and Python and Qt and FLTK are toolkits because they provide GUI builders. The combination of libraries and tools is an important alternative to dialects and special-purpose languages [133, 151].

A library (toolkit, framework) can support a huge user community. Such user communities can be larger than the user communities of many programming languages and a library can completely dominate the world-view of their users. For example, Qt [10] is a commercial product with about 7,500 paying customers in 2006 plus about 150,000 users of its open-source version [141]. Two Norwegian programmers, Eirik Chambe-Eng and Haavard Nord, started what became Qt in 1991-92 and the first commercial release was in 1995. It is the basis of the popular desktop KDE (for Linux, Solaris, and FreeBSD) and well known commercial products, such as Adobe Photoshop Elements, Google Earth, and Skype (Voice over IP service).

Unfortunately for C++'s reputation, a good library cannot be seen; it just does its job invisibly to its users. This often leads people to underestimate the use of C++. However, "there are of course the Windows Foundation Classes (MFC), MacApp, and PowerPlant — most Mac and Windows commercial software is built with one of these frameworks" [85].

In addition to these general and domain-specific libraries, there are many much more specialized libraries. These have their function limited to a specific organization or application. That is, they apply libraries as an application design philosophy: "first extend the language by a library, then write the application in the resulting extended language". The "string library" (part of a larger system called "Panther") used by Celera Genomics as the base for their work to sequence the human genome [70] is a spectacular example of this approach. Panther is just one of many C++ libraries and applications in the general area of biology and biological engineering.

7.4 ABIs and Environments

Using libraries on a large scale isn't without problems. C++ supports source-level compatibility (but provides only weak link-time compatibility guarantees). That's fine if

- you have (all) the source
- your code compiles with your compiler
- the various parts of your source code are compatible (e.g., with respect to resource usage and error handling)
- your code is all in C++

For a large system, typically none of these conditions hold. In other words, linking is a can of worms. The root of this problem is the fundamental C++ design decision: Use existing linkers (D&E §4.5).

It is not guaranteed that two C translation units that match according to the language definition will link correctly when compiled with different compilers. However, for every platform, agreement has been reached for an ABI (Application Binary Interface) so that the register usage, calling conventions, and object layout match for all compilers so that C programs will correctly link. C++ compilers use these conventions for function call and simple structure layout. However, traditionally C++ compiler vendors have resisted linkage standards for layout of class hierarchies, virtual function calls, and standard library components. The result is that to be sure that a legal C++ program actually works, every part (including all libraries) has to be compiled by the same compiler. On some platforms, notably Sun's and also Itanium (IA64) [60], C++ ABI standards exist but historically the rule "use a single compiler or communicate exclusively through C functions and `struct`s" is the only really viable rule.

Sticking with one compiler can ensure link compatibility on a platform, but it can also be a valuable tool in providing portability across many platforms. By sticking to a single implementer, you gain "bug compatibility" and can target all platforms supported by that vendor. For Microsoft platforms, Microsoft C++ provides that opportunity; for a huge range of platforms, GNU C++ is portable; and for a diverse set of platforms, users get a pleasant surprise when they notice that many of their local implementations use an EDG (Edison Design Group) front-end making their source code portable. This only (sic!) leaves the problems of version skew. During this time period every C++ compiler went through a series of upgrades, partly to increase standard conformance, to adjust to platform ABIs, and to improve performance, debugging, integration with IDEs, etc.

Link compatibility with C caused a lot of problems, but also yielded significant advantages. Sean Parent (Adobe) observes: "one reason I see for C++'s success is that it

is ‘close enough’ to C that platform vendors are able to provide a single C interface (such as the Win32 API or the Mac Carbon API) which is C++ compatible. Many libraries provide a C interface which is C++ compatible because the effort to do so is low — where providing an interface to a language such as Eiffel or Java would be a significant effort. This goes beyond just keeping the linking model the same as C but to the actual language compatibility”.

Obviously, people have tried many solutions to the linker problem. The platform ABIs are one solution. CORBA is a platform- and language-independent (or almost so) solution that has found widespread use. However, it seems that C++ and Java are the only languages heavily used with CORBA. COM was Microsoft’s platform-dependent and language-independent solution (or almost so). One of the origins of Java was a perceived need to gain platform independence and compiler independence; the JVM solved that problem by eliminating language independence and completely specifying linkage. The Microsoft CLI (Common Language Infrastructure) solves the problem in a language-independent manner (sort of) by requiring all languages to support a Java-like linkage, metadata, and execution model. Basically all of these solutions provide platform independence by becoming a platform: To use a new machine or operating system, you port a JVM, an ORB, etc.

The C++ standard doesn’t directly address the problem of platform incompatibilities. Using C++, platform independence is provided through platform-specific code (typically relying on conditional compilation — `#ifdef`). This is often messy and ad hoc, but a high degree of platform independence can be provided by localizing dependencies in the implementation of a relatively simple platform layer — maybe just a single header file [16]. In any case, to implement the platform-independent services effectively, you need a language that can take advantage of the peculiarities of the various hardware and operating systems environments. More often than not, that language is C++.

Java, C#, and many other languages rely on metadata (that is, data that defines types and services associated with them) and provide services that depend on such metadata (such as marshalling of objects for transfer to other computers). Again, C++ takes a minimalist view. The only “metadata” available is the RTTI (§5.1.2), which provides just the name of the class and the list of its base classes. When RTTI was discussed, some of us dreamed of tools that would provide more data for systems that needed it, but such tools did not become common, general-purpose, or standard.

7.5 Tools and Research

Since the late 1980s, C++ developers have been supported by a host of analysis tools, development tools, and development environments available in the C++ world. Examples are:

- Visual Studio (IDE; Microsoft)
- KDE (Desktop Environment; Free Software)
- Xcode (IDE; Apple)
- lint++ (static analysis tool; Gimpel Software)
- Vtune (multi-level performance tuning; Intel)
- Shark (performance optimization; Apple)
- PreFAST (static source code analysis; Microsoft)
- Klocwork (static code analysis; Klocwork)
- LDRA (testing; LDRA Ltd.)
- QA.C++ (static analysis; Programming Research)
- Purify (memory leak finder; IBM Rational)
- Great Circle (garbage collector and memory usage analyzer; Geodesic, later Symantec)

However, tools and environments have always been a relative weakness of C++. The root of that problem is the difficulty of parsing C++. The grammar is not LR(N) for any N. That’s obviously absurd. The problem arose because C++ was based directly on C (I borrowed a YACC-based C parser from Steve Johnson), which was “known” not to be expressible as a LR(1) grammar until Tom Pennello discovered how to write one in 1985. Unfortunately, by then I had defined C++ in such a way that Pennello’s techniques could not be applied to C++ and there was already too much code dependent on the non-LR grammar to change C++. Another aspect of the parsing problem is the macros in the C preprocessor. They ensure that what the programmer sees when looking at a line of code can — and often does — dramatically differ from what the compiler sees when parsing and type checking that same line of code. Finally, the builder of advanced tools must also face the complexities of name lookup, template instantiation, and overload resolution.

In combination, these complexities have confounded many attempts to build tools and programming environments for C++. The result was that far too few software development and source code analysis tools have become widely used. Also, the tools that were built tended to be expensive. This led to relatively fewer academic experiments than are conducted with languages that were easier to analyze. Unfortunately, many take the attitude that “if it isn’t standard, it doesn’t exist” or alternatively “if it costs money, it doesn’t exist”. This has led to lack of knowledge of and underuse of existing tools, leading to much frustration and waste of time.

Indirectly, this parsing problem has caused weaknesses in areas that rely on run-time information, such as GUI-builders. Since the language doesn’t require it, compilers generally don’t produce any form of metadata (beyond the minimum required by RTTI; §5.1.2). My view (as stated in the original RTTI papers and in D&E) was that tools could be used to produce the type information needed by a specific application or application area. Unfortunately, the parsing problem then gets in the way. The tool-to-generate-metadata approach has been successfully used for database access

systems and GUI, but cost and complexity have kept this approach from becoming more widely used. In particular, academic research again suffered because a typical student (or professor) doesn't have the time for serious infrastructure building.

A focus on performance also plays a part in lowering the number and range of tools. Like C, C++ is designed to ensure minimal run-time and space overheads. For example, the standard library `vector` is not by default range checked because you can build an optimal range-checked vector on top of an unchecked vector, but you cannot build an optimally fast vector on top of a range-checked one (at least not portably). However, many tools rely on additional actions (such as range-checking array access or validating pointers) or additional data (such as meta-data describing the layout of a data structure). A subculture of strong concern about performance came into the C++ community with much else from C. Often that has been a good thing, but it did have a limiting effect on tool building by emphasizing minimalism even where there were no serious performance issues. In particular, there is no reason why a C++ compiler couldn't supply superb type information to tool builders [134].

Finally, C++ was a victim of its own success. Researchers had to compete with corporations that (sometimes correctly) thought that there was money to be made in the kind of tools researchers would like to build. There was also a curious problem with performance: C++ was too efficient for any really significant gains to come easily from research. This led many researchers to migrate to languages with glaring inefficiencies for them to eliminate. Elimination of virtual function calls is an example: You can gain much better improvements for just about any object-oriented language than for C++. The reason is that C++ virtual function calls are very fast and that colloquial C++ already uses non-virtual functions for time-critical operations. Another example is garbage collection. Here the problem was that colloquial C++ programs don't generate much garbage and that the basic operations are fast. That makes the fixed overhead of a garbage collector looks far less impressive when expressed as a percentage of run time than it does for a language with less efficient basic operations and more garbage. Again, the net effect was to leave C++ poorer in terms of research and tools.

7.6 C/C++ Compatibility

C is C++'s closest relative and a high degree of C compatibility has always been a design aim for C++. In the early years, the primary reasons for compatibility were to share infrastructure and to guarantee completeness (§2.2). Later, compatibility remained important because of the huge overlap in applications areas and programmer communities. Many little changes were made to C++ during the '80s and '90s to bring C++ closer to ISO C [62] (C89). However, during 1995-2004, C also evolved. Unfortunately, C99 [64] is in significant ways less compatible with C++ than C89 [62] and harder to coexist with. See [127] for a detailed discussion of the C/C++ relationship. Here is a diagram of the relationships among the various generations of C and C++:

"Classic C" is what most people think of as K&R C, but C as defined in [76] lacks structure copy and enumerations. ARM C++ is C++ as defined by the ARM [35] and the basis for most pre-standard C++. The basic observation is that by now C (i.e., C89 or C99) and C++ (i.e., C++98) are siblings (with "Classic C" as their common ancestor), rather than the more conventional view that C++ is a dialect of C that somehow failed to be compatible. This is an important issue because people commonly proclaim the right of each language to evolve separately, yet just about everybody expects ISO C++ to adopt the features adopted by ISO C — despite the separate evolution of C and a tendency of the C committee to adopt features that are similar to but incompatible with what C++ already offers. Examples selected from a long list [127] are `bool`, `inline` functions, and `complex` numbers.

7.7 Java and Sun

I prefer not to compare C++ to other programming languages. For example, in the "Notes to the Reader" section of D&E [121], I write:

> Several reviewers asked me to compare C++ to other languages. This I have decided against doing. ... Language comparisons are rarely meaningful and even less often fair. A good comparison of major programming languages requires more effort than most people

> are willing to spend, experience in a wide range of application areas, a rigid maintenance of a detached and impartial point of view, and a sense of fairness. I do not have the time, and as the designer of C++, my impartiality would never be fully credible. ... Worse, when one language is significantly better known than others, a subtle shift in perspective occurs: Flaws in the well-known language are deemed minor and simple workarounds are presented, whereas similar flaws in other languages are deemed fundamental. Often, the workarounds commonly used in the less-well-known languages are simply unknown to the people doing the comparison or deemed unsatisfactory because they would be unworkable in the more familiar language. ... Thus, I restrict my comments about languages other than C++ to generalities and to very specific comments.

However, many claims about the C++/Java relationship have been made and the presence of Java in the marketplace has affected the C++ community. Consequently, a few comments are unavoidable even though a proper language comparison is far beyond the scope of this paper and even though Java has left no traces in the C++ definition.

Why not? From the earliest days of Java, the C++ committee has always included people with significant Java experience: users, implementers, tool builders, and JVM implementers. I think at a fundamental level Java and C++ are too different for easy transfer of ideas. In particular,

- C++ relies on direct access to hardware resources to achieve many of its goals whereas Java relies on a virtual machine to keep it away from the hardware.
- C++ is deliberately frugal with run-time support whereas Java relies on a significant amount of metadata
- C++ emphasizes interoperability with code written in other languages and sharing of system tools (such as linkers) whereas Java aims for simplicity by isolating Java code from other code.

The "genes" of C++ and Java are quite dissimilar. The syntactic similarities between Java and C++ have often been deceptive. As an analogy, I note that it is far easier for English to adopt "structural elements" from closely related languages, such as French or German, than from more different languages, such as Japanese or Thai.

Java burst onto the programming scene with an unprecedented amount of corporate backing and marketing (much aimed at non-programmers). According to key Sun people (such as Bill Joy), Java was an improved and simplified C++. "What Bjarne would have designed if he hadn't had to be compatible with C" was — and amazingly still is — a frequently heard statement. Java is not that; for example, in D&E §9.2.2, I outlined fundamental design criteria for C++:

> What would be a better language than C++ for the things C++ is meant for? Consider the first-order decisions:
>
> - Use of static type checking and Simula-like classes.
> - Clean separation between language and environment.
> - C source compatibility ("as close as possible").
> - C link and layout compatibility ("genuine local variables").
> - No reliance on garbage collection.
>
> I still consider static type checking essential for good design and run-time efficiency. Were I to design a new language for the kind of work done in C++ today, I would again follow the Simula model of type checking and inheritance, *not* the Smalltalk or Lisp models. As I have said many times, 'Had I wanted an imitation Smalltalk, I would have built a much better imitation. Smalltalk is the best Smalltalk around. If you want Smalltalk, use it'.

I think I could express that more clearly today, but the essence would be the same; these criteria are what define C++ as a systems programming language and what I would be unwilling to give up. In the light of Java, that section seems more relevant today than when I wrote it in 1993 (pre-Java). Having the built-in data types and operators mapped directly to hardware facilities and being able to exploit essentially every machine resource is implicit in "C compatibility".

C++ does not meet Java's design criteria; it wasn't meant to. Similarly, Java doesn't meet C++'s design criteria. For example, consider a couple of language-technical criteria:

- Provide as good support for user-defined types as for built-in types
- Leave no room for a lower-level language below C++ (except assembler)

Many of the differences can be ascribed to the aim of keeping C++ a systems programming language with the ability to deal with hardware and systems at the lowest level and with the least overhead. Java's stated aims seem more directed towards becoming an applications language (for some definition of "application").

Unfortunately, the Java proponents and their marketing machines did not limit themselves to praising the virtues of Java, but stooped to bogus comparisons (e.g., [51]) and to name calling of languages seen as competitors (most notably C and C++). As late as 2001, I heard Bill Joy claim (orally with a slide to back it up in a supposedly technical presentation) that "reliable code cannot be written in C/C++ because they don't have exceptions" (see §5, §5.3). I see Java as a Sun commercial weapon aimed at Microsoft that missed

and hit an innocent bystander: the C++ community. It hurt many smaller language communities even more; consider Smalltalk, Lisp, Eiffel, etc.

Despite many promises, Java didn't replace C++ ("Java will completely kill C++ within two years" was a graphic expression I repeatedly heard in 1996). In fact, the C++ community has trebled in size since the first appearance of Java. The Java hype did, however, harm the C++ community by diverting energy and funding away from much-needed tools, library and techniques work. Another problem was that Java encouraged a limited "pure object-oriented" view of programming with a heavy emphasis on run-time resolution and a de-emphasis of the static type system (only in 2005 did Java introduce "generics"). This led many C++ programmers to write unnecessarily inelegant, unsafe, and poorly performing code in imitation. This problem gets especially serious when the limited view infects education and creates a fear of the unfamiliar in students.

As I predicted [136] when I first heard the boasts about Java's simplicity and performance, Java rapidly accreted new features — in some cases paralleling C++'s earlier growth. New languages are always claimed to be "simple" and to become useful in a wider range of real-world applications they increase in size and complexity. Neither Java nor C++ was (or is) immune to that effect. Obviously, Java has made great strides in performance — given its initial slowness it couldn't fail to — but the Java object model inhibits performance where abstraction is seriously used (§4.1.1, §4.1.4). Basically, C++ and Java are far more different in aims, language structure, and implementation model than most people seem to think. One way of viewing Java is as a somewhat restricted version of Smalltalk's run-time model hidden behind a C++-like syntax and statically type-checked interfaces.

My guess is that Java's real strength compared to C++ is the binary compatibility gained from Sun having de facto control of the linker as expressed through the definition of the Java virtual machine. This gives Java the link-compatibility that has eluded the C++ community because of the decision to use the basic system linkers and the lack of agreement among C++ vendors on key platforms (§7.4).

In the early 1990s, Sun developed a nice C++ compiler based on Mike Ball and Steve Clamage's Taumetric compiler. With the release of Java and Sun's loudly proclaimed pro-Java policy where C++ code was referred to (in advertising "literature" and elsewhere) as "legacy code" that needed rewriting and as "contamination", the C++ group suffered some lean years. However, Sun never wavered in its support of the C++ standards efforts and Mike, Steve and others made significant contributions. Technical people have to live with technical realities — such as the fact that many Sun customers and many Sun projects rely on C++ (in total or in part). In particular, Sun's Java implementation, HotSpot, is a C++ program.

7.8 Microsoft and .Net

Microsoft is currently the 800-pound gorilla of software development and it has had a somewhat varied relationship with C++. Their first attempt at an object-oriented C in the late 1980s wasn't C++ and I have the impression that a certain ambivalence about standard conformance lingers. Microsoft is better known for setting de facto standards than for strictly sticking to formal ones. However, they did produce a C++ compiler fairly early. Its main designer was Martin O'Riordan, who came from the Irish company Glockenspiel where he had been a Cfront expert and produced and maintained many ports. He once amused himself and his friends by producing a Cfront that spoke its error messages through a voice synthesizer in (what he believed to be) a thick Danish accent. To this day there are ex-Glockenspiel Irishmen on the Microsoft C++ team.

Unfortunately, that first release didn't support templates or exceptions. Eventually, those key features were supported and supported well, but that took years. The first Microsoft compiler to provide a close-to-complete set of ISO C++ features was VC++ 6.0, released in July 1998; its predecessor, VC++ 5.0 from February 1997, already had many of the key features. Before that, some Microsoft managers used highly visible platforms, such as conference keynotes, for some defensive bashing of these features (as provided only by their competitors, notably Borland) as "expensive and useless". Worse, internal Microsoft projects (which set widely followed standards) couldn't use templates and exceptions because their compiler didn't support those features. This established bad programming practices and did long-term harm.

Apart from that, Microsoft was a responsible and attentive member of the community. Microsoft sent and sends members to the committee meetings and by now — somewhat belatedly — provides an excellent C++ compiler with good standard conformance.

To effectively use the .Net framework, which Microsoft presents as the future of Windows, a language has to support a Java-like set of facilities. This implies support for a large number of language features including a massive metadata mechanism and inheritance — complete with covariant arrays (§4.1.1). In particular, a language used to produce components for consumption by other languages must produce complete metadata and a language that wants to consume components produced by other languages must be able to accept the metadata they produce. The Microsoft C++ dialect that supports all that, ISO C++ plus "The CLI extensions to ISO C++", colloquially referred to as C++/CLI [41], will obviously play a major role in the future of the C++ world. Interestingly, C++ with the C++/CLI extensions is the only language that provides access to every feature of .Net. Basically, C++/CLI is a massive set of extensions to ISO C++ and provides a degree of integration with Windows that makes it unlikely that a program that relies on C++/CLI features in

any significant way will be portable beyond the Microsoft platforms that provide the massive infrastructure on which C++/CLI depends. As ever, systems interfaces can be encapsulated, and must be encapsulated to preserve portability. In addition to ISO C++, C++/CLI provides its own loop construct, overloading mechanisms (indexers), "properties", event mechanism, garbage-collected heap, a different class object initialization semantics, a new form of references, a new form of pointers, generics, a new set of standard containers (the .Net ones), and more.

In the early years of .Net (around 2000 onwards), Microsoft provided a dialect called "managed C++" [24]. It was generally considered "pretty lame" (if essential to some Microsoft users) and appears to have been mostly a stopgap measure — without any clear indication what the future might bring for its users. It has now been superseded by the much more comprehensive and carefully designed C++/CLI.

One of my major goals in working in the standards committee was to prevent C++ from fracturing into dialects. Clearly, in the case of C++/CLI, I and the committee failed. C++/CLI has been standardized by ECMA [41] as a binding to C++. However, Microsoft's main products, and those of their major customers, are and will remain in C++. This ensures good compiler and tool support for C++ — that is ISO C++ — under Windows for the foreseeable future. People who care about portability can program around the Microsoft extensions to ensure platform independence (§7.4). By default the Microsoft compiler warns about use of C++/CLI extensions.

The members of the C++ standards committee were happy to see C++/CLI as an ECMA standard. However, an attempt to promote that standard to an ISO standard caused a flood of dissent. Some national standards bodies — notably the UK's C++ Panel — publicly expressed serious concern [142]. This caused people at Microsoft to better document their design rationale [140] and to be more careful about not confusing ISO C++ and C++/CLI in Microsoft documentation.

7.9 Dialects

Obviously, not everyone who thought C++ to be basically a good idea wanted to go though the long and often frustrating ISO standards process to get their ideas into the mainstream. Similarly, some people consider compatibility overrated or even a very bad idea. In either case, people felt they could make faster progress and/or better by simply defining and implementing their own dialect. Some hoped that their dialect would eventually become part of the mainstream; others thought that life outside the mainstream was good enough for their purposes, and a few genuinely aimed at producing a short-lived language for experimental purposes.

There have been too many C++ dialects (many dozens) for me to mention more than a tiny fraction. This is not ill will — though I am no great fan of dialects because they fracture the user community [127, 133] — but a reflection that they have had very little impact outside limited communities. So far, no major language feature has been brought into the mainstream of C++ from a dialect. However, C++0x will get something that looks like C++/CLI's properly scoped `enum`s [138], a C++/CLI-like `for`-statement [84], and the keyword `nullptr` [139] (which curiously enough was suggested by me for C++/CLI).

Concurrency seems to be an extremely popular area of language extension. Here are a few C++ dialects supporting some form of concurrency with language features:

- Concurrent C++[46]
- micro C++[18]
- ABC++ [83]
- Charm++ [75]
- POOMA [86]
- C++// [21]

Basically, every trend and fad in the computer science world spawned a couple of C++ dialects, such as Aspect C++ [100], R++ [82], Compositional C++ [25], Objective C++ [90], Open C++ [26], and dozens more. The main problem with these dialects is their number. Dialects tend to split up a sub-user community to the point where none reach a large enough user community to afford a sustainable infrastructure [133].

Another kind of dialect appears when a vendor adds some (typically minor) "neat features" to their compiler to help their users with some specific tasks (e.g. OS access and optimization). These become barriers to portability even if they are often beloved by some users. They are also appreciated by some managers and platform fanatics as a lock-in mechanism. Examples exist for essentially every implementation supplier; for example Borland (Delphi-style properties),GNU (variable and type attributes), and Microsoft (C++/CLI; §7.8). Such dialect features are nasty when they appear in header files, setting off a competitive scramble among vendors keen on being able to cope with their competitors' code. They also significantly complicate compiler, library, and tool building.

It is not obvious when a dialect becomes a completely separate language and many languages borrowed heavily from C++ (with or without acknowledgments) without aiming for any degree of compatibility. A recent example is the "D" language (the currently most recent language with that popular name): "D was conceived in December 1999 by Walter Bright as a reengineering of C and C++" [17].

Even Java started out (very briefly) as a C++ dialect [164], but its designers soon decided that maintaining compatibility with C++ would be too constraining for their needs. Since their aims included 100% portability of Java code and a restriction of code styles to a version of object-oriented programming, they were almost certainly correct in

that. Achieving any degree of useful compatibility is very hard, as the C/C++ experience shows.

8. C++0x

From late 1997 until 2002, the standards committee deliberately avoided serious discussion of language extension. This allowed compiler, tool, and library implementers to catch up and users to absorb the programming techniques supported by Standard C++. I first used the term "C++0x" in 2000 and started a discussion of "directions for C++0x" through presentations in the committee and elsewhere from 2001 onwards. By 2003, the committee again considered language extensions. The "extensions working group" was reconstituted as the "evolution working group". The name change (suggested by Tom Plum) is meant to reflect a greater emphasis on the integration of language features and standard library facilities. As ever, I am the chairman of that working group, hoping to help ensure a continuity of vision for C++ and a coherence of the final result. Similarly, the committee membership shows continuous participation of a large number of people and organizations. Fortunately, there are also many new faces bringing new interests and new expertise to the committee.

Obviously, C++0x is still work in progress, but years of work are behind us and and many votes have been taken. These votes are important in that they represent the response of an experienced part of the C++ community to the problems with C++98 and the current challenges facing C++ programmers. The committee intended to be cautious and conservative about changes to the language itself, and strongly emphasize compatibility. The stated aim was to channel the major effort into an expansion of the standard library. In the standard library, the aim was to be aggressive and opportunistic. It is proving difficult to deliver on that aim, though. As usual, the committee just doesn't have sufficient resources. Also, people seem to get more excited over language extensions and are willing to spend more time lobbying for and against those.

The rate of progress is roughly proportional to the number of meetings. Since the completion of the 1998 standard, there has been two meetings a year, supplemented by a large amount of web traffic. As the deadlines for C++0x approach, these large meetings are being supplemented by several "extra" meetings focused on pressing topics, such as concepts (§8.3.3) and concurrency (§5.5).

8.1 Technical Challenges

What technical challenges faced the C++ community at the time when C++0x was beginning to be conceived? At a high level an answer could be:

- GUI-based application building
- Distributed computing (especially the web)
- Security

The big question is how to translate that into language features, libraries (ISO standard or not), programming environment, and tools. In 2000-2002, I tried unsuccessfully to get the standards committee's attention on the topic of distributed computing and Microsoft representatives regularly try to raise the topic of security. However, the committee simply doesn't think like that. To make progress, issues have to be expressed as concrete proposals for change, such as to add a (specific) language feature to support callbacks or to replace the notoriously unsafe C standard library function `gets()` with a (specific) alternative. Also, the C++ tradition is to approach challenges obliquely though improved abstraction mechanisms and libraries rather than providing a solution for a specific problem.

After I had given a couple of talks on principles and directions, the first concrete action in the committee was a brainstorming session at the 2001 meeting in Redmond, Washington. Here we made a "wish list" for features for C++0x. The first suggestion was for a portable way of expressing alignment constraints (by P. J. Plauger supported by several other members). Despite my strongly expressed urge to focus on libraries, about 90 out of about 100 suggestions were for language features. The saving grace was that many of those were for features that would ease the design, implementation, and use of more elegant, more portable, and efficient libraries. This brainstorming provided the seeds of maintained "wish lists" for C++0x language features and standard library facilities [136].

From many discussions and presentations, I have come with a brief summary of general aims and design rules for C++0x that appear to be widely accepted. Aims:

- Make C++ a better language for systems programming and library building — rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development)
- Make C++ easier to teach and learn — through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

Rules of thumb:

- Provide stability and compatibility
- Prefer libraries to language extensions
- Make only changes that change the way people think
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

These lists have been useful as a framework for rationales for proposed extensions. Dozens of committee technical papers have used them. However, they provide only a weak set of directions and are only a weak counterweight to a widespread tendency to focus on details. The GUI and distributed computing challenges are not directly represented here, but feature prominently under "libraries" and "work directly with hardware" (§8.6).

8.2 Suggested Language Extensions

To give a flavor of the committee work around 2005, consider a (short!) incomplete list of proposed extensions:

1. `decltype` and `auto` — type deduction from expressions (§8.3.2)
2. Template aliases — a way of binding some but not all template parameters and naming the resulting partial template specialization
3. Extern templates — a way of suppressing implicit instantiation in a translation unit
4. Move semantics (rvalue references) — a general mechanism for eliminating redundant copying of values
5. Static assertions (`static_assert`)
6. `long long` and many other C99 features
7. >> (without a space) to terminate two template specializations
8. Unicode data types
9. Variadic templates (§8.3.1)
10. Concepts — a type system for C++ types and integer values (§8.3.3)
11. Generalized constant expressions [39]
12. Initializer lists as expressions (§8.3.1)
13. Scoped and strongly typed enumerations [138]
14. Control of alignment
15. `nullptr` — Null pointer constant [139]
16. A `for`-statement for ranges
17. Delegating constructors
18. Inherited constuctors
19. Atomic operations
20. Thread-local storage
21. Defaulting and inhibiting common operations
22. Lambda functions
23. Programmer-controlled garbage collection [12] (§5.4)
24. In-class member initializers
25. Allow local classes as template parameters
26. Modules
27. Dynamic library support
28. Integer sub-ranges
29. Multi-methods
30. Class namespaces
31. Continuations
32. Contract programming — direct support for preconditions, postconditions, and more
33. User-defined operator. (dot)
34. `switch` on `string`
35. Simple compile-time reflection
36. `#nomacro` — a kind of scope to protect code from unintended macro expansion
37. GUI support (e.g., slots and signals)
38. Reflection (i.e., data structures describing types for run-time use)
39. concurrency primitives in the language (not in a library)

As ever, there are far more proposals than the committee could handle or the language could absorb. As ever, even accepting all the good proposals is infeasible. As ever, there seems to be as many people claiming that the committee is spoiling the language by gratuitous complicated features as there are people who complain that the committee is killing the language by refusing to accept essential features. If you take away consistent overstatement of arguments, both sides have a fair degree of reason behind them. The balancing act facing the committee is distinctly nontrivial.

As of October 2006, Items 1-7 have been approved. Based on the state of proposals and preliminary working group votes, my guess is that items 10-21 will also be accepted. Beyond that, it's hard to guess. Proosals 22-25 are being developed aiming for votes in July 2007 and proposal 26 (modules) has been postponed to a technical report.

Most of these are being worked upon under one or more of the "rules of thumb" listed above. That list is less than half of the suggestions that the committee has received in a form that compels it to (at least briefly) consider them. My collection of suggestions from emails and meetings with users is several times that size. At my urging, the committee at the spring 2005 meeting in Lillehammer, Norway decided (by vote) to stop accepting new proposals. In October of 2006, this vote was followed up by a vote to submit a draft standard in late 2007 so as to make C++0x into C++09. However, even with the stream of new proposals stemmed, it is obvious that making a coherent whole of a selection of features will be a practical challenge as well as a technical one.

To give an idea of the magnitude of the technical challenge, consider that a paper on part of the concepts problem was accepted for the premier academic conference in the field, POPL, in 2006 [38] and other papers analyzing prob-

lems related to concepts were presented at OOPSLA [44, 52] and PLDI [74]. I personally consider the technical problems related to the support of concurrency (including the memory model) harder still — and essential. The C++0x facilities for dealing with concurrency are briefly discussed in §8.6.

The practical challenge is to provide correct and consistent detailed specifications of all these features (and standard library facilities). This involves producing and checking hundreds of pages of highly technical standards text. More often than not, implementation and experimentation are part of the effort to ensure that a feature is properly specified and interacts well with other features in the language and the standard library. The standard specifies not just an implementation (like a vendor's documentation), but a whole set of possible implementations (different vendors will provide different implementations of a feature, each with the same semantics, but with different engineering tradeoffs). This implies an element of risk in accepting any new feature — even in accepting a feature that has already been implemented and used. Committee members differ in their perception of risk, in their views of what risks are worth taking, and in their views of how risk is best handled. Naturally, this is a source of many difficult discussions.

All proposals and all the meeting minutes of the committee are available on the committee's website [69]. That's more than 2000 documents — some long. A few of the pre-1995 papers are not yet (August 2006) available online because the committee relied on paper until about 1994.

8.3 Support for Generic Programming

For the language itself, we see an emphasis on features to support generic programming because generic programming is the area where use of C++ has progressed the furthest relative to the support offered by the language.

The overall aim of the language extensions supporting generic programming is to provide greater uniformity of facilities so as to make it possible to express a larger class of problems directly in a generic form. The potentially most significant extensions of this kind are:

- general initializer lists (§8.3.1)
- `auto` (§8.3.2)
- concepts (§8.3.3)

Of these proposals, only `auto` has been formally voted in. The others are mature proposals and initial "straw votes" have been taken in their support.

8.3.1 General initializer lists

"Provide as good support for user-defined types as for built-in types" is prominent among the language-technical design principles of C++ (§2). C++98 left a major violation of that principle untouched: C++98 provides notational support for initializing a (built-in) array with a list of values, but there is no such support for a (user-defined) `vector`. For example, we can easily define an array initialized to the three `int`s `1`, `2`, and `3`:

```
int a[] = { 1,2,3 };
```

Defining an equivalent `vector` is awkward and may require the introduction of an array:

```
// one way:
     vector v1;
     v1.push_back(1);
     v1.push_back(2);
     v1.push_back(3);

// another way
     int a[] = { 1,2,3 };
     vector v2(a,a+sizeof(a)/sizeof(int));
```

In C++0x, this problem will be remedied by allowing a user to define a "sequence constructor" for a type to define what initialization with an initializer list means. By adding a sequence constructor to `vector`, we would allow:

```
vector<int> v = { 1,2,3 };
```

Because the semantics of initialization defines the semantics of argument passing, this will also allow:

```
int f(const vector<int>&);
// ...
int x = f({ 1,2,3 });
int y = f({ 3,4,5,6,7,8, 9, 10 });
```

That is, C++0x gets a type-safe mechanism for variable-length homogeneous argument lists [135].

In addition, C++0x will support type-safe variadic template arguments [54] [55]. For example:

```
template<class ... T> void print(const T& ...);
// ...
string name = "World"'
print("Hello, ",name,'!');
int x = 7;
print("x = ",x);
```

At the cost of generating a unique instantiation (from a single template function) for each call with a given set of argument types, variadic templates allow arbitrary non-homogenous argument lists to be handled. This is especially useful for tuple types and similar abstractions that inherently deal with heterogeneous lists.

I was the main proponent of the homogenous initializer list mechanism. Doug Gregor and Jaakko Järvi designed the variadic template mechanism, which was developed to simplify the implementation of libraries, such as Jakko Järvi and Gary Powell's lambda library [72] and the tuple library [71].

8.3.2 Auto

C++0x will support the notion of a variable being given a type deduced from the type of its initializer [73]. For

example, we could write the verbose example from §4.1.3 like this:

```
auto q = find(vi.begin(),vi.end(),7); // ok
```

Here, we deduce the type of `q` to be the return type of the value returned by `find`, which in turn is the type of `vi.begin()`; that is, `vector<int>::iterator`. I first implemented that use of `auto` in 1982, but was forced to back it out of "C with Classes" because of a C compatibility problem. In K&R C [76] (and later in C89 and ARM C++), we can omit the type in a declaration. For example:

```
static x;    // means static int x
auto y;      // means stack allocated int y
```

After a proposal by Dag Bruč̌k, both C99 and C++98 banned this "implicit `int`". Since there is now no legal code for "`auto q`" to be incompatible with, we allowed it. That incompatibility was always more theoretical than real (reviews of huge amounts of code confirm that), but using the (unused) keyword `auto` saved us from introducting a new keyword. The obvious choice (from many languages, including BCPL) is `let`, but every short meaningful word has already been used in many programs and long and cryptic keywords are widely disliked.

If the committee — as planned — accepts overloading based on concepts (§8.3.3) and adjusts the standard library to take advantage, we can even write:

```
auto q = find(vi,7); // ok
```

in addition to the more general, but wordier:

```
auto q = find(vi.begin(),vi.end(),7); // ok
```

Unsurprisingly, given its exceptionally long history, I was the main designer of the `auto` mechanism. As with every C++0x proposal, many people contributed, notably Gabriel Dos Reis, Jaakko Järvi, and Walter Brown.

8.3.3 Concepts

The D&E [121] discussion of templates contains three whole pages (§15.4) on constraints on template arguments. Clearly, I felt the need for a better solution — and so did many others. The error messages that come from slight errors in the use of a template, such as a standard library algorithm, can be spectacularly long and unhelpful. The problem is that the template code's expectations of its template arguments are implicit. Consider again `find_if`:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
     while (first!=last && !pred(*first))
        ++first;
     return first;
}
```

Here, we make a lot of assumptions about the `In` and `Pred` types. From the code, we see that `In` must somehow support `!=`, `*`, and `++` with suitable semantics and that we must be able to copy `In` objects as arguments and return values. Similarly, we see that we can call a `Pred` with an argument of whichever type `*` returns from an `In` and apply `!` to the result to get something that can be treated as a `bool`. However, that's all implicit in the code. Note that a lot of the flexibility of this style of generic programming comes from implicit conversions used to make template argument types meet those requirements. The standard library carefully documents these requirements for input iterators (our `In`) and predicates (our `Pred`), but compilers don't read manuals. Try this error and see what your compiler says:

```
find_if(1,5,3.14);        // errors
```

On 2000-vintage C++ compilers, the resulting error messages were uniformly spectacularly bad.

Constraints classes A partial, but often quite effective, solution based on my old idea of letting a constructor check assumptions about template arguments (D&E §15.4.2) is now finding widespread use. For example:

```
template<class T> struct Forward_iterator {
     static void constraints(T a) {
        ++a; a++;        // can increment
        T b = a; b = a;  // can copy
        *b = *a;         // can dereference
                        // and copy result
   }
     Forward_iterator()
      { void (*p)(T) = constraints; }
};
```

This defines a class that will compile if and only if `T` is a forward iterator [128]. However, a `Forward_iterator` object doesn't really do anything, so that a compiler can (and all do) trivially optimize away such objects. We can use `Forward_iterator` in a definition like this:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
     Forward_iterator<In>();  // check
     while (first!=last && !pred(*first))
        ++first;
     return first;
}
```

Alex Stepanov and Jeremy Siek did a lot to develop and popularize such techniques. One place where they are used prominently is in the Boost library [16]. The difference in the quality of error messages can be spectacular. However, it is not easy to write constraints classes that consistently give good error messages on all compilers.

Concepts as a language feature Constraints classes are at best a partial solution. In particular, the type checking is done in the template definition. For proper separation of concerns, checking should rely only on information presented in a declaration. That way, we would obey the usual rules for

interfaces and could start considering the possibility of genuine separate compilation of templates.

So, let's tell the compiler what we expect from a template argument:

```
template<ForwardIterator In, Predicate Pred>
In find_if(In first, In last, Pred pred);
```

Assuming that we can express what a `ForwardIterator` and a `Predicate` are, the compiler can now check a call of `find_if` in isolation from its definition. What we are doing here is to build a type system for template arguments. In the context of modern C++, such "types of types" are called *concepts* (see §4.1.8). There are various ways of specifying such concepts; for now, think of them as a kind of constraints classes with direct language support and a nicer syntax. A concept says what facilities a type must provide, but nothing about how it provides those facilities. The use of a concept as the type of a template argument (e.g. `<ForwardIterator In>`) is very close to a mathematical specification "for all types `In` such that an `In` can be incremented, dereferenced, and copied", just as the original `<class T>` is the mathematical "for all types `T`".

Given only that declaration (and not the definition) of `find_if`, we can write

```
int x = find_if(1,2,Less_than<int>(7));
```

This call will fail because `int` doesn't support unary `*` (dereference). In other words, the call will fail to compile because an `int` isn't a `ForwardIterator`. Importantly, that makes it easy for a compiler to report the error in the language of the user and at the point in the compilation where the call is first seen.

Unfortunately, knowing that the iterator arguments are `ForwardIterator`s and that the predicate argument is a `Predicate` isn't enough to guarantee successful compilation of a call of `find_if`. The two argument types interact. In particular, the predicate takes an argument that is an iterator dereferenced by `*`: `pred(*first)`. Our aim is complete checking of a template in isolation from the calls and complete checking of each call without looking at the template definition. So, "concepts" must be made sufficiently expressive to deal with such interactions among template arguments. One way is to parameterize the concepts in parallel to the way the templates themselves are parameterized. For example:

```
template<Value_type T,
   ForwardIterator<T> In,  // sequence of Ts
   Predicate<bool,T> Pred> // takes a T;
                           // returns a bool
   In find_if(In first, In last, Pred pred);
```

Here, we require that the `ForwardIterator` must point to elements of a type `T`, which is the same type as the `Predicate`'s argument type. However, that leads to overly rigid interactions among template argument types and very complex patterns and redundant of parameterization [74].

The current concept proposal [129, 130, 132, 97, 53] focuses on expressing relationships among arguments directly:

```
template<ForwardIterator In,  // a sequence
         Predicate Pred>      // returns a bool
   requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);
```

Here, we require that the `In` must be a `ForwardIterator` with a `value_type` that is acceptable as an argument to `Pred` which must be a `Predicate`.

A *concept* is a compile-time predicate on a set of types and integer values. The concepts of a single type argument provide a type system for C++ types (both built-in and user-defined types) [132].

Specifying template's requirements on its argument using concepts also allows the compiler to catch errors in the template definition itself. Consider this plausible pre-concept definition:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
     while(first!=last && !pred(*first))
        first = first+1;
     return first;
}
```

Test this with a `vector` or an array, and it will work. However, we specified that `find_if` should work for a `ForwardIterator`. That is, `find_if` should be usable for the kind of iterator that we can supply for a list or an input stream. Such iterators cannot simply move `N` elements forward (by saying `p=p+N`) — not even for `N==1`. We should have said `++first`, which is not just simpler, but correct. Experience shows that this kind of error is very hard to catch, but concepts make it trivial for the compiler to detect it:

```
template<ForwardIterator In, Predicate Pred>
      requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred)
{
     while(first!=last && !pred(*first))
             first = first+1;
     return first;
}
```

The + operator simply isn't among the operators specified for a `ForwardIterator`. The compiler has no problems detecting that and reporting it succinctly.

One important effect of giving the compiler information about template arguments is that overloading based on the properties of argument types becomes easy. Consider again `find_if`. Programmers often complain about having to specify the beginning and the end of a sequence when all they want to do is to find something in a container. On the other hand, for generality, we need to be able to express algorithms in terms of sequences delimited by iterators. The obvious solution is to provide both versions:

```
template<ForwardIterator In, Predicate Pred>
   requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);

template<Container C, Predicate Pred>
   requires Callable<Pred,C::value_type>
In find_if(C& c, Pred pred);
```

Given that, we can handle examples like the one in §8.3.2 as well as examples that rely on more subtle differences in the argument types.

This is not the place to present the details of the concept design. However, as presented above, the design appears to have a fatal rigidity: The expression of required properties of a type is often in terms of required member types. However, built-in types do not have members. For example, how can an `int*` be a `ForwardIterator` when a `ForwardIterator` as used above is supposed to have a member `value_type`? In general, how can we write algorithms with precise and detailed requirements on argument types and expect that authors of types will define their types with the required properties? We can't. Nor can we expect programmers to alter old types whenever they find a new use for them. Real-world types are often defined in ignorance of their full range of uses. Therefore, they often fail to meet the detailed requirements even when they possess all the fundamental properties needed by a user. In particular, `int*` was defined 30 years ago without any thought of C++ or the STL notion of an iterator. The solution to such problems is to (non-intrusively) map the properties of a type into the requirements of a concept. In particular, we need to say "when we use a pointer as a forward iterator we should consider the type of the object pointed to its `value_type`". In the C++0x syntax, that is expressed like this:

```
template<class T>
concept_map ForwardIterator<T*> {
     typedef T value_type;
}
```

A `concept_map` provides a map from a type (or a set of types; here, `T*`) to a `concept` (here, `ForwardIterator`) so that users of the concept for an argument will see not the actual type, but the mapped type. Now, if we use `int*` as a `ForwardIterator` that `ForwardIterator`'s `value_type` will be `int`. In addition to providing the appearance of members, a `concept_map` can be used to provide new names for functions, and even to provide new operations on objects of a type.

Alex Stepanov was probably the first to call for "concepts" as a C++ language feature [104] (in 2002) — I don't count the numerous vague and nonspecific wishes for "better type checking of templates". Initially, I was not keen on the language approach because I feared it would lead in the direction of rigid interfaces (inhibiting composition of separately developed code and seriously hurting performance), as in earlier ideas for language-supported "constrained genericity". In the summer of 2003, Gabriel Dos Reis and I analyzed the problem, outlined the design ideals, and documented the basic approaches to a solution [129] [130]. So, the current design avoids those ill effects (e.g., see [38]) and there are now many people involved in the design of concepts for C++0x, notably Doug Gregor, Jaakko Järvi, Gabriel Dos Reis, Jeremy Siek, Andrew Lumsdaine and me. An experimental implementation of concepts has been written by Doug Gregor, together with a version of the STL using concepts [52].

I expect concepts to become central to all generic programming in C++. They are already central to much design using templates. However, existing code — not using concepts — will of course continue to work.

8.4 Embarrassments

My other priority (together with better support for generic programming) is better support for beginners. There is a remarkable tendency for proposals to favor the expert users who propose and evaluate them. Something simple that helps only novices for a few months until they become experts is often ignored. I think that's a potentially fatal design bias. Unless novices are sufficiently supported, only few will become experts. Bootstrapping is most important! Further, many — quite reasonably — don't want to become experts; they are and want to remain "occasional C++ users". For example, a physicist using C++ for physics calculations or the control of experimental equipment is usually quite happy being a physicist and has only limited time for learning programming techniques. As computer scientists we might wish for people to spend more time on programming techniques, but rather than just hoping, we should work on removing unnecessary barriers to adoption of good techniques. Naturally, most of the changes needed to remove "embarrassments" are trivial. However, their solution is typically constrained by compatibility concerns and concerns for the uniformity of language rules.

A very simple example is

```
vector<vector<double>> v;
```

In C++98, this is a syntax error because >> is a single lexical token, rather than two >s each closing a template argument list. A correct declaration of `v` would be:

```
vector< vector<double> > v;
```

I consider that an embarrassment. There are perfectly good language-technical reasons for the current rule and the extensions working group twice rejected my suggestions that this was a problem worth solving. However, such reasons are language technical and of no interest to novices (of all backgrounds — including experts in other languages). Not accepting the first and most obvious declaration of `v` wastes time for users and teachers. A simple solution of the ">> problem" was proposed by David Vandevoorde [145] and voted in at the 2005 Lillehammer meeting, and I expect

many small "embarrassments" to be absent from C++0x. However, my attempt together with Francis Glassborow and others, to try to systematically eliminate the most frequently occurring such "embarrassments" seems to be going nowhere.

Another example of an "embarrassment" is that it is legal to copy an object of a class with a user-defined destructor using a default copy operation (constructor or assignment). Requiring user-defined copy operations in that case would eliminate a lot of nasty errors related to resource management. For example, consider an oversimplified string class:

```
class String {
public:
     String(char* pp);          // constructor
     ~String() { delete[] pp; } // destructor
     char& operator[](int i);
private:
     int sz;
     char* p;
};

void f(char* x)
{
   String s1(x);
   String s2 = s1;
}
```

After the construction of `s2`, `s1.p` and `s2.p` point to the same memory. This memory (allocated by the constructor) will be deleted twice by the destructor, probably with disastrous results. This problem is obvious to the experienced C++ programmer, who will provide proper copy operations or prohibit copying. The problem has also been well documented from the earliest days of C++: The two obvious solutions can be found in TC++PL1 and D&E. However, the problem can seriously baffle a novice and undermine trust in the language. Language solutions have been proposed by Lois Goldtwaite, Francis Glassborow, Lawrence Crowl, and I [30]; some version may make it into C++0x.

I chose this example to illustrate the constraints imposed by compatibility. The problem could be eliminated by not providing default copy of objects of a class with pointer members; if people wanted to copy, they could supply a copy operator. However, that would break an unbelievable amount of code. In general, remedying long-standing problems is harder than it looks, especially if C compatibility enters into the picture. However, in this case, the existence of a destructor is a strong indicator that default copying would be wrong, so examples such as `String` could be reliably caught by the compiler.

8.5 Standard libraries

For the C++0x standard library, the stated aim was to make a much broader platform for systems programming. The June 2006 version "Standard Libraries Wish List" maintained by Matt Austern lists 68 suggested libraries including

- Container-based algorithms
- Random access to files
- Safe STL (completely range checked)
- File system access
- Linear algebra (vectors, matrices, etc.)
- Date and time
- Graphics library
- Data compression
- Unicode file names
- Infinite-precision integer arithmetic
- Uniform use of std::string in the library
- Threads
- Sockets
- Comprehensive support for unicode
- XML parser and generator library
- Graphical user interface
- Graph algorithms
- Web services (SOAP and XML bindings)
- Database support
- Lexical analysis and parsing

There has been work on many of these libraries, but most are postponed to post-C++0x library TRs. Sadly, this leaves many widely desired and widely used library components unstandardized. In addition, from observing people struggling with C++98, I also had high hopes that the committee would take pity on the many new C++ programmers and provide library facilities to support novices from a variety of backgrounds (not just beginning programmers and refugees from C). I have low expectations for the most frequently requested addition to the standard library: a standard GUI (Graphical User Interface; see §1 and §5.5).

The first new components of the C++0x standard library are those from the library TR (§6.2). All but one of the components were voted in at the spring 2006 meeting in Berlin. The special mathematical functions were considered too specialized for the vast majority of C++ programmers and were spun off to become a separate ISO standard.

In addition to the more visible work on adding new library components, much work in the library working group focuses on minor improvements to existing components, based on experience, and on improved specification of existing components. The accumulative effect of these minor improvements is significant.

The plan for 2007 includes going over the standard library to take advantage of new C++0x features. The first and most obvious example is to add rvalue initializers [57] (primarily the work of Howard Hinnant, Peter Dimov, and Dave Abrahams) to significantly improve the performance

of heavily used components, such as `vector`, that occasionally have to move objects around. Assuming that concepts (§8.3.3) make it into C++0x, their use will revolutionize the specification of the STL part of the library (and other templated components). Similarly, the standard containers, such as `vector` should be augmented with sequence constructors to allow then to accept initializer lists (§8.3.1). Generalized constant expressions (`constexpr`, primarily the work of Gabriel Dos Reis and I [39]) will allow us to define simple functions, such as the ones defining properties of types in the `numeric_limits` and the `bitset` operators, so that they are usable at compile time. The variadic template mechanism (§8.3.1) dramatically simplifies interfaces to standard compunents, such as `tuple`, that can take a wide variety of template arguments. This has significant implications on the usability of these standard library components in performance-critical areas.

Beyond that, only a threads library seems to have gathered enough support to become part of C++0x. Other components that are likely to become part of another library TR (TR2) are:

- File system library — platform-independent file system manipulation [32]
- Date and time library [45]
- Networking — sockets, TCP, UDP, multicast, iostreams over TCP, and more [80]
- `numeric_cast` — checked conversions [22]

The initial work leading up to the proposals and likely votes mentioned here has been the main effort of the library working group 2003-06. The networking library has been used commercially for applications on multiple continents for some time. At the outset, Matt Austern (initially AT&T, later Apple) was the chair; now Howard Hinnant (initially Metrowerks, later Apple) has that difficult job.

In addition, the C committee is adopting a steady stream of technical reports, which must be considered and (despite the official ISO policy that C and C++ are distinct languages) will probably have to be adopted into the C++ library even though they — being C-style — haven't benefited from support from C++ language features (such as user-defined types, overloading, and templates). Examples are decimal floating-point and unicode built-in types with associated operations and functions.

All in all, this is a massive amount of work for the couple of dozen volunteers in the library working group, but it is not quite the "ambitious and opportunistic" policy that I had hoped for in 2001 (§8). However, people who scream for more (such as me) should note that even what's listed above will roughly double the size of the standard library. The process of library TRs is a hope for the future.

8.6 Concurrency

Concurrency cannot be perfectly supported by a library alone. On the other hand, the committee still considers language-based approaches to concurrency, such as is found in Ada and Java, insufficiently flexible (§5.5). In particular, C++0x must support current operating-system thread library approaches, such as POSIX threads and Windows threads.

Consequently, the work on concurrency is done by an ad hoc working group stradding the library-language divide. The approach taken is to carefully specify a machine model for C++ that takes into account modern hardware architectures [14] and to provide minimal language primitives:

- thread local storage [29]
- atomic types and operations [13]

The rest is left to a threads library. The current threads library draft is written by Howard Hinnant [58].

This "concurrency effort" is led by Hans Boehm (Hewlett-Packard) and Lawrence Crowl (formerly Sun, currently Google). Naturally, compiler writers and hardware vendors are most interested and supportive in this. For example, Clark Nelson from Intel redesigned the notion of sequencing that C++ inherited from C to better fit C++ on modern hardware [89]. The threads library will follow the traditions of Posix threads [19], Windows threads [163], and libraries based on those, such as boost::thread [16]. In particular, the C++0x threads library will not try to settle every issue with threading; instead it will provide a portable set of core facilities but leave some tricky issues system dependent. This is necessary to maintain compatibility both with earlier libraries and with the underlying operating systems. In addition to the typical thread library facilities (such as lock and mutex), the library will provide thread pools and a version of "futures" [56] as a higher-level and easier-to-use communications facility: A future can be described as a placeholder for a value to be computed by another thread; synchronization potentially happens when the future is read. Prototype implementations exist and are being evaluated [58].

Unfortunately, I had to accept that my hopes of direct support for distributed programming are beyond what we can do for C++0x.

9. Retrospective

This retrospective looks back with the hope of extracting lessons that might be useful in the future:

- Why did C++ succeed?
- How did the standards process serve the C++ community?
- What were the influences on C++ and what has been its impact?
- Is there a future for the ideals that underlie C++?

9.1 Why Did C++ Succeed?

That's a question thoughtful people ask me a few times every year. Less thoughtful people volunteer imaginative answers to it on the web and elsewhere most weeks. Thus, it is worth while to both briefly answer the question and to contradict the more popular myths. Longer versions of the answer can be found in D&E [121], in my first HOPL paper [120], and in sections of this paper (e.g., §1, §7, and §9.4). This section is basically a summary.

C++ succeeded because

- *Low-level access plus abstraction*: The original conception of C++, "C-like for systems programming plus Simula-like for abstraction," is a most powerful and useful idea (§7.1, §9.4).
- *A useful tool*: To be useful a language has to be complete and fit into the work environments of its users. It is neither necessary nor sufficient to be the best in the world at one or two things (§7.1).
- *Timing*: C++ was not the first language to support object-oriented programming, but from day one it was available as a useful tool for people to try for real-world problems.
- *Non-proprietary*: AT&T did not try to monopolize C++. Early on, implementations were relatively cheap (e.g., $750 for educational institutions) and in 1989 all rights to the language were transferred to the standard bodies (ANSI and later ISO). I and others from Bell Labs actively helped non-AT&T C++ implementers to get started.
- *Stable*: A high degree (but not 100%) of C compatibility was considered essential from day one. A higher degree of compatibility (but still not 100%) with earlier implementations and definitions was maintained throughout. The standards process was important here (§3.1, §6).
- *Evolving*: New ideas have been absorbed into C++ throughout (e.g., exceptions, templates, the STL). The world evolves, and so must a living language. C++ didn't just follow fashion; occasionally, it led (e.g., generic programming and the STL). The standards process was important here (§4.1, §5, §8).

Personally, I particularly like that the C++ design never aimed as solving a clearly enumerated set of problems with a clearly enumerated set of specific solutions: I'd never design a tool that could only do what I wanted [116]. Developers tend not to appreciate this open-endedness and emphasis on abstraction until their needs change.

Despite frequent claims, the reason for C++'s success was *not*:

- *Just luck*: I am amazed at how often I hear that claimed. I fail to understand how people can imagine that I and others involved with C++ could — by pure luck — repeatedly have provided services preferred by millions of systems builders. Obviously an element of luck is needed in any endeavor, but to believe that mere bumbling luck can account for the use of C++ 1.0, C++ 2.0, ARM C++, and C++98 is stretching probabilities quite a bit. Staying lucky for 25 years can't be easy. No, I didn't imagine every use of templates or of the STL, but I aimed for generality (§4.1.2).
- *AT&T's marketing might*: That's funny! All other languages that have ever competed commercially with C++ were better financed. AT&T's C++ marketing budget for 1985-1988 (inclusive) was $5,000 (out of which we only managed to spend $3,000; see D&E). At the first OOPSLA, we couldn't afford to bring a computer, we had no flyers, and not a single marketer (we used a chalkboard, a signup sheet for research papers, and researchers). In later years, things got worse.
- *It was first*: Well, Ada, Eiffel, Objective C, Smalltalk and various Lisp dialects were all available commercially before C++. For many programmers, C, Pascal, and Modula-2 were serious contenders.
- *Just C compatibility*: C compatibility helped — as it should because it was a design aim. However, C++ was never just C and it earned the hostility from many C devotees early on for that. To meet C++'s design aims, some incompatibilities were introduced. C later adopted several C++ features, but the time lag ensured that C++ got more criticism than praise for that (§7.6). C++ was never an "anointed successor to C".
- *It was cheap*: In the early years, a C++ implementation was relatively cheap for academic institutions, but about as expensive as competitive languages for commercial use (e.g., about $40,000 for a commercial source license). Later, vendors (notably Sun and Microsoft) tended to charge more for C++ than for their own proprietary languages and systems.

Obviously, the reasons for success are as complex and varied as the individuals and organizations that adopted C++.

9.2 Effects of the Standards Process

Looking back on the '90s and the first years of the 21st century, what questions can we ask about C++ and the standards committee's stewardship of the language?

- Is C++98 better than ARM C++?
- What was the biggest mistake?
- What did C++ get right?
- Can C++ be improved? and if so, how?
- Can the ISO standards process be improved? and if so, how?

Yes, C++98 is a better language (with a much better library) than ARM C++ for the kinds of problems that C++ was designed to address. Thus, the efforts of the committee must be

deemed successful. In addition, the population of C++ programmers has grown by more than an order of magnitude (from about 150,000 in 1990 to over 3 million in 2004; §1) during the stewardship of the committee. Could the committee have done better? Undoubtedly, but exactly how is hard to say. The exact conditions at the time where decisions are made are hard to recall (to refresh your memory, see D&E [121]).

So what was the biggest mistake? For the early years of C++, the answer was easy ("not shipping a larger/better foundation library with the early AT&T releases" [120]). However, for the 1991-1998 period, nothing really stands out — at least if we have to be realistic. If we don't have to be realistic, we can dream of great support for distributed systems programming, a major library including great GUI support (§5.5), standard platform ABIs, elimination of overelaborate design warts that exist only for backward compatibility, etc. To get a serious candidate for regret, I think we have to leave the technical domain of language features and library facilities and consider social factors. The C++ community has no real center. Despite the major and sustained efforts of its members, the ISO C++ committee is unknown or incredibly remote to huge numbers of C++ users. Other programming language communities have been better at maintaining conferences, websites, collaborative development forums, FAQs, distribution sites, journals, industry consortiums, academic venues, teaching establishments, accreditation bodies, etc. I have been involved in several attempts to foster a more effective C++ community, but I must conclude that neither I nor anyone else have been sufficiently successful in this. The centrifugal forces in the C++ world have been too strong: Each implementation, library, and tool supplier has their own effort that usually does a good job for their users, but also isolates those users. Conferences and journals have fallen victim to economic problems and to a curious (but pleasant) lack of dogmatism among the leading people in the C++ community.

What did C++ get right? First of all, the ISO standard was completed in reasonable time and was the result of a genuine consensus. That's the base of all current C++ use. The standard strengthened C++ as a multi-paradigm language with uncompromising support for systems programming, including embedded systems programming.

Secondly, generic programming was developed and effective ways of using the language facilities for generic programming emerged. This not only involved work on templates (e.g., concepts) but also required a better understanding of resource management (§5.3) and of generic programming techniques. Bringing generic programming to the mainstream will probably be the main long-term technical contribution of C++ in this time period.

Can C++ be improved? and if so, how? Obviously, C++ can be improved technically, and I also think that it can be improved in the context of its real-world use. The latter is much harder, of course, because that imposes Draconian compatibility constraints and cultural requirements. The C++0x effort is the major attempt in that direction (§8). In the longer term (C++1x?), I hope for perfect type safety and a general high-level model for concurrency.

The question about improving the standards process is hard to answer. On the one hand, the ISO process is slow, bureaucratic, democratic, consensus-driven, subject to objections from very small communities (sometimes a single person), lacking of focus ("vision"), and cannot respond rapidly to changes in the industry or academic fashions. On the other hand, that process has been successful. I have sometimes summed up my position by paraphrasing Churchill: "the ISO standards process is the worst, except for all the rest". What the standards committee seems to lack is a "secretariat" of a handful of technical people who could spend full time examining needs, do comparative studies of solutions, experiment with language and library features, and work out detailed formulation of proposals. Unfortunately, long-time service in such a "secretariat" could easily be career death for a first-rate academic or industrial researcher. On the other hand, such a secretariat staffed by second-raters would lead to massive disaster. Maybe a secretariat of technical people serving for a couple of years supporting a fundamentally democratic body such as the ISO C++ committee would be a solution. That would require stable funding, though, and standards organizations do not have spare cash.

Commercial vendors have accustomed users to massive "standard libraries". The ISO standards process — at least as practiced by the C and C++ committees — has no way of meeting user expectations of a many hundred thousands of lines of free standard-library code. The Boost effort (§4.2) is an attempt to address this. However, standard libraries have to present a reasonably coherent view of computation and storage. Developers of such libraries also have to spend a huge amount of effort on important utility libraries of no significant intellectual interest: just providing "what everybody expects". There are also a host of mundane specification issues that even the commercial suppliers usually skimp on — but a standard must address because it aims to support multiple implementations. A loosely connected group of volunteers (working nights) is ill equipped to deal with that. A multi-year mechanism for guidance is necessary for coherence (and for integrating novel ideas) as well as ways of getting "ordinary development work" done.

9.3 Influences and impact

We can look at C++ in many ways. One is to consider influences:

1. What were the major influences on C++?
2. What languages, systems, and techniques were influenced by C++?

Recognizing the influences on the C++ language features is relatively easy. Recognizing the influences on C++ programming techniques, libraries, and tools is far harder because there is so much more going on in those areas and the decisions are not channeled through the single point of the standards committee. Recognizing C++'s influences on other languages, libraries, tools, and techniques is almost impossible. This task is made harder by a tendency of intellectual and commercial rivalries to lead to a lack of emphasis on documenting sources and influences. The competition for market share and mind share does not follow the ideal rules of academic publishing. In particular, most major ideas have multiple variants and sources, and people tend to refer to sources that are not seen as competitors — and for many new languages C++ is "the one to beat".

9.3.1 Influences on C++

The major influences on early C++ were the programming languages C and Simula. Along the way further influences came from Algol68, BCPL, Clu, Ada, ML, and Modula-2+ [121]. The language-technical borrowings tend to be accompanied by programming techniques related to those features. Many of the design ideas and programming techniques came from classical systems programming and from UNIX. Simula contributed not just language features, but ideas of programming technique relating to object-oriented programming and data abstraction. My emphasis on strong static type checking as a tool for design, early error detection, and run-time performance (in that order) reflected in the design of C++ came primarily from there. With generic programming and the STL, mainstream C++ programming received a solid dose of functional programming techniques and design ideas.

9.3.2 Impact of C++

C++'s main contribution was and is through the many systems built using it (§7 [137]). The primary purpose of a programming language is to help build good systems. C++ has become an essential ingredient in the infrastructure on which our civilization relies (e.g. telecommunications systems, personal computers, entertainment, medicine, and electronic commerce) and has been part of some of the most inspiring achievements of this time period (e.g., the Mars Rovers and the sequencing of the human genome).

C++ was preeminent in bringing object-oriented programming into the mainstream. To do so, the C++ community had to overcome two almost universal objections:

- Object-oriented programming is inherently inefficient
- Object-oriented programming is too complicated to be used by "ordinary programmers"

C++'s particular variant of OO was (deliberately) derived from Simula, rather than from Smalltalk or Lisp, and emphasized the role of static type checking and its associated design techniques. Less well recognized is that C++ also brought with it some non-OO data abstraction techniques and an emphasis on statically typed interfaces to non-OO facilities (e.g., proper type checking for plain old C functions; §7.6). When that was recognized, it was often criticized as "hybrid", "transitory", or "static". I consider it a valuable and often necessary complement to the view of object-oriented programming focused on class hierarchies and dynamic typing.

C++ brought generic programming into the mainstream. This was a natural evolution of the early C++ emphasis on statically typed interfaces and brought with it a number of functional programming techniques "translated" from lists and recursion to general sequences and iteration. Function templates plus function objects often take the role of higher-order functions. The STL was a crucial trendsetter. In addition, the use of templates and function objects led to an emphasis on static type safety in high-performance computing (§7.3) that hitherto had been missing in real-world applications.

C++'s influence on specific language features is most obvious in C, which has now "borrowed":

- function prototypes
- `const`
- `inline`
- `bool`
- `complex`
- declarations as statements
- declarations in for-statement initializers
- `//` comments

Sadly, with the exception of the `//` comments (which I in turn borrowed from BCPL), these features were introduced into C in incompatible forms. So was `void*`, which was a joint effort between Larry Rosler, Steve Johnson, and me at Bell Labs in mid-1982 (see D&E [121]).

Generic programming, the STL, and templates have also been very influential on the design of other languages. In the 1980s and 1990s templates and the programming techniques associated with them were frequently scorned as "not object-oriented", too complicated, and too expensive; workarounds were praised as "simple". The comments had an interesting similarity to the early community comments on object-oriented programming and classes. However, by 2005, both Java and C# have acquired "generics" and statically typed containers. These generics look a lot like templates, but Java's are primarily syntactic sugar for abstract classes and far more rigid than templates. Though still not as flexible or efficient as templates, the C# 2.0 generics are better integrated into the type system and even (as templates always did) offer a form of specialization.

C++'s success in real-world use also led to influences that are less beneficial. The frequent use of the ugly and irregular

C/C++ style of syntax in modern languages is not something I'm proud of. It is, however, an excellent indicator of C++ influence — nobody would come up with such syntax from first principles.

There is obvious C++ influence in Java, C#, and various scripting languages. The influence on Ada95, COBOL, and Fortran is less obvious, but definite. For example, having an Ada version of the Booch components [15] that could compare in code size and performance with the C++ version was a commonly cited goal. C++ has even managed to influence the functional programming community (e.g. [59]). However, I consider those influences less significant.

One of the key areas of influence has been on systems for intercommunication and components, such as CORBA and COM, both of which have a fundamental model of interfaces derived from C++'s abstract classes.

9.4 Beyond C++

C++ will be the mainstay of much systems development for years to come. After C++0x, I expect to see C++1x as the language and its community adapt to new challenges. However, where does C++ fit philosophically among current languages; in other words, ignoring compatibility, what is the essence of C++? C++ is an approximation to ideals, but obviously not itself the ideal. What key properties, ideas, and ideals might become the seeds of new languages and programming techniques? C++ is

- a set of low-level language mechanisms (dealing directly with hardware)
- combined with powerful compositional abstraction mechanisms
- relying on a minimal run-time environment.

In this, it is close to unique and much of its strength comes from this combination of features. C is by far the most successful traditional systems programming language; the dominant survivor of a large class of popular and useful languages. C is close to the machine in exactly the way C++ is (§2, §6.1), but C doesn't provide significant abstraction mechanisms. To contrast, most "modern languages" (such as Java, C#, Python, Ruby, Haskell, and ML) provide abstraction mechanisms, but deliberately put barriers between themselves and the machine. The most popular rely on a virtual machine plus a huge run-time support system. This is a great advantage when you are building an application that requires the services offered, but a major limitation on the range of application areas (§7.1). It also ensures that every application is a part of a huge system, making it hard to understand completely and impossible to make correct and well performing in isolation.

In contrast, C++ and its standard library can be implemented in itself (the few minor exceptions to this are the results of compatibility requirements and minor specification mistakes). C++ is complete both as a mechanism for dealing with hardware and for efficient abstraction from the close-to-hardware levels of programming. Occasionally, optimal use of a hardware feature requires a compiler intrinsic or an inline assembler insert, but that does not break the fundamental model; in fact, it can be seen as an integral part of that model. C++'s model of computation is that of hardware, rather than a mathematical or ad hoc abstraction.

Can such a model be applied to future hardware, future systems requirements, and future application requirements? I consider it reasonably obvious that in the absence of compatibility requirements, a new language on this model can be much smaller, simpler, more expressive, and more amenable to tool use than C++, without loss of performance or restriction of application domains. How much smaller? Say 10% of the size of C++ in definition and similar in front-end compiler size. In the "Retrospective" chapter of D&E [121], I expressed that idea as "Inside C++, there is a much smaller and cleaner language struggling to get out". Most of the simplification would come from generalization — eliminating the mess of special cases that makes C++ so hard to handle — rather than restriction or moving work from compile time to run time. But could a machine-near plus zero-overhead abstraction language meet "modern requirements"? In particular, it must be

- completely type safe
- capable of effectively using concurrent hardware

This is not the place to give a technical argument, but I'm confident that it could be done. Such a language would require a realistic (relative to real hardware) machine model (including a memory model) and a simple object model. The object model would be similar to C++'s: direct mapping of basic types to machine objects and simple composition as the basis for abstraction. This implies a form of pointers and arrays to support a "sequence of objects" basic model (similar to what the STL offers). Getting that type safe — that is, eliminating the possibility of invalid pointer use, range errors, etc. — with a minimum of run-time checks is a challenge, but there is plenty of research in software, hardware, and static analysis to give cause for optimism. The model would include true local variables (for user-defined types as well as built-in ones), implying support for "resource acquisition is initialization"-style resource management. It would not be a "garbage collection for everything" model, even though there undoubtedly would be a place for garbage collection in most such languages.

What kind of programming would this sort of language support? What kind of programming would it support better than obvious and popular alternatives? This kind of language would be a systems programming language suitable for hard real-time applications, device drivers, embedded devices, etc. I think the ideal systems programming language belongs to this general model — you need machine-near features, predictable performance, (type) safety, and powerful

abstraction features. Secondly, this kind of language would be ideal for all kinds of resource-constrained applications and applications with large parts that fitted these two criteria. This is a huge and growing class of applications. Obviously, this argument is a variant of the analysis of C++'s strengths from §7.1.

What would be different from current C++? Size, type safety, and integral support for concurrency would be the most obvious differences. More importantly, such a language would be more amenable to reasoning and to tool use than C++ and languages relying on lots of run-time support. Beyond that, there is ample scope for improvement over C++ in both design details and specification techniques. For example, the integration between core language and standard library features can be much smoother; the C++ ideal of identical support for built-in and user-defined types could be completely achieved so that a user couldn't (without examining the implementation) tell which was which. It is also obvious that essentially all of such a language could be defined relative to an abstract machine (thus eliminating all incidental "implementation defined" behavior) without compromising the close-to-machine ideal: machine architectures have huge similarities in operations and memory models that can be exploited.

The real-world challenge to be met is to specify and implement correct systems (often under resource constraints and often in the presence of the possibility of hardware failure). Our civilization critically depends on software and the amount and complexity of that software is steadily increasing. So far, we (the builders of large systems) have mostly addressed that challenge by patch upon patch and incredible numbers of run-time tests. I don't see that approach continuing to scale. For starters, apart from concurrent execution, our computers are not getting faster to compensate for software bloat as they did for the last decades. The way to deal that I'm most optimistic about is a more principled approach to software, relying on more mathematical reasoning, more declarative properties, and more static verification of program properties. The STL is an example of a move in that direction (constrained by the limitations of C++). Functional languages are examples of moves in that direction (constrained by a fundamental model of memory that differs from the hardware's and underutilization of static properties). A more significant move in that direction requires a language that supports specification of desired properties and reasoning about them (concepts are a step in that direction). For this kind of reasoning, run-time resolution, run-time tests, and multiple layers of run-time mapping from code to hardware are at best wasteful and at worst serious obstacles. The ultimate strength of a machine-near, type-safe, and compositional abstraction model is that it provides the fewest obstacles to reasoning.

10. Acknowledgments

Obviously, I owe the greatest debt to the members of the C++ standards committee who worked — and still work — on the language and library features described here. Similarly, I thank the compiler and tools builders who make C++ a viable tool for an incredible range of real-world problems.

Thanks to Dave Abrahams, Matt Austern, Paul A. Bistow, Steve Clamage, Greg Colvin, Gabriel Dos Reis, S. G. Ganesh, Lois Goldthwaite, Kevlin Henney, Howard Hinnant, Jaakko Järvi, Andy Koenig, Bronek Kozicki, Paul McJones, Scott Meyers, Nathan Myers, W. M. (Mike) Miller, Sean Parent, Tom Plum, PremAnand M. Rao, Jonathan Schilling, Alexander Stepanov, Nicholas Stroustrup, James Widman, and J. C. van Winkel for making constructive comments on drafts of this paper.

Also thanks to the HOPL-III reviewers: Julia Lawall, Mike Mahoney, Barbara Ryder, Herb Sutter, and Ben Zorn who — among other things — caused a significant increase in the introductory material, thus hopefully making this paper more self-contained and more accessible to people who are not C++ experts.

I am very grateful to Brian Kernighan and Doug McIlroy for all the time they spent listening to and educating me on a wide variety of language-related topics during the 1990s. A special thanks to Al Aho of Columbia University for lending me an office in which to write the first draft of this paper.

References

[1] David Abrahams: *Exception-Safety in Generic Components*. M. Jazayeri, R. Loos, D. Musser (eds.): Generic Programming, Proc. of a Dagstuhl Seminar. Lecture Notes on Computer Science. Volume 1766. 2000. ISBN: 3-540-41090-2.

[2] David Abrahams and Aleksey Gurtovoy: *C++ Template Meta-programming* Addison-Wesley. 2005. ISBN 0-321-22725-5.

[3] Andrei Alexandrescu: *Modern C++ Design*. Addison-Wesley. 2002. ISBN 0-201-70431.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.

[5] AT&T C++ translator release notes. *Tools and Reusable Components*. 1989.

[6] M. Austern: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison Wesley. 1998. ISBN: 0-201-30956-4.

[7] John Backus: *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. Communications of the ACM 21, 8 (Aug. 1978).

[8] J. Barreiro, R. Fraley, and D. Musser: *Hash Tables for the Standard Template Library*. Rensselaer Polytechnic Institute and Hewlett Packard Laboratories. February, 1995.

ftp://ftp.cs.rpi.edu/pub/stl/hashdoc.ps.Z

[9] Graham Birtwistle et al.: *SIMULA BEGIN*. Studentlitteratur. Lund. Sweden. 1979. ISBN 91-44-06212-5.

[10] Jasmin Blanchette and Mark Summerfield: *C++ GUI Programming with Qt3*. Prentice Hall. 2004. ISBN 0-13-124072-2.

[11] Hans-J. Boehm: *Space Efficient Conservative Garbage Collection*. Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices. June 1993. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[12] Hans-J. Boehm and Michael Spertus: *Transparent Garbage Collection for C++*. ISO SC22 WG21 TR NN1943==06-0013.

[13] Hans-J. Boehm: *An Atomic Operations Library for C++*. ISO SC22 WG21 TR N2047==06-0117.

[14] Hans-J. Boehm: *A Less Formal Explanation of the Proposed C++ Concurrency Memory Model*. ISO SC22 WG21 TR N2138==06-0208.

[15] Grady Booch: *Software Components with Ada*. Benjamin Cummings. 1988. ISBN 0-8053-0610-2.

[16] The Boost collection of libraries. http://www.boost.org.

[17] Walter Bright: *D Programming Language*. http://www.digitalmars.com/d/.

[18] Peter A. Buhr and Glen Ditchfield: *Adding Concurrency to a Programming Language*. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[19] David Butenhof: *Programming With Posix Threads*. ISBN 0-201-63392-2. 1997.

[20] T. Cargill: *Exception handling: A False Sense of Security*. The C++ Report, Volume 6, Number 9, November-December 1994.

[21] D. Caromel et al.: *C++//*. In *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[22] Fernando Cacciola: *A proposal to add a general purpose ranged-checked numeric_cast<>* (Revision 1). ISO SC22 WG21 TR N1879==05-0139.

[23] CGAL: Computational Geometry Algorithm Library. *http://www.cgal.org/*.

[24] Siva Challa and Artur Laksberg: *Essential Guide to Managed Extensions for C++*. Apress. 2002. ISBN: 1893115283.

[25] K. M. Chandy and C. Kesselman: *Compositional C++: Compositional Parallel Programming*. Technical Report. California Institute of Technology. [Caltech CSTR: 1992.cs-tr-92-13].

[26] Shigeru Chiba: *A Metaobject Protocol for C++*. Proc. OOPSLA'95. http://www.csg.is.titech.ac.jp/~chiba/openc++.html.

[27] Steve Clamage and David Vandevoorde. Personal communications. 2005.

[28] J. Coplien: *Multi-paradigm Design for C++*. Addison Wesley. 1998. ISBN 0-201-82467-1.

[29] Lawrence Crowl: *Thread-Local Storage*. ISO SC22 WG21 TR N1966==06-0036.

[30] Lawrence Crowl: *Defaulted and Deleted Functions*. ISO SC22 WG21 TR N2210==07-0070.

[31] Krzysztof Czarnecki and Ulrich W. Eisenecker: *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000. ISBN 0-201-30977-7.

[32] Beman Dawes: *Filesystem Library Proposal for TR2 (Revision 2)* . ISO SC22 WG21 TR N1934==06-0004.

[33] Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[34] D. Detlefs: *Garbage collection and run-time typing as a C++ library*. Proc. USENIX C++ conference. 1992.

[35] M. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual* ("The ARM") Addison Wesley. 1989.

[36] Dinkumware: *Dinkum Abridged Library* http://www.dinkumware.com/libdal.html.

[37] Gabriel Dos Reis and Bjarne Stroustrup: *Formalizing C++*. TAMU CS TR. 2005.

[38] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. Proc. ACM POPL 2006.

[39] Gabriel Dos Reis and Bjarne Stroustrup: *Generalized Constant Expressions* (Revision 3). ISO SC22 WG21 TR N1980==06-0050.

[40] The Embedded C++ Technical Committee: *The Language Specification & Libraries Version*. WP-AM-003. Oct 1999 (http://www.caravan.net/ec2plus/).

[41] Ecma International: *ECMA-372 Standard: C++/CLI Language Specification*. http://www.ecma-international.org/publications/standards/Ecma-372.htm. December 2005.

[42] Boris Fomitchev: *The STLport Story*. http://www.stlport.org/doc/story.html.

[43] Eric Gamma, et al.: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2.

[44] R. Garcia, et al.: *A comparative study of language support for generic programming*. ACM OOPSLA 2003.

[45] Jeff Garland: *Proposal to Add Date-Time to the C++ Standard Library*. ISO SC22 WG21 TR N1900=05-0160.

[46] N.H. Gehani and W.D. Roome: *Concurrent C++: Concurrent Programming with Class(es)*. Software—Practice and Experience, 18(12):1157—1177, December 1988.

[47] Geodesic: Great circle. Now offered by Veritas.

[48] M. Gibbs and B. Stroustrup: *Fast dynamic casting*. Software—Practice&Experience. 2005.

[49] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir: *MPI: The Complete Reference — 2nd Edition: Volume 2 — The MPI-2 Extensions* . The MIT Press. 1998.

[50] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[51] J. Gosling and H. McGilton: *The Java(tm) Language Environment: A White Paper*. `http://java.sun.com/docs/white/langenv/`.

[52] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: *Concepts: Linguistic Support for Generic Programming*. Proc. of ACM OOPSLA'06. October 2006.

[53] D. Gregor and B. Stroustrup: *Concepts*. ISO SC22 WG21 TR N2042==06-0012.

[54] D. Gregor, J. Järvi, G. Powell: *Variadic Templates* (Revision 3). ISO SC22 WG21 TR N2080==06-0150.

[55] Douglas Gregor and Jaakko Järvi: *Variadic Templates for C++*. Proc. 2007 ACM Symposium on Applied Computing. March 2007.

[56] R. H. Halstead: *MultiLisp: A Language for Concurrent Symbolic Computation*. TOPLAS. October 1985.

[57] H. Hinnant, D. Abrahams, and P. Dimov: *A Proposal to Add an Rvalue Reference to the C++ Language*. ISO SC22 WG21 TR N1690==04-0130.

[58] Howard E. Hinnant: *Multithreading API for C++0X - A Layered Approach*. ISO SC22 WG21 TR N2094==06-0164.

[59] J. Hughes and J. Sparud: *Haskell++: An object-oriented extension of Haskell*. Proc. Haskell Workshop, 1995.

[60] IA64 C++ ABI. `http://www.codesourcery.com/cxx-abi`.

[61] IDC report on programming language use. 2004. http://www.idc.com.

[62] *Standard for the C Programming Language*. ISO/IEC 9899. ("C89").

[63] *Standard for the C++ Programming Language*. ISO/IEC 14882. ("C++98").

[64] *Standard for the C Programming Language*. ISO/IEC 9899:1999. ("C99").

[65] International Organization for Standards: *The C Programming Language*. ISO/IEC 9899:2002. Wiley 2003. ISBN 0-470-84573-2.

[66] International Organization for Standards: *The C++ Programming Language* ISO/IEC 14882:2003. Wiley 2003. ISBN 0-470-84674-7.

[67] *Technical Report on C++ Performance*. ISO/IEC PDTR 18015.

[68] *Technical Report on C++ Standard Library Extensions*. ISO/IEC PDTR 19768.

[69] ISO SC22/WG21 website: `http://www.open-std.org/jtc1/sc22/wg21/`.

[70] Sorin Istrail and 35 others: *Whole-genome shotgun assembly and comparison of human genome assemblies*. Proc. National Academy of Sciences. February, 2004. `http://www.pantherdb.org/`.

[71] Jaakko Järvi: *Proposal for adding tuple type into the standard library*. ISO SC22 WG21 TR N1382==02-0040.

[72] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine: *The Lambda Library: Unnamed Functions in C++*. Software-Practice and Experience, 33:259-291, 2003.

[73] J. Järvi, B. Stroustrup and G. Dos Reis: *Deducing the type of a variable from its initializer expression*. ISO SC22 WG21 TR N1894, Oct. 2005.

[74] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek: *Algorithm specialization in generic programming: challenges of constrained generics in C++*. Proc. PLDI 2006.

[75] L. V. Kale and S. Krishnan: *CHARM++* in *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[76] Brian Kernighan and Dennis Richie: *The C Programming Language* ("K&R"). Prentice Hall. 1978. ISBN 0-13-110163-3.

[77] Brian Kernighan and Dennis Richie: *The C Programming Language (2nd edition)* ("K&R2" or just "K&R"). Prentice Hall. 1988. ISBN 0-13-110362-8.

[78] Brian Kernighan: *Why Pascal isn't my favorite programming language*. AT&T Bell Labs technical report No. 100. July 1981.

[79] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++(revised)*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990. Also, Journal of Object-Oriented Programming. July 1990.

[80] Christopher Kohlhoff: *Networking Library Proposal for TR2*. ISO SC22 WG21 TR N2054==06-0124.

[81] Mark A. Linton and Paul R. Calder: *The Design and Implementation of InterViews*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.

[82] A. Mishra et al.: R++: *Using Rules in Object-Oriented Designs*. Proc. OOPSLA-96. `http://www.research.att.com/sw/tools/r++/`.

[83] W. G. O'Farrell et al.: *ABC++* in *Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[84] Thorsten Ottosen: *Proposal for new for-loop*. ISO SC22 WG21 TR N1796==05-0056.

[85] Sean Parent: personal communications. 2006.

[86] J. V. W. Reynolds et al.: *POOMA* in *Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.

[87] Mike Mintz and Robert Ekendahl: *Hardware Verification with C++ — A Practitioner's Handbook*. Springe Verlag. 2006. ISBN 0-387-25543-5.

[88] Nathan C. Myers: *Traits: a new and useful template technique*. The C++ Report, June 1995.

[89] C. Nelson and H.-J. Boehm: *Sequencing and the concurrency memory model*. ISO SC22 WG21 TR N2052==06-0122.

[90] Mac OS X 10.1 November 2001 Developer Tools CD Release Notes: *Objective-C++*. `http://developer.apple.com/releasenotes/Cocoa/Objective-C++.html`

[91] Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in C++*. Internal AT&T Bell Labs Technical Memorandum. January 1984. Reprinted in AT&T C++ Translator Release Notes. November 1985.

[92] P. Rovner, R. Levin, and J. Wick: *On extending Modula-2 for building large integrated systems*. DEC research report #3. 1985.

[93] J. Schilling: *Optimizing Away C++ Exception Handling* ACM SIGPLAN Notices. August 1998.

[94] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra: *MPI: The Complete Reference* The MIT Press. 1995. `http://www-unix.mcs.anl.gov/mpi/`.

[95] D. C. Schmidt and S. D. Huston: *Network programming using C++*. Addison-Wesley Vol 1, 2001. Vol. 2, 2003. ISBN 0-201-60464-7 and ISBN 0-201-79525-6.

[96] Jeremy G. Siek and Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra* ISCOPE '98. `http://www.osl.iu.edu/research/mtl`.

[97] J. Siek et al.: *Concepts for C++*. ISO SC22 WG21 WG21-N1758.

[98] Jeremy G. Siek and Walid Taha: *A Semantic Analysis of C++ Templates*. Proc. ECOOP 2006.

[99] Yannis Smargdakis: *Functional programming with the FC++ library*. ICFP'00.

[100] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban: *AspectC++: an AOP Extension for C++*. Software Developer's Journal. June 2005. `http://www.aspectc.org/`.

[101] A. A. Stepanov and D. R. Musser: *The Ada Generic Library: Linear List Processing Packages*. Compass Series, Springer-Verlag, 1989.

[102] A. A. Stepanov: *Abstraction Penalty Benchmark, version 1.2 (KAI)*. SGI 1992. Reprinted as Appendix D.3 of [67].

[103] A. Stepanov and M. Lee: *The Standard Template Library*. HP Labs TR HPL-94-34. August 1994.

[104] Alex Stepanov: Foreword to Siek, et al.: *The Boost Graph Library*. Addison-Wesley 2002. ISBN 0-21-72914-8.

[105] Alex Stepanov: personal communications. 2004.

[106] Alex Stepanov: personal communications. 2006.

[107] Christopher Strachey: *Fundamental Concepts in Programming Languages*. Lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.

[108] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Revised, August 1981. Revised yet again and published as [109].

[109] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982. Revised version of [108].

[110] B. Stroustrup: *Data abstraction in C*. Bell Labs Technical Journal. Vol 63. No 8 (Part 2), pp 1701-1732. October 1984.

[111] B. Stroustrup: *A C++ Tutorial*. Proc. 1984 National Communications Forum. September, 1984.

[112] B. Stroustrup: *The C++ Programming Language* ("TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1986. ISBN 0-201-12078-X.

[113] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in *IEEE Software Magazine*. May 1988.

[114] Bjarne Stroustrup: *An overview of C++*. ACM Sigplan Notices, Special Issue. October, 1986

[115] B. Stroustrup and J. E. Shopiro: *A Set of Classes for Coroutine Style Programming*. Proc. USENIX C++ Workshop. November, 1987.

[116] Bjarne Stroustrup: quote from 1988 talk.

[117] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, CO. October 1988. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989.

[118] B. Stroustrup: *The C++ Programming Language*, 2nd Edition ("TC++PL2" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1991. ISBN 0-201-53992-6.

[119] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992. Revised version. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[120] B. Stroustrup: *A History of C++: 1979-1991*. Proc ACM HOPL-II. March 1993. Also in Begin and Gibson (editors): *History of Programming Languages*. Addison-Wesley. 1996. ISBN 0-201-89502-1.

[121] B. Stroustrup: *The Design and Evolution of C++*. ("D&E"). Addison-Wesley Longman. Reading Mass. USA. 1994. ISBN 0-201-54330-3.

[122] B. Stroustrup: *Why C++ is not just an object-oriented programming language*. Proc. ACM OOPSLA 1995.

[123] B. Stroustrup: *Proposal to Acknowledge that Garbage Collection for C++ is Possible*. WG21/N0932 X3J16/96-0114.

[124] B. Stroustrup: *The C++ Programming Language*, 3rd Edition ("TC++PL3" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1997. ISBN 0-201-88954-4.

[125] B. Stroustrup: *Learning Standard C++ as a New Language*. The C/C++ Users Journal. May 1999.

[126] B. Stroustrup: *The C++ Programming Language*, Special Edition ("TC++PL"). Addison-Wesley, February 2000. ISBN 0-201-70073-5.

[127] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, *C and C++: Case Studies in Compatibility*.

The C/C++ Users Journal. July, August, and September 2002.

[128] B. Stroustrup: *Why can't I define constraints for my template parameters?*. http://www.research.att.com/~bs/bs_faq2.html#constraints.

[129] B. Stroustrup and G. Dos Reis: *Concepts — Design choices for template argument checking*. ISO SC22 WG21 TR N1522. 2003.

[130] B. Stroustrup: *Concept checking — A more abstract complement to type checking*. ISO SC22 WG21 TR N1510. 2003.

[131] B. Stroustrup: *Abstraction and the C++ machine model*. Proc. ICESS'04. December 2004.

[132] B. Stroustrup, G. Dos Reis: *A concept design* (Rev. 1). ISO SC22 WG21 TR N1782=05-0042.

[133] B. Stroustrup: *A rationale for semantically enhanced library languages*. ACM LCSD05. October 2005.

[134] B. Stroustrup and G. Dos Reis: *Supporting SELL for High-Performance Computing*. LCPC05. October 2005.

[135] Bjarne Stroustrup and Gabriel Dos Reis: *Initializer lists*. ISO SC22 WG21 TR N1919=05-0179.

[136] B. Stroustrup: *C++ pages*. http://www.research.att.com/~bs/C++.html.

[137] B. Stroustrup: *C++ applications*. http://www.research.att.com/~bs/applications.html.

[138] H. Sutter, D. Miller, and B. Stroustrup: *Strongly Typed Enums* (revison 2). ISO SC22 WG21 TR N2213==07-0073.

[139] H. Sutter and B. Stroustrup: *A name for the null pointer: nullptr* (revision 2). ISO SC22 WG21 TR N1601==04-0041.

[140] Herb Sutter: *A Design Rationale for C++/CLI*, Version 1.1 — February 24, 2006 (later updated with minor editorial fixes). http://www.gotw.ca/publications/C++CLIRationale.pdf.

[141] Numbers supplied by Trolltech. January 2006.

[142] UK C++ panel: *Objection to Fast-track Ballot ECMA-372 in JTC1 N8037*. http://public.research.att.com/~bs/uk-objections.pdf. January 2006.

[143] E. Unruh: *Prime number computation*. ISO SC22 WG21 TR N462==94-0075.

[144] D. Vandevoorde and N. M. Josuttis: *C++ Templates — The Complete Guide*. Addison-Wesley. 2003. ISBN 0-201-73884-2.

[145] D. Vandevoorde: *Right Angle Brackets* (Revision 1). ISO SC22 WG21 TR N1757==05-0017 .

[146] D. Vandevoorde: *Modules in C++* (Version 3). ISO SC22 WG21 TR N1964==06-0034.

[147] Todd Veldhuizen: *expression templates*. C++ Report Magazine, Vol 7 No. 4, May 1995.

[148] Todd Veldhuizen: *Template metaprogramming*. The C++ Report, Vol. 7 No. 5. June 1995.

[149] Todd Veldhuizen: *Arrays in Blitz++*. Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'98). 1998.

[150] Todd Veldhuizen: *C++ Templates are Turing Complete* 2003.

[151] Todd L. Veldhuizen: *Guaranteed Optimization for Domain-Specific Programming*. Dagstuhl Seminar of Domain-Specific Program Generation. 2003.

[152] Andre Weinand et al.: *ET++ — An Object-Oriented Application Framework in C++*. Proc. OOPSLA'88. September 1988.

[153] Gregory V. Wilson and Paul Lu: *Parallel programming using C++*. Addison-Wesley. 1996.

[154] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974. ISBN 0-11-771600-6.

[155] *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd edition*, ISBN 1-930934-12-2. http://public.kitware.com/VTK.

[156] *FLTK: Fast Light Toolkit*. http://www.fltk.org/.

[157] Lockheed Martin Corporation: *Joint Strike Fighter air vehicle coding standards for the system development and demonstration program*. Document Number 2RDU00001 Rev C. December 2005. http://www.research.att.com/~bs/JSF-AV-rules.pdf.

[158] Microsoft Foundation Classes.

[159] *Smartwin++: An Open Source C++ GUI library*. http://smartwin.sourceforge.net/.

[160] *WTL: Windows Template Library —- A C++ library for developing Windows applications and UI components*. http://wtl.sourceforge.net.

[161] *gtkmm: C++ Interfaces for GTK+ and GNOME*. http://www.gtkmm.org/

[162] *wxWidgets: A cross platform GUI library*. http://wxwidgets.org/.

[163] Windows threads: *Processes and Threads*. http://msdn.microsoft.com/library/en-us/dllproc/base/processes_and_threads.asp.

[164] Wikipedia: *Java (Sun) — Early history*. http://en.wikipedia.org/wiki/Java_(Sun).

Lock-free Dynamically Resizable Arrays

Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup

Texas A&M University
College Station, TX 77843-3112
{dechev, peter.pirkelbauer}@tamu.edu, bs@cs.tamu.edu

Abstract. We present a first lock-free design and practical implementation of a dynamically resizable array (vector). The most extensively used container in the C++ Standard Library is *vector*, offering a combination of dynamic memory management and efficient random access. Our approach is based on a single 32-bit word atomic compare-and-swap (CAS) instruction and our implementation is portable to all systems supporting CAS, and more. It provides a flexible, generic, linearizable and highly parallelizable STL like interface, effective lock-free memory allocation and management, and fast execution. Our current implementation is designed to be most efficient on the most recent multi-core architectures. The test cases on a dual-core Intel processor indicate that our lock-free vector outperforms its lock-based STL counterpart and the latest concurrent vector implementation provided by Intel by a factor of 10. The implemented approach is also applicable across a variety of symmetric multiprocessing (SMP) platforms. The performance evaluation on an 8-way AMD system with non-shared L2 cache demonstrated timing results comparable to the best available lock-based techniques for such systems. The presented design implements the most common STL vector's interfaces, namely random access read and write, tail insertion and deletion, pre-allocation of memory, and query of the container's size.

Keywords: lock-free, STL, C++, vector, concurrency, real-time systems

1 Introduction

The ISO C++ Standard [19] does not mention concurrency or thread-safety (though it's next revision, C++0x, will [3]). Nevertheless, ISO C++ is widely used for parallel and multi-threaded software. Developers writing such programs face challenges not known in sequential programming: notably to correctly manipulate data where multiple threads access it. Currently, the most common synchronization technique is to use mutual exclusion locks. While conceptually simple, lock-based programs suffer performance and safety problems. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system by diminishing its parallelism. The behavior of mutual exclusion locks can sometimes be optimized by using fine-grained locks, context-switching locks [2], and reader/writer locks [25]. However, the interdependence of processes implied by the use of locks – even efficient locks – introduces the dangers of deadlock, livelock, and priority inversion. To many systems, the problem with locks is one of difficulty of providing correctness more than one of performance.

The widespread use of multi-core symmetric multiprocessing (SMP) architectures and the hardware support of multi-threading in some modern hardware platforms pose the challenge to develop effective, practical and robust concurrent data structures. The main target of our design is to deliver most optimal performance for such systems (Section 4). In addition, many real-time and autonomous systems, such as the Mission Data Systems Project at the Jet Propulsion Laboratory [8], require effective fine-grained synchronization. In such systems, the application of locks is a significantly complex and challenging problem due to the hazards of priority inversion and deadlock. Furthermore, the implementation of distributed parallel containers and algorithms such as STAPL[2] can benefit from a shared lock-free vector. The use of non-blocking (lock-free) techniques have been suggested to prevent the interdependence of the concurrent processes introduced by the application of locks. By definition, a lock-free concurrent data structure guarantees that when multiple threads operate simultaneously

on it, *some* thread will complete its task in a *finite* number of steps despite failures and waits experienced by other threads [13]. The vector is the most versatile and ubiquitous data structure in the C++ Standard Template Library (STL) [28]. It is a dynamically resizable array (dynamic table) that provides automatic memory management, random access, and tail element insertion and deletion with an amortized cost of O(1). [4].

This paper presents the following contributions:

(a) A first design and practical implementation of a lock-free dynamically resizable array. Our lock-free vector provides a set of common STL vector operations in a generic and linearizable [15] fashion. The presented design defines effective concurrent semantics and allows disjoint-access parallelism for some operations.
(b) A portable algorithm and implementation. Our design is based on the word-size compare-and-swap (CAS) instruction available on a large number of hardware platforms.
(c) A fast and space-efficient implementation. In particular, it avoids excessive copying [1] and on a variety of tests executed on an Intel dual-core architecture it outperforms its lock-based STL counterpart and the latest concurrent vector provided by Intel [18] by a factor of 10. In addition, a large number of tests executed on an 8-way AMD architecture with non-shared L2 cache indicate performance comparable to the best available lock-based techniques for such systems..
(d) An effective incorporation of non-blocking memory management and memory allocation schemes into the algorithm. Without such an integration a design is not useful in real-world applications.

The rest of the paper is structured like this: 2: Background, 3: Implementation, 4: Performance Evaluation, and 5: Conclusion.

2 Background

As defined by Herlihy [13], a concurrent object is *lock-free* if it guarantees that *some* process in the system will make progress in a *finite* number of steps. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. Lock-free and wait-free algorithms do not apply mutual exclusion locks. Instead they rely on a set of atomic primitives such as the word-size CAS instruction or Load-Linked (LL) / Store Conditional (SC) operations. Common CAS implementations require three arguments: a memory location,Mem, an old value,V_{old}, and a new value,V_{new}. The instruction's purpose is to atomically exchange the value stored in Mem with V_{new}, provided that its current value corresponds to V_{old}. The architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (e.g.: IA-32 [17]). The use of a hardware lock does not violate the non-blocking property as defined by Herlihy. Common locking synchronization methods such as semaphores, mutexes, monitors, and critical sections utilize the same atomic primitives to manipulate a control token. Such an application of the atomic instructions introduces interdependencies of the contending processes. In the most common scenario, lock-free systems utilize CAS in order to implement a speculative manipulation of a shared object. Each contending process speculates by applying a set of writes on a local copy of the shared data and attempts to CAS the shared object with the updated copy. Such an approach guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps. Therefore, the system is non-blocking and the performed operations could be implemented in a linearizable fashion at the expense of some extra work performed by the contending processes.

2.1 Pragmatic Lock-Free Programming

The practical implementation of lock-free containers is notoriously difficult: in addition to addressing the hazards of race conditions, the developer must also use non-blocking memory management and memory allocation schemes [14]. As explained in [1] and [5], a single-word CAS operation is inadequate for the practical implementation of a non-trivial concurrent container. A better design requires atomic

update of several memory locations. The use of a double-compare-and-swap primitive (DCAS) has been suggest by Detlefs et al. in [5], however it is rarely supported by the hardware architecture.

A software implementation of a multiple-compare-and-swap (MCAS) algorithm, based on CAS, has been proposed by Harris et al. [11]. This software-based MCAS algorithm has been effectively applied by Fraser towards the implementation of a number of lock-free containers such as binary search trees and skip lists [7]. The cost of this MCAS operation is relatively expensive requiring $2M + 1$ CAS instructions. Consequently, the direct application of this MCAS scheme is not an optimal approach for the design of lock-free algorithms. However, the MCAS implementation employs a number of techniques (such as pointer bit marking and the use of descriptors) that are useful for the design of practical lock-free systems. As discussed by Harris et al., a descriptor is an object that allows an interrupting thread to help an interrupted thread to complete successfully.

2.2 Lock-Free Data Structures

Recent research into the design of lock-free data structures includes linked-lists [30], [10], [21], [6], double-ended queues [20], [29], stacks [12], hash tables [21], [27] and binary search trees [7]. The problems encountered include excessive copying, low parallelism, inefficiency and high overhead. Despite the widespread use of the STL vector in real-world applications, the problem of the design and implementation of a lock-free dynamic array has not yet been discussed. The vector's random access, data locality, and dynamic memory management poses serious challenges for its non-blocking implementation. Our goal is to provide an effecient and practical lock-free STL-style vector.

2.3 Design Principles

We developed a set of design principles to guide our implementation:

(a) *thread-safety*: all data can be shared by multiple processors at all times.
(b) *lock-freedom*: apply non-blocking techniques to provide an implementation of thread-safe C++ dynamic array based on the current C++ memory model.
(c) *portability*: do not rely on uncommon architecture-specific instructions.
(d) *easy-to-use interfaces*: offer the interfaces and functionality available in the sequential STL vector.
(e) *high level of parallelism*: concurrent completion of non-conflicting operations should be possible.
(f) *minimal overhead*: achieve lock-freedom without excessive copying and minimize the time spent on CAS-based looping as well as the number of calls to CAS.
(g) *simplicity*: keep the implementation simple to allow model-based testing and verification.

The lock-free vector's design and implementation provided follow the syntax and semantics of the ISO STL vector as defined in ISO C++ [19]. The discussed implementation techniques in the following section can be applied towards the lock-free design and implementation of the complete set of ISO C++ vector's interfaces.

3 Algorithms

In this section we define a concurrent semantic model of the vector's operations, provide a description of the design and the applied implementation techniques, outline a correctness proof based on the adopted semantic model, address concerns related to memory management, and discuss some alternative solutions to our problem. The presented algorithms have been implemented in ISO C++ and designed for execution on an ordinary multi-threaded shared-memory system supporting only single-word read, write, and CAS instructions.

3.1 Concurrent Semantics

The semantics of the vector's operations are based on a number of assumptions. In the presented design, we assume that each processor can execute a number of the vector's operations. This establishes a *history* of invocations and responses and defines a real-time order between them. An operation O_1 is said to precede an operation O_2 if O_2*'s* invocation occurs after O_1*'s* response. Operations that do not have real-time ordering are defined as *concurrent*. The vector's operations are of two types: those whose progress depends on the vector's descriptor and those who are independent of it. We refer to the former as *descriptor-modifying operations* and to the latter as *non-descriptor modifying*. Each vector's operation in a set of concurrent descriptor-modifying operations S_1 is thread-safe and lock-free. The execution of the operations in S_1 always leads to a legal and desirable sequential history. A *sequential history* is defined as a history where an operation's invocation is immediately followed by its response. The non-descriptor modifying operations such as random access read and write are implemented through the direct application of atomic read and write instructions on the shared data. In a set of non-descriptor modifying operations S_2, all operations are thread-safe and wait-free. The execution of the set of operations $S_3 = S_1 \cup S_2$ leads to legal but not always desirable sequential history. The reason is the possibility to concurrently execute an operation $O_1 \in S_1$ that reduces the size of the vector (such as a tail deletion) and an operation $O_2 \in S_2$. Such a scenario can result in a legal but undesirable sequential history in the case when operation O_1's response precedes operation O_2's response. This limitation can be eliminated by transforming all non-descriptor modifying operations into descriptor-modyfing. We prefer not to do so in order to preserve the efficiency and wait-freedom of the current non-descriptor modifying operations. Instead we define the following:

Usage Rule: Non-descriptor modifying operations that access the tail should never be executed concurrently with descriptor modifying operations that reduce the vector's size.

3.2 Implementation Overview

The major challenges of providing lock-free vector implementation stem from the fact that key operations need to atomically modify two or more non-colocated words. For example, the critical vector operation `push_back` increases the size of the vector and stores the new element. Moreover, capacity-modifying operations such as `reserve` and `push_back` potentially allocate new storage and relocate all elements in case of a dynamic table [4] implementation. Element relocation must not block concurrent operations (such as `write` and `push_back`) and must guarantee that interfering updates will not compromise data consistency. Therefore, an update operation needs to modify up to four vector values: size, capacity, storage, and a vector's element.

Fig. 1. Lock-free Vector. T denotes a data structure parameterized on T.

The UML diagram in Fig. 1 presents the collaborating classes, their programming interfaces and data members. Each vector object contains the memory locations of the data storage of its elements as well as an object named "`Descriptor`" that encapsulates the container's size, a reference counter required by the applied memory management scheme (Section 3.5) and an optional reference to a write descriptor. Our approach requires that datatypes bigger than word size are indirectly stored

through pointers. Like Intel's concurrent vector [26], our implementation avoids the synchronization hazards during storage relocation by utilizing a two-level array. Whenever `push_back` exceeds the current capacity, a new memory block twice the size of the previous one is added. The semantics of the `pop_back` and `push_back` operations are guaranteed by the "`Descriptor`" object. Consider the case when a `pop_back` operation is interrupted by any matching number of `push_back` and `pop_back` operations. In a naïve implementation, the size of the vector would appear unchanged when the original `pop_back` resumes and the operation could produce an erroneous result. This is an instance of the ABA problem (Section 3.6). The use of a descriptor allows a thread-safe update of two memory locations thus eliminating the need for a DCAS instruction. An interrupting thread intending to change the descriptor, will need to complete any pending write. Not counting memory management overhead, this mechanism requires *two successful CAS* instructions to update *two memory locations* in the case of a `push_back` operation.

3.3 Operations

We have modeled the vector's interfaces according to the STL and implemented the following operations with the given interfaces and runtime guarantees (Tab. 1).

	Operations	Vector	Complexity
push_back	$Vector \times Elem \rightarrow void$	$Descriptor_t \rightarrow Descriptor_{t+1}$	$O(1) \times congestion$
pop_back	$Vector \rightarrow Elem$	$Descriptor_t \rightarrow Descriptor_{t+1}$	$O(1) \times congestion$
reserve	$Vector \times size_t \rightarrow Vector$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$
read	$Vector \times size_t \rightarrow Elem$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$
write	$Vector \times size_t \times Elem \rightarrow Vector$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$
size	$Vector \rightarrow size_t$	$Descriptor_t \rightarrow Descriptor_t$	$O(1)$

Table 1. Vector - Operations

The remaining part of this section presents the generalized pseudo-code of the implementation and hence omits code necessary for a particular memory management scheme. We use the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access respectively. The function $HighestBit$ returns the bit-number of the highest bit that is set in an integer value. On Intel's x86 processor $HighestBit$ corresponds to the BSR assembly instruction. `First_Bucket_Size` is a constant representing the capacity of the first bucket and corresponds to eight in our implementation.

Push_back (add one element to end) Since `push_back` is a descriptor-modifying operation, the first step is to complete a pending operation that the current descriptor might hold. Second, in case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Third, `push_back` defines a new "`Descriptor`" object and describes the current write operation. Finally, `push_back` uses a CAS to swap the previous "`Descriptor`" object with the new one. Should CAS fail, the routine is re-executed. After succeeding, `push_back` finishes by writing the element to the storage.

Pop_back (remove one element from end) Besides `push_back`, `pop_back` is the only other descriptor modifying operation. Unlike `push_back`, `pop_back` does not utilize a write descriptor. To provide a better correspondence to the concurrent semantics of the lock-free vector, we have slightly augmented the interface. In a sequential program, calling `read_i(size-1)` before `pop_back` reads and removes the last element. In a parallel environment, such semantics cannot be assumed. Hence, `pop_back` does not only remove the last element from the stack but also returns its value.

Reserve (increase allocated space) `Reserve` allocates new storage according to the description in Section 3.2. In the case of concurrently executing `reserve` operations, only one succeeds, while the others deallocate the acquired memory.

Algorithm 1 pushback $vector, elem$

```
repeat
    descriptor_current ← vector.descriptor
    CompleteWrite(vector, descriptor_current.pending)
    bucket ← HighestBit(descriptor_current.size + First_Bucket_Size) − HighestBit(First_Bucket_Size)
    if vector.memory[bucket] = NULL then
        AllocBucket(vector, bucket)
    end if
    writeop ← new WriteDesc(At(descriptor_current.size)^, elem, descriptor_current.size)
    descriptor_next ← new Descriptor(descriptor_current.size + 1, writeop)
until CAS(&vector.descriptor, descriptor_current, descriptor_next)
CompleteWrite(vector, descriptor_next.pending)
```

Algorithm 2 popback $vector$

```
repeat
    descriptor_current ← vector.descriptor
    CompleteWrite(vector, descriptor_current.pending)
    elem ← At(vector, descriptor_current.size − 1)^
    descriptor_next ← new Descriptor(descriptor_current.size − 1, NULL)
until CAS(&vector.descriptor, descriptor_current, descriptor_next)
return elem
```

Algorithm 3 Read $vector, i$

```
return At(vector, i)^
```

Algorithm 4 Write $vector, i, elem$

```
At(vector, i)^ ← elem
```

Algorithm 5 Reserve $vector, size$

```
i ← HighestBit(vector.descriptor.size + First_Bucket_Size − 1) − HighestBit(First_Bucket_Size)
if i < 0 then
    i ← 0
end if
while i < HighestBit(size + First_Bucket_Size − 1) − HighestBit(First_Bucket_Size) do
    i ← i + 1
    AllocBucket(vector, i)
end while
```

Algorithm 6 At $vector, i$

```
pos ← i + First_Bucket_Size
hibit ← HighestBit(pos)
idx ← pos xor 2^hibit
return &vector.memory[hibit − HighestBit(First_Bucket_Size)][idx]
```

Algorithm 7 CompleteWrite $vector, writeop$

```
if writeop.pending then
    CAS(At(vector, writeop.pos), writeop.value_old, writeop.value_new)
    writeop.pending ← false
end if
```

Read and Write at position i (use of subscripting) The random access `read` and `write` do not utilize the descriptor and their success is independent of the descriptor's value. The provided design corresponds to the STL's vector subscripting operations that do not offer bound checking.

Algorithm 8 Size *vector*

```
descriptor ← vector.descriptor
descriptor ← descriptor.size
if descriptor.writeop.pending then
    size ← size − 1
end if
return size
```

Algorithm 9 AllocBucket *vector*, *bucket*

```
bucketsize ← First_Bucket_Size^(bucket+1)
mem ← new T[bucketsize]
if not CAS(&vector.memory[bucket], NULL, mem) then
    Free(mem)
end if
```

Size (read number of elements) The `size` operations returns the size stored in the "`Descriptor`" minus a potential pending write operation at the end of the vector.

3.4 Correctness

The main correctness requirement for the semantics of the vector's operations is linearizability [15]. A concurrent operation is linearizable if it appears to execute instantaneously in a given moment of time between the time point t_1 of its invocation and the time point t_2 of its completion. The definition of linearizability implies that each concurrent history yields responses that are equivalent to the responses of some legal sequential history for the same requests. Additionally, the order of the operations within the sequential history should be consistent with the real-time order. In the context of the operations' semantics discussed earlier in Section 3.1, our proof of linearizability is straightforward by distinguishing the points where the execution of our operations appears to be atomic with respect to other operations.

Linearization Points

Let us assume that there is an operation $O_i \in S_{vec}$, where S_{vec} is the set of all possible vector's operations. We assume that O_i can be executed concurrently with n other operations $\{O_{op1}, O_{op2}..., O_{op\mathtt{n}}\} \in S_{vec}$. We provide a proof that operation O_i is linearizable. Assume O_i is a non-descriptor-modifying operation such as a random read or write. These operations perform a single atomic access directly to the shared data and their linearization point occurs naturally at the time point t_a when the atomic `read` or `write` instruction is executed. Assume O_i is a descriptor-modifying operation such as a `push_back` or a `pop_back`. Such operations are carried out in two stages: modify the "`Descriptor`" variable and then update the data structure's contents. Let us define that the time points t_{call}, and t_{return} represent the moments of time when O_i has been called and when it returns, respectively. Similarly, time points t_{desc} and $t_{writedesc}$ denote the instances of time when O_i executes an atomic update to the vector's "`Descriptor`" variable and when O_i's write descriptor is completed by O_i itself or another concurrent operation $O_c \in \{O_{op1}, O_{op2}..., O_{op\mathtt{n}}\}$, respectively. If the set $S_{tps} = \{T_{tp1}, T_{tp2}..., T_{tp\mathtt{n}}\}$ represents all instances of time in between t_{call} and t_{return}, it is clear that $t_{desc} \in S_{tps}$ and $t_{writedesc} \in S_{tps}$. As mentioned earlier, the `pop_back` operation does not need to utilize a write descriptor. In this case the linearization point of O_i is t_{desc}. In the case when O_i is a `push_back`, its linearization point is $t_{writeop}$.

Non-blocking We prove the non-blocking property of our implementation by showing that out of n threads at least one makes progress. Since the progress of non-descriptor modifying operations is independent, they are wait-free. Thus, it suffices to consider an operation O_1, where O_1 is either a `push_back` or `pop_back`. A write descriptor can be simultaneously read by n threads. While one of them will successfully perform the write descriptor's operation (O_2), the others will fail and not attempt it again. This failure is nonsignificant for the outcome of operation O_1. The first thread attempting to change the descriptor will succeed, which guarantees the progress of the system.

Testing We have executed a large number of test scenarios with our implementation. In addition, we have formalized the semantics of the lock-free vector as well as the atomic primitives using the SPIN model checker's modeling language Promela[16]. We have utilized SPIN in order to perform exhaustive testing and simulation of the vector's operations.

3.5 Memory Management

Our algorithms do not require the use of a particular memory management scheme. Assuming a garbage collected environment significantly reduces the complexity of the implementation. However, we are unaware of any available general lock-free garbage collector for C++.

Object Reclamation Our concrete implementation uses reference counting as described by Valois et al [30] and [24]. The scheme has two inherent drawbacks: each access to a managed object requires atomic modification of a counter. Second, a timing window allows objects to be reclaimed while a different thread is about to increase the counter. Consequently, objects cannot be freed but only recycled. We do not regard the latter problem as significant, since our usage pattern of the algorithm guarantees an upper bound of $2n$ number of objects in the system, where n is the number of concurrent threads operating on the data structure. Alternatively, we could also integrate Michael's hazard pointers [22], which use fewer atomic CAS instructions, reduce contention on the counter, and allow reclamation and arbitrary memory-reuse. This method requires to scan all threads for published pointers in use before objects can be freed. Although a correct selection of its parameters yields an amortized constant time for these scans, this technique does not uniformely distribute the overhead across the operations. The pass the bucket algorithm described by Herlihy et al. in [14] offers a solution presenting equivalent trade-offs in contrast to the reference counting schemes.

Allocator Recent research by Michael [23] and Gidenstam [9] presents implementations of true lock-free memory allocators. Due to its availability and performance, we selected Gidenstam's allocator for our performace tests.

3.6 The ABA Problem

The ABA problem is fundamental to all CAS-based systems. In our current implementation we have not incorporated a remedy to prevent it. The ABA problem occurs when a thread T_1 reads a value A from a shared object and then an interrupting thread T_2 modifies the value of the shared object from A to B and then back to A. When T_1 resumes, it erroneously assumes that the object has not been modified. Given such behavior, there is a serious risk that T_2 's execution is going to violate the correctness of the object's semantics. Practical solutions to the ABA problem include the use of hazard pointers [22] or the association of a version counter to each element in the platforms supporting a double-word compare-and-swap primitive (CAS2) such as IA-32 [17].

The semantics of the lock-free vector's operations can be corrupted by the occurrence of the ABA problem. Consider the following execution: assume a thread T_0 attempts to perform a *push_back* and stores in the vector's "`Descriptor`" object a write-descriptor announcing that the value of the object at position i should be changed from A to B. Then a thread T_1 interrupts and reads the write-descriptor. Later, after T_0 resumes and successfully completes the operation, a third thread T_2 can modify the value at position i from B back to A. When, T_1 resumes its CAS is going to succeed and erroneously execute the update from A to B. There are two particular instances when the ABA problem can affect the correctness of the vector's operations:

(1) the user intends to store a memory address value A in multiple locations.

(2) the memory allocator reuses the address of an already freed object.

The use of tags or version counters eliminates the dangers in both cases. However, this solution is applicable only on a limited set of architectures supporting a CAS2 instruction. The application of Michael's hazard pointers prevents the reuse of memory addresses that are likely to cause the ABA problem. Thus, hazard pointers solve the problem stated in Scenario 2. There are a number of possibilities in order to eliminate the vector's vulnerability to the ABA problem in the former

scenario. One option is to require the vector to copy the objects and store pointers to them. Such behavior complies with the STL value-semantics [28], however it can incur higher overhead in some cases due to the additional heap allocation and object construction. In a lock-free system, both the object construction and heap allocation can execute concurrently with other operations. In a scenario where the vector is under high contention, the additional overhead would not cause a significant delay of the operation's performance. In our future work we intend to evaluate the impact on performance of such ABA prevention techniques.

3.7 Alternatives

In this section we discuss several alternative designs for lock-free vectors.

Copy on Write In [1] Alexandrescu and Michael present a lock-free map, where every write operation creates a clone of the original map, which insulates modifications from concurrent operations. Once completed, the pointer to the map-representation is redirected from the original to the new map. The same idea could be adopted to implement a vector. Since the complexity of any write operation deteriorates to $O(n)$ instead of $O(1)$, this scheme would be limited to applications exhibiting read-often but write-rarely access patterns.

Using Software DCAS In [11] Harris et al. present a software multi-compare and swap (MCAS) implementation based on CAS instructions. While convenient, the MCAS operation is expensive (requiring $2M + 1$ CAS instructions). Thus, it is not the best choice for an effective implementation.

Contiguous storage Techniques similar to the ones used in our vector implementation could be applied to achieve a vector with contiguous storage. Compared to the presented approach, the major difference is that the storage area can change during lifetime. This requires `resize` to move all elements to the new location. Hence, storage and its capacity should become members of the descriptor. Synchronization between `write` and `resize` operations is what makes this approach difficult. One solution would make every vector-modification change the descriptor. In such case, the success of `resize` depends on the absence of concurrent writes. Furthermore, making all write operations modify the descriptor implies serialization of all writes and therefore diminishes the desired disjoint-access parallelism.

We discussed the descriptor- and non-descriptor modifying writes in the context of the two-level array and the contiguous storage vector. However, these write properties are not inherent in these two approaches. In the two-level array, it is possible to make each write operation descriptor-modifying, thus ensure a write within bounds. In the contiguous storage approach, element relocation could replace the elements with marked pointers to the new location. Every access to these marked pointers would get redirected to the new storage.

4 Performance Evaluation

We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC OS 10.4.6 operating system. In our performance analysis, we compare the lock-free approach (with its integrated lock-free memory management and memory allocation) with the most recent concurrent vector provided by Intel [18] as well as an STL vector protected by a lock. For the latter scenario we applied different types of locking synchronizations - an operating system dependent mutex, a reader/writer lock, a spin lock, as well as a queuing lock. We used this variety of lock-based techniques in order to contrast our non-blocking implementation to the best available locking synchronization technique for a given distribution of operations. We utilize the locking synchronization provided by Intel [18].

Similarly to the evaluation of other lock-free concurrent containers [7] and [21], we have designed our experiments by generating a workload of various operations (`push_back`, `pop_back`, random access `write`, and `read`). In the experiments, we varied the number of threads, starting from 1 and exponentially increased their number to 32. Every active thread executed 500,000 operations on the shared vector. We measured the CPU time (in seconds) that all threads needed in order to complete. Each

iteration of every thread executed an operation with a certain probability, `push_back` (`+`), `pop_back` (`-`), random access `write` (`w`), random access `read` (`r`). We use per-thread linear congruential random number generators where the seeds preserve the exact sequence of operations within a thread across all containers. We executed a number of tests with a variety of distributions and found that the differences in the containers' performances is generally preserved. As discussed by Fraser [7], it has been observed that in real-world concurrent application, the read operations dominate and account to about 70% to 75% of all operations. For this reason we illustrate the performance of the concurrent vectors with a distribution of `+`:15%, `-`:5%, `w`:10%, `r`:70% on Figure 2A. Similarly, Figure 2C demonstrates the performance results with a distribution containing predominantly writes, `+`:30%, `-`:20%, `w`:20%, `r`:30%. In these diagrams, the number of threads are plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. Both axes use locarithmic scale.

Fig. 2. Performace Results - Intel Core Duo

The current release of Intel's concurrent vector does not offer `pop_back` or any alternative to it. To include its performance results in our analysis, we excluded the `pop_back` operation from a number of distributions. Figure 2B and 2D present two of these distributions. For clarity we do not depict the results from the QueuingLock and SpinLock implementations. According to our observations, the QueuingLock performance is consistently slower than the other lock-based approachs. As indicated in [18], SpinLocks are volatile, unfair, and unscalable. They showed fast execution for the experiments with 8 threads or lower, however their performance significantly deteriorated with the experiments conducted with 16 or more active threads. To find a lower bound for our experiments we timed the tests with a non-thread safe STL-vector with preallocated memory for all operations. For example, in the scenario described in figure 2D, the lower bound is about a $\frac{1}{10}$ of the lock-free vector.

Our non-blocking implementation of the STL vector consistently outperforms the alternative lock-based approaches in all possible operation mixes by a significantly large factor (generally a factor of 10). It has also proved to be scalable as demonstrated by the performance analysis. These large performance gains have been consistently observed in all tested distributions of operations. Lock-free algorithms are particularly beneficial to shared data under high contention. It is expected that in a

scenario with low contention, the performance gains will not be as considerable as in the case of high contention.

Fig. 3. Performace Results - AMD 8-way Opteron

On systems without shared L2 cache, shared data structures suffer from performance degradation due to cache coherency problems. To test the applicability of our approach on such architecture we have performed the same experiments on an AMD 2.2GHz 8-way Opteron architecture with 1 MB L2 cache for each processor and 4GB shared RAM running the MS Windows 2003 operating system. (Fig.3). The applied lock-free memory allocation scheme is not available for MS Windows. For the sake of our performance evaluation we applied a regular lock-based memory allocator. The experimental results on this architecture lack the impressive performance gains we have observed on the dual-core L2 shared-cache system. However, the graph (Fig.3) demonstrates that the performance of our lock-free approach on such architectures is comparable to the performance of the best lock-based alternatives. An essential direction in our future work is to further optimize the effectiveness of our approach on such hardware architectures. This can be achieved by eliminating the bottlenecks introduced by the reference counting scheme applied for memory management. In the current scheme, each access to the container's tail requires a CAS-based manipulation of the descriptor's reference counter, invalidating copies in other caches. The memory latencies increase even more by the enqueing and dequeuing of descriptors at the head of a single freelist. We expect that the application of an alternative memory management scheme eliminate these problems [14] [22].

5 Conclusion

We presented a first practical and portable design and implementation of a lock-free dynamically resizable array. We developed a highly efficient algorithm that supports disjoint-access parallelism and incurs minimal overhead. To provide a practical implementation, our approach integrates non-blocking memory management and memory allocation schemes. We compared our implementation to the best available concurrent lock-based vectors on a dual-core system and have observed an overall speed-up of a factor of 10. The applied memory management is costly, thus we intend to integrate a better scheme. In addition, it is our goal to define the concurrent semantics of the remaining STL vector interface and incorporate them in our implementation.

References

1. A. Alexandrescu and M. Michael. Lock-free data structures with hazard pointers. *C++ User Journal*, November 2004.
2. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, pages 193–208, Cumberland Falls, Kentucky, aug 2001.

3. P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21N2009, April 2006.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.
5. D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. In *International Symposium on Distributed Computing*, pages 59–73, 2000.
6. M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM Press.
7. K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.
8. D. Garlan, W. K. Reinholtz, B. Schmerl, N. D. Sherman, and T. Tseng. Bridging the gap between systems design and space systems software. In *SEW '05: Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, pages 34–46, Washington, DC, USA, 2005. IEEE Computer Society.
9. A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *ESA*, pages 329–342, 2005.
10. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, London, UK, 2001. Springer-Verlag.
11. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.
12. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.
13. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.
14. M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.
15. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
16. G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
17. Intel. Ia-32 intel architecture software developer's manual, volume 3: System programming guide, 2004.
18. Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.
19. ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.
20. M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par 2003: The Ninth Euro-Par Conference on Parallel Processing, LNCS volume 2790*, pages 651–660, 2003.
21. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.
22. M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
23. M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2004. ACM Press.
24. M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, 1995.
25. M. Reiman and P. E. Wright. Performance analysis of concurrent-read exclusive-write. *SIGMETRICS Perform. Eval. Rev.*, 19(1):168–177, 1991.
26. A. Robison, Intel Corporation. Personal communication, April 2006.
27. O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 102–111, New York, NY, USA, 2003. ACM Press.
28. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
29. H. Sundell and P. Tsigas. Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap. In *OPODIS*, pages 240–255, 2004.
30. J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM Press.

Specifying C++ Concepts

Gabriel Dos Reis
Texas A&M University
gdr@cs.tamu.edu

Bjarne Stroustrup
Texas A&M University
and AT&T Labs — Research
bs@cs.tamu.edu

Abstract

C++ templates are key to the design of current successful mainstream libraries and systems. They are the basis of programming techniques in diverse areas ranging from conventional general-purpose programming to software for safety-critical embedded systems. Current work on improving templates focuses on the notion of *concepts* (a type system for templates), which promises significantly improved error diagnostics and increased expressive power such as concept-based overloading and function template partial specialization. This paper presents C++ templates with an emphasis on problems related to separate compilation. We consider the problem of how to express concepts in a precise way that is simple enough to be usable by ordinary programmers. In doing so, we expose a few weakness of the current specification of the C++ standard library and suggest a far more precise and complete specification. We also present a systematic way of translating our proposed concept definitions, based on use-patterns rather than function signatures, into constraint sets that can serve as convenient basis for concept checking in a compiler.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Logic and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Design, Performance

Keywords C++ Templates, C++ Concepts, Type Systems, Separate Compilation, Generic Programming

1. Introduction and overview

ISO Standard C++ [ISO03, Str00] directly supports generic programming through the notion of *template*. Templates are essential in C++ for capturing commonalities of abstractions while retaining optimal performance. Those properties are key to the success and the wide acceptance of the Standard Template Library [SL94]. Templates have also been used to reduce abstraction penalties and code bloat in embedded systems to an extent that is impractical in conventional object-oriented systems [Str04]. There are two key reasons for that. First, template instantiation combine information from both definition and use contexts (§2.4). This means that full information from both calling and called contexts (including full type information) is made available to the code generator. Current code generators are adept at using this information to minimize run time and/or code space. This contrasts to the usual case in object-oriented programming languages where the caller and the callee are completely separated through an interface relying on indirect function calls. Second, a C++ template is typically implicitly instantiated if and only if it is used in a way that is essential to the program semantics, automatically minimizing the footprint of an application (§2.4). This contrasts to systems that require the programmer to explicitly manage instantiation, such as Ada [TDBP00] or System F [Gir72, Rey74].

The work described here is part of an effort to design a type system — called *concepts* — for C++ types and values that can be used for template arguments as currently successfully used. It is a critical design concern for "concepts" that they should be usable by programmers who currently successfully use templates [SDR05a]. In some form or other, concepts will be part of the revised ISO C++ standard, C++0x [Str05]. This paper focuses on a specific problem: How to express concept definitions in a way that is sufficiently simple and flexible to be used, yet precise enough to be implementable in current C++ compilers [DRS05a]. Our notation for concept definition is based on "use patterns" that can be translated into a set of operation signatures suitable for type checking. Basically, concepts are compile-time predicates on types and values (e.g., integral constant values). They can be combined with the usual logical operators (*and*, *or*, *not*.)

This paper discusses concepts for C++ templates. However, the fundamental ideas generalize to a type system that supports parametric polymorphism, supports some forms of local type inference, and extends the notion of dependent names (§2.2). Our contributions include the development of a formal framework for specifying concepts, clarifications of the C++ standard iterator library requirements, and a precise formulation of iterator concepts. Concepts differ from type classes [WB89, HHPJW96] which also act as predicates over types. In particular, concepts cope with general overloading, including use of the same name for functions of different number of arguments and operations with argument types that are not generic instances of each other (§6).

The remaining of the paper is structured as follows. We

1. examine the fundamental properties of templates, the problems caused by those (§2) and the requirements on possible solutions (§2.5);

Copyright © 2006 ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *POPL'06* January 11–13, 2006, Charleston, South Carolina, USA.

2. introduce concepts (§3) based on "use patterns" and an algorithm for generating constraints sets for template arguments; apply it to simplified examples (§4);
3. apply our concepts to a known hard problem of significant practical importance: The specification of the lowest levels of the C++ standard iterator library, exposing some weakness (§5);
4. survey recent related works (§6) and briefly explain the differences between type classes and concepts;
5. outline several directions for future work and conclude (§7).

2. The problem

The near-optimal performance offered by ISO C++ templates comes at the price of very weak separation between template definitions and their uses. In C++ source code, the complete template definition is the only expression of the assumptions it makes about its parameters. However, it is clearly desirable to check a template definition independently of its uses, and to check the uses independently of the definition (see §2.3). To do that, we must concisely specify the assumptions separately from the code in the template definition. In short, we need a type system for template parameters. The holy grail of concept design for C++ is a system that allows for perfect separate checking of template definitions and uses, without loss of expressive power or performance. That is, if the definition of a template is sucessfully checked against the concepts specified for its parameters and if arguments specified in its uses are also successfully checked against those concepts, then the resulting instantiation will type-check and all information available will be used to generate optimal code. Please note that separate checking without optimal code generation is trivial: Just use some form of abstract class, as is often done explicitly in C++ and is the basis of the "generic" language facilities of Java and C#. However, the cost of doing so is to over-constrain solutions and to require unreasonable foresight of designers. In the absence of whole-program optimization (which is infeasible for many important C++ application domains), the abstract class approach also implies the use of indirect function calls as opposed to the inlining provided by templates.

2.1 Template basics

A template is a recipe from which a C++ translator generates declarations. For example, the program fragment

```
template<typename T>
T square(T x) { return x * x; }
```

declares a family of functions indexed by a type parameter. Like System F, a particular member of that family can be referred to by applying the template name `square` to a template-argument as in `square<int>`. We say that a template specialization is requested for `square` with template-argument list `<int>`. The process of creating a specialization is called template instantiation, so a specialization is also colloquially known as an instantiation. A C++ compiler would internally generate the moral equivalent of the function definition

```
int square(int x) { return x * x; }
```

where the type argument `int` is substituted for the type parameter `T`. The resulting code is type-checked to ensure that no type errors resulted from that substitution. Template instantiation is done only once for a given specialization even if the program contains multiple requests for that specialization.

Unlike programming languages like Ada or System F, template-argument list can be omitted from a function template instantiation request. Usually, the values for template-parameters are deduced. For example, given the definition above, the following is idiomatic C++:

```
double d = square(2.0);
```

The type-argument is deduced [Str00, ISO03] to be `double`. It worths noting that, unlike programming languages like Haskell [PJ03] or System F, C++ template parameters are not restricted to types. For example, the program fragment

```
template<typename T, size_t N>
struct buffer {
  T data[N];
};
```

declares a data type parameterized by an element type and a buffer-size, which can be any positive compile-time integral value.

2.2 Parameterization

Conceptually, the parameters of a template are specified in two ways:

1. *template-parameters* — explicitly mentioned as parameters in the template declaration; and
2. *dependent names* — inferred from the use of parameters in the definition of the template.

In C++, a name cannot be used without a prior declaration. This requires careful treatment of template definitions. For example, in the context of the definition of the template `square`, the symbol `*` has no visible declaration. However, in the instantiation context of `square<int>`, the compiler can resolve `*` to the (built-in) multiplication operator for `int` values. For a call `square(complex(2.0))`, `*` would be resolved to the (user-defined) multiplication operator for `complex` values. We say that the symbol `*` is a *dependent name* in the definition of `square`. That is, it is an implicit parameter of the template definition. Had we wanted to, we could have made the multiplication operations an explicit parameter like this:

```
template<typename Mul, typename T>
T square(T x) { return Mul()(x,x); }
```

Here, the sub-expression `Mul()` constructs a function object (see [Str00, Chapter 18]) that implements the multiplication operation over values of type `T`. The notion of dependent names helps keep the number of explicit parameters manageable.

2.3 Instantiations and checking

Only minimal semantics processing is done when a template definition or a template use is first seen. Full semantics processing is postponed to instantiation time (just before link time), on a *per instantiation* basis. That implies that assumptions made on template arguments are not checked before instantiation time. Consider

```
string hello = "Hello World";
square(hello);
```

The nonsensical use of a `string` as function-argument for `square` is not caught at the point of use. Only at instantiation time will the compiler discover that there is no suitable declaration for `*`. This is a huge practical problem because that instantiation may be caused by code written by a user

who wrote neither the definition of `square` nor the definition of `string`. A programmer who did not know the definition of `square` and `string` would have great difficulty understanding error messages relating to their interaction (such as "*illegal operand for* `*`").

The existence of an operator symbol `*` is not sufficient to ensure successful compilation for `square`. There must exist a `*` that takes arguments of the appropriate types and that `*` must be the unique best match under the C++ overload resolution rules [ISO03, Clause 13]. Furthermore, `square` takes its argument by-value and returns its result by-value. That is, it must be possible to copy objects of the deduced type. We need a rigorous framework for describing a template definition's requirements on its arguments. For space limitations and various practical reasons, such a formalism is described elsewhere [DRS05a].

Experience shows that successful compilation and linking may not be the end to our problems. A successful build only shows that the template instantiations were type correct given the arguments we passed it. What about template-argument types and values with which we didn't try to use it? The template definition may contain assumptions about its arguments that held for the arguments we gave it, but would fail for other, apparently reasonable, arguments. A simplified version of a classical example is

```
template<typename FwdIter>
bool palindrome(FwdIter first, FwdIter last)
{
  if (last <= first) return true;     // the middle
  if (*first != *last) return false;  // a difference
  return palindrome(++first, --last);
}
```

Here, we test whether a sequence, designated by a pair of iterators to its first and last elements, is a palindrome. The iterators are assumed to be of the *forward iterator category* (see §2.6), *i.e.*, they are assumed to support at least the operations `*`, `!=` and `++`. It is a widely used convention in the C++ community to use suggestive names such as `FowardIterator` (or a briefer `FwdIter`) for the template-parameters to convey assumptions. The definition of `palindrome` examines the sequence elements going from each end towards the middle. We can test it with a `vector`, with a C-style array, and with a `string`. In each case, our `palindrome` function template will instantiate and run correctly. Unfortunately, putting that `palindrome` into a library would be a time bomb: Not all sequences support `--` and `<=`. For example, singly-linked lists do not. Experts use elaborate systematic techniques to avoid such problems. However, the fundamental problem is that a template definition is not (by itself) a good specification of its requirements on its parameters. We need to make those requirements explicit and less ad hoc than the expression of an algorithm. "Concepts" are such requirements.

2.4 Performance

"Preserving the performance of the current template techniques" is a major requirement for templates with concepts. Users of concepts should not only get better ways of expressing ideas and better error messages when they make mistakes. They should also get performance at least equal to what is currently achieved using templates. After all, templates play a key role in C++ programming for performance critical applications. This performance has three sources:

- elimination of function calls in favor of inlining;
- combining information from several contexts for better optimization;
- avoiding generating code for unused functions.

The first point is not particular to templates but a general feature of C++ inline functions. However, inlining is crucial for fine-grained parameterization as is commonly used in the STL and other libraries relying on generic programming techniques. The performance in question is both run-time and memory footprint. When used well, as described here and used in major libraries, templates can simultaneously reduce both. The reduction in code size is especially important because on modern processors a reduction in code size implies a reduction in memory traffic and improved cache performance.

As an example, let's consider the function template `accumulate` from the Standard Library:

```
template<typename FwdIter, typename T>
T accumulate(FwdIter first, FwdIter last, T init)
{
   for (FwdIter cur = first; cur != last; ++cur)
      init = init + *cur;
   return init;
}
```

In other words, `accumulate` returns the sum of the elements in its input sequence with its third argument ("the accumulator") as the initial value. We might call it like this:

```
vector<complex<double>> v;
// ...
complex<double> z = 0;
z = accumulate(v.begin(),v.end(),z);
```

To do its job, `accumulate` will use addition and assignment operators on elements of type `complex<double>` and dereference `vector<complex<double>>` iterators. Adding `complex<double>` values involves addition of `double` values. To do this efficiently, all of these operations must be inlined. Both `vector` and `complex` are user-defined types; that is, they and their operations are defined somewhere in the C++ source code. Current C++ compilers handle this example so that the only function call generated is the one of `accumulate`. Access to `vector` members become simple load machine operation, addition of `complex` values becomes two floating-point add machine instructions, etc. To accomplish this, the compiler needs access to the complete definition of `vector` and `complex`. However, the result is a vast improvement (arguably optimal) over the naive approach of generating a function call for each use of an operation on a template parameter. Obviously, an add instruction executes much faster than a function call containing an addition. Furthermore, there is no function call preamble, passing of arguments, etc., so the resulting code is also much smaller.

Further reduction in generated code size is achieved by not emitting code for unused functions. The `vector` template class has many member functions that are not used by this example. Similarly, the `complex` class template has many member function and non-member supporting functions that are not used by this example. The C++ standard guarantees that no code is emitted for those unused functions.

To contrast, consider the more conventional case where arguments are accessed through interfaces defined as indirect function calls [Aug93, Str94]. Each operation then becomes a function call in the executable generated for user code such as `accumulate`. Furthermore, it would be distinctly non-trivial to avoid laying down code for unused (virtual)

member functions. It is beyond the ability of current C++ compilers and is likely to remain so for mainstream C++ programs where separate compilation and dynamic linking is the norm. This problem is not unique to C++; it is rooted in the fundamental difficulty of assessing which part of the source code is used and which is not when any form of run-time dispatch is in effect. Templates do not suffer this problem because their specializations are resolved at compile time.

This `accumulate` example is not quite perfect for illustrating the subtleties of generating object code from source code found in different parts of a program: It does not rely on implicit conversions or non-trivial overloading. However, consider a variant where `int` values are accumulated in a `complex<double>` object:

```
vector<int> v;
// ...
complex<double> s = 0;
s = accumulate(v.begin(),v.end(),s);
```

Here, the addition is done by converting the `int` value to a `double` value and then adding that to the accumulator, `s`, using a `+` operator of a `complex<double>` and a `double`. That's basically a simple floating-point addition. The point is that the `+` operator in `accumulate` depends on two template parameters and that it is the compiler's job to pick the most appropriate `+` based on all information about those two arguments. It would have been possible to maintain a better separation between the different contexts by always converting the element type to that of the accumulator. In this case, doing so would have resulted in the creation of an additional `complex<double>` for each element and an addition of two complex values. The code size and run time would have more than doubled.

We would not expect to see this last example directly in source code. If we saw it, we would consider it poorly written. However, equivalent code is common as the result of nested abstractions. It is especially important to generate good code in such cases because not doing so would be to discourage abstraction.

Please note that these optimizations are common place. Large amounts of real-world software depend on them. Consequently, improved type checking, as promised through the use of concepts, must not come at the expense of these optimizations.

2.5 A more realistic example

The `square` and `palindrome` examples are very simplistic cases where the assumptions can be stated on individual template-parameters in isolation. However, reality is more complicated than that. Consider `fill` from the Standard Library:

```
template<typename FwdIter, typename T>
void fill(FwdIter first, FwdIter last, const T& t)
{
   for (FwdIter cur = first; cur != last; ++cur)
     *cur = t;
}
```

In this definition, the symbols `!=`, `++`, `*` and `=` are dependent names. A call `fill(p, q, v)` will assign `v` to each element of the sequence defined by the interval $[p, q)$.

The ISO C++ rules for successful instantiation of that template require that values ι and τ for the type parameters `FwdIter` and `T` must fulfill the following assumptions:

1. Objects of type ι must be copy-initializable, so that they can be used as function arguments in calls to `fill`.
2. Two such objects must be equality comparable in the sense that the expression `cur != last` must be valid and its value convertible to the boolean type `bool`.
3. An expression of type ι must support the pre-increment operation.
4. The expression `*cur = t` must be valid (which implies that every sub-expression must also be valid).

For example, the following program fragment

```
vector<double> v(42);
fill(v.begin(), v.end(), 7);
```

constitutes a valid use of `fill` and the corresponding instantiation is type correct: The (deduced) template-arguments are iterator type for `vector<double>` and `int`, respectively, and all the constraints are satisfied. On the other hand, the function call in the fragment

```
int i = 0;
int j = 39;
fill(i, j, 43);
```

passes type checking, but produces errors during instantiation: The deduced type for the first template-argument, `int`, does not support the dereference operation (unary `*`). To diagnose that error, we need both the argument types (here, the built-in type `int`) and the body of the template definition (not just its declaration). This is the kind of error that we want immediately caught and reported at the point of call.

Finally consider, this fragment:

```
struct Generator {
   Generator(int);
   operator double();
   // ...
};

vector<int> v(42);
fill(v.begin(), v.end(), Generator(25));
```

This is valid. The result of `Generator(25)` can be converted to a `double` value (user-defined conversion) which can be converted to the `int` value (built-in conversion).

It follows that our type system must include a way to state assumptions on combinations of template-parameters. Such combinations often involve implicit user-defined and built-in conversions. Here is our current best bet for a convenient and compatible syntax:

```
template<Forward_iterator Iter, class T>
      where Assignable<Iter::value_type,T>
void fill(Iter first, Iter last, const T& t);
```

`Forward_iterator` and `Assignable` are concepts, *i.e.* compile-time predicates on types. For example, the boolean-valued expression `Assignable<double,int>` is true as we can assign an `int` value to a `double` object but `Assignable<double,string>` is false as we cannot assign a `string` value to a `double` object.

The checking of the use of `fill` proceeds as follows:

1. Deduce values ι and τ for the type parameters `Iter` and `T` from a call `fill(p, q, v)`.
2. Concept check; that is, evaluate `Forward_iterator<`ι`>` and `Assignable<`ι`::value_type, `τ`>`.
3. If the concept check succeeds, then type check the call.

Our problem now becomes how to provide a way to define such predicates. Everybody's first idea for that is to specify a concept as a set of operations with signatures. However, we found that specifying such complete sets of operations was feasible only for small examples or given incredible amounts of time and patience [SDR03a, SDR03b, DRS05a, DRS05b]. Producing the complete list of operations — complete with conversions, overloads, etc. — is distinctly non-trivial for real-world examples. For example, consider a few ways that an addition operator for a type `X` might be defined:

```
+    // built-in operation for X
X operator+(X,X);
X operator+(const X&, const X&);
X X::operator+(const X&) const;
X X::operator+(X);
const X& X::operator+(const X&);
const N& operator+(const& N, const N&); // X converts to N
X operator+(X, N); // N converts to X
```

In real-world code, conventions vary for operators, such as `+`, and often the variation reflects reasonable design choices. Conventions vary even more for named functions. Nevertheless, an algorithm written as a simple template function can just use `+` and rely on optimal code (without spurious conversions and indirections) being generated for each kind of definition of `+`.

When several operations are used in combination, a combinatorial explosion of possibilities can occur. For example, the resolution of the simple and common expression `*p++ = v` involves 3 explicit operations, each of which can be specified with a variety of signatures. In addition, it's easy to imagine auxiliary type being introduced for the subexpressions `p++`, and `*p++`. Furthermore, each of the operations `++`, `*`, and `=` may require a conversion operator to bring its argument(s) to the required type(s). In all, we may have to use 2 auxiliary types and $3+4$ operations (or $3+3*4$ operations if we count built-in conversions and user-defined conversions separately in each possible conversion), each of which we would have to explicitly specify if we used a primitive signature-based approach.

2.6 An iterator concept for `fill`

We must distinguish between the *internal form* and *external form* of a language. The internal language is usually biased toward implementations whereas the external language is directed towards programmers for use in source program. In this context, the external form of the concept system must be simple and flexible enough to cope with millions of lines of existing code in the hands of hundreds of thousands of programmers. On the other hand, the internal form must be precise and straightforward enough for use in compilers (including being retrofitted into existing compilers). One solution is to define a special language for defining "abstract signatures" [DR02, SDR03a, SGG+05]. Another solution, the one we describe here, is to generate sets of "primitive operations" with signatures from a notationally simpler and more abstract language, that we call "use patterns". In the use pattern approach, `*p++=v` is the user's notation for requiring the caller to (somehow, usually indirectly) supply a set of types and operations to type-safely compile expressions similar to `*p++=v`.

Before digging into technical difficulties of precise and concise concept definitions, let us illustrate the general idea with the concept `Mutable_fwd` defined below. This is a "first attempt" that will only serve `fill`, but it will get us started and provide a basis for refinements (see §4 and §5). The concept definition must express assumptions needed to separately check the definition of the template `fill`.

```
concept Mutable_fwd<typename Iter, typename T> {
  Var<Iter> p;       // a variable of type Iter.
  Var<const T> v;    // a variable of type  const T.

  Iter q = p;        // an Iter must be copy-able

  bool b = (p != q); // must support ``!=’’ operation,
                     // and the resulting expression
                     // must be convertible to ``bool’’

  ++p;               // must support pre-increment, no
                     // requirements on the result type

  *p = v;            // must be able to dereference p,
                     // and assign a ``const T’’ to the
                     // result of that dereference; no
                     // requirements on the result type
};
```

A concept definition is introduced with the keyword `concept`, followed by the concept-name, the declarations of the parameters for the concept (in template-parameter list notation, to emphasize the compile-time nature of concepts.) The set of assumptions, represented by a concept, are expressed as ordinary — if somewhat stylized — C++.

`Mutable_fwd` is a binary predicate that expects types as arguments, corresponding to the type parameters `Iter` and `T`. The notation "`Var<Iter> p;`" introduces the name `p` for a variable of type `Iter`. We could not just write "`Iter p;`" because that would imply default initialization — and we need not require that an `Iter` supports default initialization. The declaration of `q` states that it must be possible to copy-initialize a variable of type `Iter`. The declaration of `b` states that the symbol `!=` is required and the resulting expression must be implicitly convertible to `bool`. We also require pre-increment, but impose no requirement on the type of the result (except that it must be valid C++ type). The last line is interesting because it states a requirement involving both parameter types: It must be possible to dereference an expression of type `Iter` and to assign a `const T` to the result.

During checking of a *template definition* guarded by the concept `Mutable_fwd`, the compiler makes sure that all syntax trees it creates respect the type assumptions listed above. We can use this to check the definition of `fill`:

```
template<typename Iter, typename T>
   where Mutable_fwd<Iter, T>
void fill(Iter first, Iter last, const T& t)
{
   for (Iter cur; cur != last; ++cur)
     *cur = t;
}
```

As expected, this definition concept checks. The reasons are:

1. An `Iter` is copy-initializable, therefore it can serve as a function parameter — in other words, the declarations for `first` and `last` are well-formed.
2. The declaration for `t` is well-formed, because the type parameter `T` is assumed to be an object type according to the concept `Mutable_fwd`.
3. Since two `Iter`s are comparable and the result is implicitly convertible to a `bool`, it is valid to write the expression `cur!=last` in the second part of the `for`-loop header.
4. Pre-increment is part of the assumptions.

5. Since a reference of type `const T&` is indistinguishable from a variable of type `const T` in expression contexts, it follows that it is legitimate to use the reference `t` in the assignment expression `*cur=t`.

It can be argued that this long justification is unnecessary since we have abstracted the expressions so that the concept `Mutable_fwd` gives us exactly the code we want to write. It is nevertheless instructive to go through those trivialities before everything gets obscured by the technicalities. In particular, please note the use of implicit conversion, copy-initialization and substitution of reference for variables.

Using the concept `Mutable_fwd` (only and not also the definition of `fill`), we can check *uses* of `fill`. For example:

```
vector<double> v(42);
fill(v.begin(), v.end(), 7);
```

First, the compiler deduces `vector<double>::iterator` for the type parameter `Iter` and `int` for `T`. Then it goes on checking for the satisfiability of the predicate `Mutable_fwd` with the argument list `<vector<double>::iterator,int>`, which succeeds with the appropriate operations.

Specifying requirements as use patterns has a long story in C++ programming [Str94] and adopted in the ISO C++ standard. However, such usage has been informal and conventional, we promote it to a formally defined and automated mechanism supported by language constructs. From use patterns, we derive sets of primitive, easy to use in checking, constraints that template arguments must fulfill. Note that this difficult step is already done by C++ compilers as part of template instantiations.

2.7 Iterator concepts

Throughout this paper, we draw examples from the theory of iterators [SL94, Aus98, Str00, ISO03]. We emphasize that the iterator classification is just one example (albeit important one for programming in C++ using the Standard Library). Other sources of inspiration include the theory of mathematical structures in computer algebra [JS92].

The C++ standard library contains a classification of iterators, which are divided into five major categories: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators* and *random access iterators*. See [Str00, Chapter 19] and [ISO03, Clause 24] for detailed exposition. We base our discussion on a particularly simple, yet difficult example from that classification. However, to follow the discussion here, the reader needs only a few key observations: An input iterator is a data-source abstraction and an output iterator is a data-sink abstraction. Each provides an operation called `++` to *advance* to the next element of a sequence. A forward iterator supports the notion of multi-pass algorithms (in particular, it is copyable) and an input iterator does not. A forward iterator that is mutable fulfills both input and output iterator assumptions. Iterators are pervasive in performance-critical code and optimal performance is expected. This implies that we can't impose significant overheads, such as a function call per operation, on iterators.

3. A concept system

The "use pattern"-based concept system presented here is based on the observation that a C++ expression, such as `*p++`, as it appears in a template is far more abstract, general, and readable than the set of operations and auxiliary types needed to implement it. Similarly, we can represent type predicates (concepts) as C++ code. C++ compilers typically represent template definitions as parse trees. Using identical compiler techniques, we can convert concepts to parse trees as well. Given that, we can implement concept checking as abstract tree matching. A convenient way of implementing this matching is to generate and compare sets of required functions and types (called "constraint sets", see §3.4) from templates and concepts definitions.

A concept definition is a set of abstract syntax tree equations with type assumptions. Concepts serves two purposes:

1. In *template definitions*, concepts act as typing judgment rules. If an abstract syntax tree depends on template parameters and cannot be resolved by the surrounding typing environment, then it must appear in the guarding concept bodies. Such dependent abstract syntax trees are implicit parameters of the concepts and will be resolved by concept checking at use sites.
2. In *template uses*, concepts act as set of predicates that the template-arguments must satisfy. Concept checking resolves implicit parameters at instantiation points.

So, if the set of concepts for a template definition specifies too few operations, the compilation of the template will fail concept checking: The template is under-constrained. Conversely, if the set of concepts for a template definition specifies more operations than needed, some otherwise legitimate uses may also fail concept checking: The template is over-constrained. By "otherwise legitimate" here, we mean that type checking would have succeeded in the absence of concept checking.

3.1 Concept definition

The concrete syntax for a concept definition is

`concept` *ConceptName*`<`P`> where` G `{` B `};`

It is a triple $\langle \mathsf{P}, \mathsf{G}, \mathsf{B} \rangle$ where:

- P is a list of explicit concept-parameters, with exactly the same declaration syntax as same for template-parameters. In particular, non-type concept-parameters are considered compile-time values (as the case in the current template system) in the scope of a concept definition.
- G is the "guard." It is a logical formula, made of compile-time expressions combined with the usual logical operators. The `where`-clause is optional. It usually expresses additional assumptions on combinations of the parameters P.
- B is the body of the concept. It is a sequence of simple declarations and expression-statements that enunciate syntax and type equations between the concept-parameters.

For example, here is a concept that captures the notion of *small object type* relative to some maximum size (also a parameter to the concept):

```
concept Small<typename T, int N>
   where sizeof (T) <= N
{ };
```

`Small`'s body is empty: the only "interesting" information is in the `where`-clause. More complicated concepts will involve constraints on individual arguments and in the body. A guard involves only compile-time expressions. In particular, a `sizeof`-expression is a compile-time value, and the non-type concept-parameter `N` is a compile-time expression.

A concept is a compile-time predicate. Therefore concepts can be combined with the logical operators and used in

where-clauses. As a short cut, concepts usable as unary predicates can also be used as the type of template-parameter. For example:

```
concept C<typename X>
   where C1<X> && C2<X>
{ };
```

is equivalent to

```
concept C<C1 X>
   where C2<X>
{ };
```

which again is equivalent to

```
concept C<C1&&C2 X>
{ };
```

A template type parameter introduced with the keyword typename is unconstrained. That is, any type can be used as an argument. Code involving such parameters cannot be concept checked. This "loophole" leaves existing code using templates valid, with its meaning unchanged. This compatibility feature is essential for the adoption of concepts into the C++ standard. There also appears to be uses for such completely unconstrained types in template meta-programming where properties of an argument are sometimes not needed until deep in a sequence of instantiations

3.2 Explicit check request

Programmers can explicitly ask the compiler to check conformance of a specific combination of types and values with respect to a concept. The syntax for that is either

```
assert Concept<argument-list>;
```

or

```
assert Concept<argument-list> with {
     declaration sequence
};
```

The first form is primarily used for checking whether a concept-argument list satisfies a concept. The program is considered in error if concept checking fails with those arguments. Otherwise, the translation of the program is carried on. For example, the explicit check

```
assert Mutable_fwd<double*, int>;
```

succeeds and the following values are deduced for the implicit parameters:

1. *CopyInitialize* is the built-in copy operator for values of type double*, and built-in copy operator for bool values;
2. *Convert* is the identity conversion on double*, and on the built-in conversion operator from int and double;
3. != is built-in inequality comparison operator for double* values;
4. prefix ++ is the usual built-in pre-increment operator on pointers to double;
5. unary * is built-in dereference operator on double*;
6. and = is built-in assignment operator on double.

The second form of explicit check request is used in situations where it is necessary to "rewrite" syntax for the check to succeed. For example, the pointer type int* does not have members so it is necessary to map the abstract requirement of member syntax to something appropriate when checking for random access iterator properties:

```
assert Random_access<int*> with {
  int*::value_type = int;   // init*'s value_type is int
};
```

More generally, an explicit concept conformance assertion can be written for a family of types and values using one of

```
assert template<parameter-list> where G
          Concept<argument-list>;

assert template<parameter-list> where G
          Concept<argument-list> with {
              declaration sequence
          };
```

In either case, the where-clause is optional, just like in a template or concept declaration. For instance, instead of trying to request explicit check Random_access<T*> for pointer type T* individually for each type T, it is better to write the assertion in a template form

```
assert template<typename T> Random_access<T*> with {
  T*::value_type = T;  // T*'s value type is T
};
```

With that assertion, when the compiler needs to evaluate Random_access<A*> for a given type argument A it first (internally) generates the equivalent of

```
assert Random_access<A*> with {
  A*::value_type = A;
};
```

then evaluate that assertion, like in a non-template case. In particular, the program is erroneous if the generated assertion fails.

3.3 Implicit check request

Implicit checking of concepts typically happens in situations where an implicit instantiation is requested for a function template and the declaration is guarded by concepts like in

```
fill(v.begin(), v.end(), 42);
```

assuming the declaration of fill from §2.6. If concept checking fails for the deduced template-arguments then the function is disregarded; this failure is considered an error only if no declaration of the function can match the use.

For example, consider a situation where a function f needs to allocate a temporary object for its internal work. If the object is small we can allocate it in f's activation record (e.g., execution stack). That is usually far more efficient than allocation from free store. What constitutes "small" is determined by the program's need, execution environment, and other implementation considerations. We can define f as overloaded on the size of its argument type:

```
const int max_size = 256;   // implementation-defined
// declare two variants of f().
template<typename T> where Small<T, max_size>
  void f(const T&);         // #1
template<typename T> where !Small<T, max_size>
  void f(const T&);         // #2
 int main()
 {
    buffer<char, 32> sbuf;
    f(small_buf);           // call #1
    buffer<char, 1024> lbuf;
    f(lbuf);                // call #2
 }
```

The function f is overloaded because where-clauses are part of function template signatures. The construction of overload set is very similar to the way implicit instantiation of function

template works in C++. The difference here is that we have added an additional step for concept check, before overload resolution proceeds.

3.4 From concepts to constraints sets

How is a concept definition — assumptions written in very abstract forms — turned into notations or requirements suitable for checking with conventional compiler technology?

A concept definition $\langle \mathsf{P}, \mathsf{G}, \mathsf{B} \rangle$ is further processed and refined into a quadruple $\langle \mathsf{P}_{\text{exp}}, \mathsf{P}_{\text{impl}}, \mathsf{G}, \mathsf{C} \rangle$ for the purpose of concept checking. The components are determined as follows:

1. A set of explicit parameter declarations P_{exp} as P in the definition, where properties implied by each nominated concept are assumed to hold for the corresponding parameter.
2. A list P_{impl} of dependent names.
3. A guard G, as in the definition. This predicate is assumed to be true during the definition of the concept body and checking of template definition.
4. A sequence of constraint equations C derived from the body B of the concept definition. The constraints are constructed reading "backwards" the static type rules of ISO C++.

Let's illustrate the general idea with the definition of `Mutable_fwd` from §2.6. The set of explicit parameters and implicit parameters are $\mathsf{P}_{\text{exp}} = \{\texttt{Iter}, \texttt{T}\}$ and

$$\mathsf{P}_{\text{impl}} = \{\textit{CopyInitialize}, \textit{Convert}, \texttt{!=}, \text{prefix } \texttt{++}, \text{unary } \texttt{*}, \texttt{=}\},$$

respectively. The symbol *CopyInitialize* and *Convert* are not explicitly mentioned; they are implied and overloaded at several places in the concept definition as detailed below. The implicit parameters are resolved (through name lookup) when deducing the values for the explicit parameters. In the definition of `fill`, the compiler assumes that the set of parameters is $\mathsf{P}_{\text{exp}} \cup \mathsf{P}_{\text{impl}}$. The constraint set generated from the body of `Mutable_fwd` was surveyed in §2.6 and is now detailed as follows:

1. The declaration "`Var<Iter> p`", introducing `p` as the name of a variable, requires `Iter` to be an object type.
2. Similarly, the declaration "`Var<const T> v`" generates the constraint that `const T`, hence `T`, is an object type.
3. The declaration `Iter q = p;` uses the C++ syntax of copy-initialization; consequently it generates the copy-initialization constraint. We denote that operation by *CopyInitialize*. It is dependent on the concept parameter `Iter`, therefore it is added to the list of dependent names.
4. The declaration `bool b = (p != q)` also generates a copy-initialization constraint. The type of the initializer in that declaration depends on a concept-parameter; therefore the *CopyInitialize* symbol required is also added to the dependent name list.

   The initializer requires the existence of a symbol `!=`, which in this case is also a dependent name. Furthermore, the argument types of `!=` need not be `Iter`. We only need that both operands be convertible to the types expected by `!=`. Consequently, we generate the corresponding implicit conversion constraints. Implicit conversion is denoted by the symbol *Convert*. The implicit conversions needed for both operands depend on a concept-parameter, so *Convert* is added to the list of dependent names.
5. The pre-increment expression `++p` requires the existence of a prefix `++` operator, added to the dependent name list. That expression also generates an implicit conversion constraint, that converts an expression of type `Iter` to the argument type of the pre-increment operator.
6. Finally, the expression `*p = v` requires the dereference operator and the assignment operators (added to the dependent name list). It also generates three implicit conversion constraints: One to convert unary `p` to the argument type of `*`, and two for converting `*p` and `v` to the argument types of `=`.

Please note that checking for copy-initialization and implicit conversion are simple operations done in C++ compilers.

3.5 Concept checking

When checking for the satisfiability of predicates at the template use site with typing environment Γ, the program translator recursively applies the following steps:

1. Substitute the concept-arguments for the concept-parameters P in the environment Γ, the concept guard G and body B.
2. If the guard evaluates to false, then concept check fails.
3. Look up the dependent names in the environment Γ. If lookup fails for a name, then concept check fails.
4. The result of name lookup for dependent names add additional equations for constraints variables introduced for symbols in function calls. Solve those equations through overload resolution. If overload resolution fails, then concept check fails.

When concept check succeeds, it should produce

1. a new typing environment;
2. a substitution, mapping dependent names to actual declarations;
3. and set of rewrite rules necessary for implicit conversions as required.

That triple is then used to produce instantiations from template. Only after concept checking succeeds, is type checking carried on. A nontrivial step here is to ensure that if the definition of the template is concept-correct, then the substitutions and rewrites resulting from concept and type checking of its uses will be well-formed in the new typing environment.

3.6 Associated types and values

An associated type or value is the value of a constraint variable, or an implicit parameter. They need not be named. For example, in the concept `Mutable_fwd` the type of the expression `*p` is an associated type. It is colloquially known as the *value type* of the iterator `Iter`. It is associated to `Mutable_fwd`, but it is not an explicit parameter. Associated types and values are essential in composing independently developed concepts. They help bridge the gap between different concepts. For example, in the theory of iterators (as briefly outlined in §5) the value type of an iterator plays a key rôle in the expression of `where`-clauses of function templates. For mathematical concepts like *group*, *ring* or *field*, the units of the respective structures are associated values. Uses of associated

types are presented in §5, where standard iterator concepts are discussed.

4. `fill` and associates revisited

Having outlined a framework for discussing concept definition and template checking; we now turn to the examples considered in §2.5 and §2.6.

4.1 The Assignable and Movable puzzle

The notion of iterator expressed as `Mutable_fwd` is a simplified version of the notion of *forward iterator* used by the C++ Standard Library. Unfortunately, it is oversimplified so that we can't use it for other algorithms where the C++ standard requires a forward iterator. Consider another (slightly simplified version) of the standard function `copy`:

```
template<typename Iter>
   where Mutable_fwd<Iter, Iter::value_type>
void copy(Iter first, Iter last, Iter out)
{
   for (Iter cur = first; cur != last; ++cur, ++out)
      *out = *cur;
}
```

The `value_type` is `Iter`'s associated type, the type of the element that the iterator refers to. However, `copy` fails to concept-check because we did not include the assumption that we could *read* from an iterator that meets the `Mutable_fwd` properties. We can compensate by adding a read operation:

```
concept Mutable_fwd<typename Iter, typename T> {
  Var<Iter> p;  // placeholder for variable of type Iter.
  Var<const T> v;
  Var<T> v2;
  // ... as before ...
  *p = v;      // we can write to *p
  v2 = *p;     // we can read from *p
};
```

With this version of `Mutable_fwd`, we slightly over-constrained `fill` (because `fill` never reads from elements of its sequence) but that is probably acceptable as the C++ Standard Library does the same. Unfortunately, we also over-constrained `copy` in a way that is unacceptable. Consider:

```
auto_ptr<Resource> v[10];
auto_ptr<Resource> w[10];
// ...
copy(v, v + 10, w);
```

An `auto_ptr` holds a pointer to a value and implements ownership semantics for that value. That is, instead of making a duplicate `auto_ptr`, assignment makes the target of the assignment the owner and invalidates the source. To do that invalidation, the assignment operation on `auto_ptr` writes to its source. Looking at `Mutable_fwd` (§2.6) we see that this won't work. `Mutable_fwd` requires only read access to the source of an assignment (`v` is `const`). That may be reasonable, but that it is not what the standard requires and not what is needed to cope with `auto_ptr`. We can try to fix that by requiring write access to the source of an assign:

```
concept Mutable_fwd<typename Iter, typename T> {
  Var<Iter> p;  // a variable of type Iter.
  Var<T> v;     // note, no ``const'' here.
  Var<T> v2;
  // ... as before ...
  *p = v;       // we can write plain ``T'' to *p
  v2 = *p;      // we can read from *p
};
```

Unfortunately, this is even worse: With that definition of `Mutable_fwd`, every type `T` that defines only an assignment taking a `const T&` now fail concept checking as an element accessed through a `Mutable_fwd` — and that is most types. Fundamentally, what we see here is that it is hard to precisely specify concepts so that we get both perfect separate checking of template arguments and the flexibility we are used to given the semi-formal specification of the standard iterators, containers and algorithms. Rather than patching, we will start by making the fundamental distinction between destructive assignments and non-destructive assignments and then build upon those.

4.1.1 Unary iterator predicates

To match the Standard Library requirements, we need a unary predicate to define a forward iterator. We do that by making the second template-parameter, the element type, implicit:

```
concept Forward_iterator<typename Iter> {
  Var<Iter> p;       // a variable of type Iter.
  typename Iter::value_type // must have a named member
                     // associated type value_type.

  Iter q = p;        // an Iter must be copy-able

  bool b = (p != q); // must support ``=='' and ``!=''
  b = (p == q);      // operations, and the resulting
                     // expressions must be convertible
                     // to ``bool''.

  ++p;               // must support pre- and
  p++;               // post-increment operations, no
                     // assumption on the result type
};
```

Here we have eliminated any requirements on the element type beyond the fact that it must exist and we can refer to it as `value_type`. That solves our problem deciding what kind of access we need to the `value_type` object by leaving that to the `where`-clause. In general, the use of *associated types*, such as `value_type` simplify the expression of generic programs [GJL+03].

4.1.2 Assignable

We can define what we mean for a value of type `U` to be *assignable* to an object of type `T`:

```
concept Assignable<class T, class U = T> {
  Var<T> a;
  Var<const U> b;
  a = b;         // non-destructive assignment
};
```

The assignment operator just reads its right hand side without modifying it — it can't modify because it takes a `const` operand. Usually, we also need the semantics invariant that, after assignment, the values `a` and `b` are equivalent (in some sense). That is what the C++ Standard Library requires. However, even though concepts could be designed to express semantics notions we haven't (yet) defined a syntax for expressing semantics for our concepts. The C++ standard makes the assumption that the type of an assignment to `T` is `T&`. We do not need that extra assumption, so we don't include it in `Assignable`.

To contrast and complement, we define destructive assignment (often called "a move") like this:

```
concept Movable<class T, class U = T> {
  Var<T> a;
```

```
  Var<U> b;
  a = b;       // potentially-destructive assignment
};
```

Given the concepts `Forward_iterator`, `Assignable` and `Movable`, we can declare the templates `fill` and `copy` as

```
template<Forward_iterator Iter, class T>
   where Assignable<Iter::value_type, T>
void fill(Iter first, Iter last, const T& t);

template<Forward_iterator Iter>
   where Assignable<Iter::value_type>
         || Movable<Iter::value_type>
Out copy(Iter first, Iter last, Iter out);
```

With these declarations, both the definitions of `fill` and `copy` will concept check. Their uses will succeed in all valid cases and a `fill` with an `auto_ptr` as its third argument will fail. In other words, we have a system that allows us to specify key C++ standard library components. Please note how the `where`-clauses in the function declarations complement the general concepts `Asslignable` and `Movable` by tieing the template-parameters together for the purpose of enforcing specific requirements of each function.

5. Standard iterator concepts

In this section, we define concepts for the most difficult part of the C++ Standard Library iterator hierarchy: input iterator, output iterator, and forward iterator. The C++ standard says: "Forward iterators satisfy all the requirements of the input and output iterators and can be used wherever either kind is specified". The first half of that statement is not actually true, but it is close enough for the second half to hold in all reasonable cases given experienced Standard Library implementers with common sense. However, a type system cannot work for reasonable cases only, and compilers (enforcing a type system) are not noted for their common sense.

As part of a complete solution, we need to formally define

1. how one moves and compares iterators (++, +, ==, etc.);
2. whether one can write to the iterator (`*p = x`);
3. whether one can read from an iterator (`x = *p`);
4. whether the assignment (or copy-initialization) used is destructive (the `auto_ptr` mess);
5. whether one can use a multi-pass algorithm (visiting an element twice; impossible for an input iterator or an output iterator.)

The C++ standard, and countless successful applications, reduce that to the simple programmers' rules of thumb:

1. one can read from an input iterator;
2. one can write to an output iterator;
3. a forward iterator is both an input iterator and an output iterator
4. the rest is detail that one can look up if needed.

Users who have written successful — and type safe — applications based on this (over) simplification of the current rules would not accept an "improved" system that required them to understand significantly different rules and to write more code, just to do the same work. In other words, just parameterizing an iterator concept by all sources of variability would not do the job. Users cannot be asked to explicitly select their iterators from a set of more than a dozen iterator categories. The ideal solution would be one where what the programmers thought was true (but isn't) is, and where all reasonable code compiles. This is almost possible, but only almost. In particular, we cannot avoid `where`-clauses to express requirements on combinations of template parameters.

We will not explain the problems of the current iterator requirements in detail. That would be tedious and pointless as many of the weaknesses are well understood in the C++ implementor and language lawyer community, are being corrected, and don't actually affect applications builders. They include:

1. The lack of distinction between destructive and non-destructive assignments.
2. A failure to consistently require iterators to be copy constructible.
3. A failure to point out that a forward iterator to a `const` value type isn't an output iterator.
4. Problems with composability of requirements: Specifying that `*p = v` must work for an output iterator `p` and that and `v = *q` must work for an input iterator `q`, but failing to note that this does not imply that `*p = *q` must work even when `p` and `q` have the same value types. (The reason is that both `*p = v` and `v = *q` may require a user-defined conversion so that `*p = *q` could require two user-defined conversions, and that's disallowed by the C++ rules for implicit conversion.)

Some of these problems were first discovered as part of our effort to find a practical and formal specification of the Standard Library facilities.

First we define a few supporting concepts. Some deal with basic access issues:

```
concept Copy_constructible<typename T> {
    Var<T> a;
    T b = a;  // copy construction
    T c(a);   // direct copy construction
};

concept Assignable<typename T, typename U = T> {
    Var<T> a;
    Var<const U> b;
    a = b;    // copy (non-destructive read)
};

concept Movable<typename T, typename U = T> {
    Var<T> a;
    Var<U> b;
    a = b;    // potentially-destructive read
};

concept Equality_comparable<typename T, typename U = T> {
    Var<T> a;
    Var<U> b;
    bool eq = (a == b);
    bool neq = (a != b);
};

concept Arrow<typename T> {
              // built-in
};
```

`Arrow<typename T>` is a built-in predicate expressing the curious Standard Library requirement for iterators that if `(*p).m` is legal then `p->m` is legal. Note that `Arrow<P>` is true for all C++ built-in pointer types and for all standard conforming user-defined ("smart") pointer types. It has been ob-

served that, for a given `m`, it may be possible possible to express the "arrow assumption" using `!` and (short-circuit) `||`:

```
concept Just_star_dot_m<typename T> {
  Var<T> p;
  (*p).m;
};
concept Just_arrow_m<typename T> {
  Var<T> p;
  p->m;
};
concept Arrow_m<tpename T>
   where !Just_star_dot_m<T> || Just_arrow_m<T>
{ };
```

However, we have no way of abstracting over `m`; that is the main reason why `Arrow` is a built-in predicate.

The notion of a `Trivial_iterator` specifies what is common for all iterators (not much):

```
concept Trivial_iterator<Copy_constructible Iter> {
  typename Iter::value_type;

  Var<Iter>  p;
  Iter& q = ++p;          // usable as
  const Iter& q2 = p++;   // converts to
};
```

Initializing a `const` reference means "converts to" and initializing a `non-const` reference means "usable as" (according to the C++ standard). Note that this use of references ensures that inheritance is taken into account. We don't feel an urgent need to invent new notation for that.

We can now define an input iterator as a trivial iterator that we can increment, compare, assign to, use `->` on, and read from:

```
concept Input_iterator<Trivial_iterator Iter>
           where Equality_comparable<Iter>
                 && Assignable<Iter>
                 && Arrow<Iter> {
  Integer difference_type; // the type of distance between
                           // two input iterators
  Var<Iter> p;
  const Iter::value_type& v = *p;    // converts to
  const Iter::value_type& v2 = *p++; // converts to
};
```

This is much more succinct than the "input iterator requirements" in the C++ standard [ISO03, §24.1.1], more precise, and also more correct. For example, the standard forgot part of the `Copy_constructible` requirement, but fortunately, none of the implementations did or this example (from the standard) wouldn't have compiled:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator out);
```

We discovered this when defining the concepts.

Note that the `Input_iterator` concept does not say what type is returned by `*`. In particular, it does not say that the result type of `*p` is `p`'s `value_type`; it could be a proxy type that implicitly converted to `value_type`. Eliminating the possibility of a proxy here would not only over-constrain the problem, it would also break real optimized code. One of the beauties of using use patterns compared to signatures is that we don't have to be explicit about possible intermediate types.

We do not assume that the result of dereferencing an input iterator is an lvalue; thus, we do not require (or allow) destructive reads from an input iterator. For example, `auto_ptr` is not an acceptable value type for an input iterator. If we want to read an `auto_ptr` from an input iterator, we need to say so in some `where`-clause.

The `difference_type` is the type used to express the number of elements between two iterators. It is defined using the concept `Integer` to require it to be a signed integer type.

For output iterator, we had to explicitly cope with the possibility of destructive assignment, so first we define concepts to express that:

```
concept Output_assign<Trivial_iterator Out, typename T> {
  Var<Out> p;
  Var<const T> v;
  *p = v;
  *p++ = v;
};

concept Output_move<Trivial_iterator Out, typename T> {
  Var<Out> p;
  Var<T> v;
  *p = v;
  *p++ = v;
};
```

Given those we can define `Output_iterator` as a `Trivial_iterator` that one can write to:

```
concept Output_iterator<Trivial_iterator Out>
          where Output_assign<Out,Out::value_type>
                || Output_move<Out,Out::value_type>
{ };
```

This does not in itself solve all problems with using output iterators. The reason is that our analysis shows that most problems with specifying iterators and the iterator hierarchy relates to specifying exactly when and how one can write to an output iterator. Furthermore, most of the troublesome variations and alternatives directly reflect algorithms and relationship between the value written and the iterator. Such issues are best dealt with in the algorithms' `where`-clauses. This definition of `Output_iterator` simply takes care of the minimal case where an algorithm simply assigns a value to an output iterator.

Note that an output iterator isn't `Equality_comparable` or `Assignable`. That's not our interpretation but a requirement that the Standard Library imposes for good reasons. Output iterators are an oddity, but a useful oddity that directly reflects the nature of output.

To help writing `where`-clauses for output iterators, we provide a helper concept reflecting the most common use involving another type, copying from another iterator:

```
concept Output_from_input<Output_iterator Out,
                          Input_iterator In> {
   Var<Out> p;
   Var<In> q;
   *p = *q;
   *p++ = *q++;
};
```

The C++ standard explicitly defines a forward iterator as something that meets all the requirements of an input iterator and an output iterator. In addition, it adds assumptions needed to support multi-pass algorithms:

```
concept Forward_iterator<Input_iterator Iter>
          where Output_iterator<Iter> {
  Iter p;                      // default constructible
  Iter::value_type& t = *p;    // usable as
  Iter::value_type& t2 = *p++; // usable as
};
```

So, how does this specification of the Standard Library requirements fare vis-à-vis our Assignable-and-Movable puz-

zle (§4.1)? For simple assignments (`*p = v`) an output iterator simply works for both ordinary and destructive assignments. The real (not simplified) `copy` copies from a sequence defined by a pair of input iterators to a sequence defined by an output iterator. To define that, we need to deal with the relationship between the value types of the two iterator types:

```
concept Move_from_input<Output_iterator Out,
                        Input_iterator In>
          where Output_from_input<Out, In> {
  Var<In> q;
  In::value_type& v = *q;
};

template<Input_iterator In, Output_iterator Out>
    where Output_from_input<Out, In>
          || Move_from_input<Out, In>
Out copy(In first, In last, Out out)
{
   for (In cur = first; cur != last; ++cur, ++out)
      *out = *cur;
}
```

The `Move_from_input` differs from `Output_from_input` only in requiring that the input iterator refer to a value that one could possible modify.

The implementation for `copy` remains the same as ever. The code for `copy` remains as good as ever. All we have done is to get perfect separate checking and some new opportunities for overloading.

We value the iterator requirements as a pretty extreme real-world challenge to any system aimed at specifying requirements for template arguments.

6. Related work

6.1 Siek's proposal

Jeremy Siek *et al.* [SGG+05] proposed a somewhat different concept system for C++. After discussions in the C++ standards committee, that system was modified [GSW+05] to handle many key issues discussed in the "use pattern based concepts" proposal [SDR05a]. In that system, a concept definition itself consists of sequence of operations with so-called pseudo-signatures. When concept checking template definitions, the pseudo-signatures act like the exact type of the operations. When concept checking a template use, wrapper functions are implicitly generated to implement conversions between the exact type of the declarations as found in the use context and the signature of the operations as assumed by the concept definition. That constitutes a huge departure from C++ semantics, which implies that parts of a program can be led to believe that some functions exist when, in fact, they do not. It is uncertain whether such forwarding functions can be consistently eliminated to allow optimal and consistent use of overloaded versions of a function. There are other differences between this system and ours, such as an absence of the *or* and *not* operators for concepts.

A similar, but slightly more abstract variant of the pseudo-signature idea called "abstract signatures", was presented and analyzed in [Str03] and [SDR05a]. We rejected that in favor of the use pattern notation because it was too verbose and because it would introduce a whole new declaration syntax with associated special semantics into an already crowded syntactic universe.

There is a close relationship between concepts, as described here, and constraints classes [Str]. This allows us to test concepts by transcribing them into constraints classes and explicitly insert them into code.

6.2 Type classes

It has been claimed that concepts as envisioned for C++0x are just Haskell's type classes. In fact, "overloading" in Haskell is limited compared to C++'s notion of overloading. In C++, overloading is exclusively a compile-time notion: Two functions are said overloaded if they have different signatures; there is no requirement of generic instance relationship between them. For example, C++ allows programs like

```
int min(int, int);                  // #1
template<typename T> T min(T, T);   // #2
int min(int, int, int);             // #3
// ...
int x = min(1, 2);                  // call #1
int y = min('a', 'b');              // call #2
int z = min(1, 2, 3);               // call #3
```

An overload set can contain function templates as well as "ordinary" functions. Haskell overloading is expressible in C++ as a combination of overriding and template specialization. Note that in C++, selection among overridden virtual functions takes place at run-time whereas overloading takes place at compile time. When all Haskell instance declarations for a given type class are available in a module, the type class construct and instance declarations correspond closely to C++'s notion of template partial or full specialization — this morally is what happens when the dictionaries, in the dictionary passing style translation [WB89], are optimized away. Given this close correspondence between Haskell type classes and C++ template specialization, it would not be surprising at all that a large body of C++ *type traits* techniques can be translated to Haskell type class techniques, and vice versa. Indeed, there are recent proposals [CKPJM05, CKPJ05] to add associated types to Haskell.

When instance declarations for a given type class are spread over several modules, information about the class members usually cross modules through indirect function calls (as do C++ virtual function calls). In that case, type classes closely correspond to parameterized abstract classes and virtual function overriding. A more detailed account is provided in the "use pattern based concept" proposal [SDR05a, Appendix B]. The problems addressed by type classes are more limited than those addressed by concepts.

The contexts of Haskell instance declarations are additional features absent from current C++. These are among the simplest contructs expressible with concepts where assumptions can be expressed individually on template-parameters in isolation.

As general predicates, concepts can be combined with the logical operators (*and*, *or*, *not*). An obvious place where the ability to write such logical formula is useful is in the `where`-clauses of template declarations as extensively illustrated.

6.3 Qualified types

In his PhD thesis [Jon94], Mark Jones introduced the notion of *qualified types* as a general framework to approach constrained type systems as studied by Stefan Kaes [Kae88], and Philip Wadler and Stephen Blott [WB89] that form the basis of Haskell's type classes. Jones' framework is general enough to account for Haskell's type classes, sub-typing and extensible records. It was later generalized to constructor classes and type classes with functional dependencies. However, Jones' system strives at describing systems where constraints are expressed purely at the type level. As we have seen, that is not accurate enough for C++ templates. Furthermore, Jones' framework seems to be more appropriate for type systems with *overriding* or *specialization* semantics than

with general overloading and type scheme as found in C++. Finally, while Jones' qualified types are formally type with predicates, the predicates cannot be directly used in formula involving logical connectors as in the concept system presented in this paper; and use of compile-time integer values is not included in the theory of qualified types.

7. Conclusion and future work

In this paper, we have defined a framework for specifying a concept system for checking C++ templates. This system, unlike conventional signature-based or object-oriented style type system, is powerful enough to express simply, concisely and accurately the C++ Standard Library notions and requirements. This formulation of concepts enables perfect checking of template definitions and uses in isolation without adverse effects on the performance of generated code. In the process, we uncovered several weaknesses in the current informal formulation of the Standard Library requirements. We have a complete typed abstract syntax tree representation for C++, including concepts, that will become a testbed for further work [SDR05b]. Future directions for work include:

- Complete our formalism for ISO C++. It is capable of handling proposed extensions that affect its type system including concepts.
- Explore various assumptions [SDR05a] on the overload resolution operator to provide a complete proof of the "fundamental theorem":

  Theorem 1 (Soundness) *If a template definition concept checks and if its uses both concept check and type check then its instantiations for those uses also type check.*

  We have a draft of that proof, but it is too long to fit in the margin here. That might require restrictions on template partial specializations as discussed in [SDR05a].
- Complete a concept checker based on abstract syntax trees.
- Experiment with the use of concepts to specify the STL and other domains, such as the theory of mathematical structures in Computer Algebra [JS92].
- The work reported in this paper focuses on the static semantics of concepts, but concepts also have dynamic semantics components that will be subject of future work.

Together with our colleagues in the ISO C++ standards committee, we will analyze the various ideas for concepts and synthesize a concept system for C++0x.

Acknowledgments

We would like to thank the anonymous reviewers. We are grateful to Phil Wadler for his helpful comments and suggestions for improving on earlier versions of this paper. Feedbacks from Sophia Drossoupoulou were also useful in improving on the introductory material to C++ templates. We appreciate Jaakko Järvi's comments on an early draft.

References

[Aug93] Lennart Augustsson. Implementing Haskell Overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.

[Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, October 1998.

[CKPJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated Type Synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, 2005. ACM Press.

[CKPJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. *ACM SIGPLAN Notices*, 40(1):1–13, January 2005.

[DR02] Gabriel Dos Reis. Generic Programming in C++: The next level. *The Association of C and C++ Users Spring Conference*, April 2002.

[DRS05a] Gabriel Dos Reis and Bjarne Stroustrup. A Formalism for C++. Technical Report N1885=05-0145, ISO/IEC SC22/JTC1/WG21, October 2005.

[DRS05b] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. Technical Report N1886=05-0146, ISO/IEC SC22/JTC1/WG21, October 2005.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Université Paris VII, 1972.

[GJL+03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 115–134. ACM Press, 2003.

[GSW+05] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concept for C++0x (Revision 1). Technical Report N1849=05-00109, ISO/IEC SC22/JTC1/WG21, August 2005.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[ISO03] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[Jon94] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[JS92] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, 1992.

[Kae88] Stefan Kaes. Parametric Overloading in Polymorphic Programming Languages. In *Proceedings of the 2nd European Symposium on Programming*, volume 300 of *Lecture Notes In Computer Science;*, pages 131–144. Springer-Verlag, 1988.

[PJ03] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[PT98] Benjamin C. Pierce and David N. Turner. Local Type Inference. In *Symposium on Principles of Programming Languages*, pages 252–265, San Diego CA, USA, 1998. ACM.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proceedings of Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, 1974. Springer-Verlag.

[SDR03a] Bjarne Stroustrup and Gabriel Dos Reis. Concepts — Design choices for template argument checking. Technical Report N1522, ISO/IEC SC22/JTC1/WG21, September 2003.

[SDR03b] Bjarne Stroustrup and Gabriel Dos Reis. Concepts — syntax and composition. Technical Report N1536, ISO/IEC SC22/JTC1/WG21, September 2003.

[SDR05a] Bjarne Stroustrup and Gabriel Dos Reis. A Concept Design (rev.1). Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21, April 2005.

[SDR05b] Bjarne Stroustrup and Gabriel Dos Reis. Supporting SELL for High-Performance Computing. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[SGG+05] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concept for C++0x. Technical Report N1758=05-0018, ISO/IEC SC22/JTC1/WG21, January 2005.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21, May 1994.

[Str] Bjarne Stroustrup. Technical FAQ: Why can't I define constraints for my template parameters? `http://www.research.att.com/~bs/bs_faq2.html#constraints`.

[Str88] Bjarne Stroustrup. Parameterized Types for C++. In *Proceedings of USENIX C++ Conference*, Denver, CO., October 1988.

[Str89] Bjarne Stroustrup. The Evolution of C++: 1985–1989. *USENIX Computer Systems*, 2(3), 1989.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[Str03] Bjarne Stroustrup. Concept checking — A more abstract complement to type checking. Technical Report N1510, ISO/IEC SC22/JTC1/WG21, September 2003.

[Str04] Bjarne Stroustrup. Abstraction and the C++ model. In *Proceedings of ICESS'04*, December 2004.

[Str05] Bjarne Stroustrup. The design of C++0x. *C/C++ Users Journal*, May 2005.

[TDBP00] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, and Erhard Ploederer, editors. *Consolidated Ada Reference Manual*, volume 2219 of *Lecture Notes in Computer Science*. Springer, 2000.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, Austin, Texas, USA, 1989.

C and C++: Siblings

Bjarne Stroustrup

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

This article presents a view of the relationship between K&R C's most prominent descendants: ISO C and ISO C++. It briefly discusses some implications of these incompatibilities, reflects on the ''Spirit of C'' notion, and gives examples of how incompatibilities can be handled.

This article is the first of three, providing a ''philosophical'' view of the C/C++ relationship. The second article will present arguments to the effect that a merger of C and C++ is the best direction for the C/C++ community [Stroustrup,2002b], and the third article will present some examples of how language incompatibilities might be reconciled [Stroustrup,2002c].

1 Introduction

Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. The resulting incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of tools for C and C++.

My focus here is the areas where C and C++ differ slightly (''the incompatibilities''), rather than on the large area of commonality or the areas where one language provide facilities not offered by the other. A longer technical report which presents more historical context and many more examples is available online [Stroustrup,2002].

2 A Family Tree

How can I call C and C++ siblings? C++ is a descendant of K&R C. However, what we call C today (the C89 or C99 standard [C89] [C99]) is also a descendent of K&R C:

A solid line means a massive inheritance of features, a dashed line borrowing of major features, a dotted line borrowing of minor features.

From this, ISO C and ISO C++ emerge as the two major descendants of K&R C, and as siblings. Each carries with it the key aspects of Classic C, and neither is 100% compatible with Classic C. For example, both siblings consider ***const*** a keyword and both deem this famous Classic C program non-standard-compliant:

```
main()
{
    printf("Hello, world\n");
}
```

As a C89 program, this has one error. As a C++98 program, it has two errors. As a C99 program, it has the same two errors, and if those were fixed, the meaning would be subtly different from the identical C++ program.

To simplify, I have left influences that appeared almost simultaneously in both languages during standardization unrepresented on the chart. Examples of that are ***void****** for C++ and C89, and the ban of ‘‘implicit ***int***’’ in C++ and C99.

Classic C is K&R C [Kernighan,1978] plus structure assignment, enumerations, and ***void***. I picked the term ‘‘Classic C’’ from a sticker that used to be affixed to Dennis Ritchie’s terminal.

Incompatibilities are nasty for programmers in part because they create a combinatorial explosion of alternatives. Leaving out Classic C for simplicity, consider a simple Venn diagram:

There are features belonging to each of the seven areas:

C89 only	call of undeclared function
C99 only	variable length arrays (VLAs)
C++ only	templates
C89 and C99	Algol-style function definitions
C89 and C++	use of the C99 keyword ***restrict*** as an identifier
C++ and C99	*//* comments
C89, C++, and C99	***struct***s

For each language feature, a programmer must remember to which language the feature belongs and what its meaning is. That is a cause of confusion and bugs. For each feature, an implementor must allow it for the appropriate language only. This becomes much worse when various proprietary extensions and compiler switches are taken into account.

One of the big questions for the C/C++ community is whether the next phase of standardization (potentially adding two more circles to the diagram) will pull the languages together or tear them further apart. In ten years, there will be large and thriving C and C++ communities. However, if the languages are allowed to drift further apart, there will not be a C/C++ community, sharing tools, implementations, techniques, headers, code, etc. My nightmare scenario looks a bit like this:

Each separate area of the diagram represents a set of incompatibilities that an implementer must address and that a programmer may have to be aware of.

The differences between C++ and C89 are documented in Appendix C of the ISO C++ standard [C++98]. The major differences between C89 and C99 are listed on two pages of the foreword of the C99 standard [C99]. The differences between C++ and C99 are not officially documented because the ISO C committee had neither the time nor the expertise to do so, and documenting C++/C99 incompatibilities was not required by the C99 committee's charter [Benito,1998]. An unofficial, but extensive list of incompatibilities can be found on the web [Tribble,2001]. See also Appendix B of [Stroustrup,2000].

3 The Spirit of C

The phrase ''The spirit of C'' is often used as a weapon to condemn notions supposedly not in the right spirit and therefore somehow illegitimate. So is the complementary phrase ''The spirit of C++.'' More reasonably, these phrases can be used to distinguish languages aimed at supporting low-level systems programming, such as C and C++, from languages without such support. However, I find these ''spirit arguments'' poisonous when thoughtlessly applied within the C/C++ community. More often than not, these phrases dress up personal likes and dislikes as philosophies supposedly backed by ''the fathers of C'' or ''the fathers of C++.'' This can be amusing and occasionally embarrassing to Dennis Ritchie and me. We are still alive and do hold opinions, though Dennis – being the older and wiser – is better able to keep quiet.

Here are a few ''rules'' often claimed to be or be part of ''the spirit of C:''

[1] keep the built-in operations close to the machine (and efficient)
[2] keep the built-in data types close to the machine (and efficient)
[3] no built-in operations on composite objects
[4] don't do in the language what can be done in a library

[5] the standard library can be written in the language itself
[6] trust the programmer
[7] the compiler is simple
[8] the run-time support is very simple
[9] in principle, the language is type-safe, but not automatically checked (use lint for checking)
[10] the language isn't perfect because practical concerns are taken seriously

All can be supported by quotes from the opening pages of K&R-1 [Kernighan,1978].

Naturally, Classic C is a good approximation to ''the spirit of C.'' C99 and C++ are less so, but they still approximate those ideals. This is significant because most languages don't. From the perspective of Ada, Java, or Python, C and C++ appear as twins. Only in discussions within the C/C++ community do the differences appear to overwhelm the commonalities.

In the spirit of [10], Classic C breaks [3] by adding structure assignment and structure argument passing to K&R C.

C++ starts out by breaking [7]: A greater emphasis on type and scope distinguishes C++ compared to C. Consequently, a C++ compiler front-end must do much more than Classic C front-end does. The introduction of exceptions complicates C++'s run-time support, violating [8]. However, that may be defended on the grounds that if you don't need exceptions, you can avoid using them. After 20 years, it is more remarkable that C++ closely follows the remaining eight criteria. In particular, C++ can be seen as the result of following [1] to [5] to their logical conclusion by allowing the user to define general and efficient types and libraries.

Compared to early C compilers, modern C implementations cannot be called simple, so C99 also breaks [7]. Since ***<tgmath.h>*** cannot be written in C (though something almost identical could be written in C++), C99 breaks [5]. Arguably, C99's ***complex*** facilities violate [1], [2], and [3].

Contrary to popular myths, there is no more tolerance of time and space overheads in C++ than there is in C. The emphasis on run-time performance varies more between different communities using the languages than between the languages themselves. In other words, overheads are found in some uses of the languages rather than in the language features.

Why is ''the spirit of C'' of interest? It is worth calmly discussing ''the spirit of C'' because this topic has been used to inflame language wars and especially because what underlies those flame wars is often a genuine concern for the direction of evolution of C and/or C++. That is, a consistent aim/philosophy is needed for a coherent language to emerge from a set of changes and extensions.

In their evolution from Classic C, C99 and C++ differ in philosophy. C++ has a clearly stated philosophy of language: the emphasis in the selection of new facilities is on mechanisms for defining and using new types safely and efficiently. Basic facilities for computation were inherited, as far as possible unchanged, from Classic C and later from C89. C++ will go a long way to avoid introducing a new fundamental type. The prevailing view is that if you need one type then many programmers will need similar types. Consequently, providing mechanisms for expressing such types in the language will serve many more programmers than would providing the one type as a built-in. In other words, the emphasis is on facilities for organizing code and building libraries (often referred to as ''abstraction mechanisms'').

To contrast, the emphasis in the evolution of C89 into C99 has been on the direct support for traditional (Fortran-style) numerical computation. Consequently, the major extensions of C99 compared to C89 are in new built-in numeric types, new mathematical functions and macros, numeric I/O facilities, and extensions to the notion of an array. The contrasting approaches to complex numbers and to ***vector***s/VLAs illustrate the difference in C++'s and C99's design philosophies: C adds built-in facilities where C++ add to the standard library [Stroustrup,2002].

Ideally, C's emphasis on built-in facilities and C++'s emphasis on abstraction mechanisms are complementary. However, for that to work smoothly, the emphasis on built-in facilities must be on fundamental computational issues (that is, on facilities that cannot elegantly and efficiently be provided by composing already existing facilities) and care must be taken not to increase reliance on mechanisms known to cause problems for the abstraction mechanisms (such as macros, uneven support for built-in types, and type violations).

3.1 Macros

Typical C and C++ programmers view macros very differently. The difference is so great that it can be considered philosophical. C++ programmers typically avoid macros wherever possible, preferring facilities that obey type and scope rules. In most cases, C programmers don't have such alternatives and use macros. For example, a C++ programmer might write something like this:

```
const int mx = 7;

template<class T> inline T abs(T a) { return (a<0)?-a:a; }

namespace N {
    void f(int i) { /* ... */ }
};

class X {
public:
    X(int);
    ~X();
    // ...
};
```

A C programmer facing a similar task might write something like this:

```
#define MX 7

#define ABS(a) (((a)<0)?-(a):(a))

void N_f(int i) { /* ... */ }

struct X { /* ... */ };
void init_X(struct X *p, int i);
void cleanup_X(struct X *p);
```

At the core of many C++ programmers' distrust of macros lies the fact that macros transform the program text before tools such as compilers see it. Because macro substitution follows rules that don't involve scope or semantics, surprises can result. Namespaces, class scopes, and function scopes provide no protection against a macro. Eliminating the use of macros to express ideas in code has been a constant aim of C++ (see Chapter 18 of [Stroustrup,1994]). This implies that a C++ programmer tends to view a solution involving a macro with suspicion, and at best as a lesser evil. On the other hand, a C programmer often view that same solution as natural, and often as the most elegant. Both can be right – in their respective languages – and this is a source of some misunderstanding. Any solution to a compatibility problems that involves a macro is automatically considered suspect by many C++ programmers. Thus, any use of a macro in the standard becomes a potential incompatibility as the C++ community looks for alternative solutions to avoid its use. The only macro found in the C++ standard beyond those inherited from C is *__cplusplus*.

4 Impact of C/C++ Feature Differences

C++ provides many features not found in C, such as virtual functions and declarations in conditions. Similarly, C99 provides several features not found in C++, such as variable length arrays (VLAs) and designated initializers. When considering compatibility issues, C/C++ features can be classified based on what they affect:

- those that affect interfaces, such as virtual functions and VLAs,
- those that affect only the form of the code that they are part of, such as declarations in conditions and designated initializers.

The following sections give examples of compatibility issues and explore the problems programmers face when dealing with C and C++.

4.1 Trivial Interfaces

C++ programmers have always known that to make code accessible to C programs they must provide interfaces that avoid non-C features, such as classes with virtual functions. These C to C++ interfaces have typically been trivial. For example:

```
// C interface:
extern int f(struct X* p, int i);

// C++ implementation of C interface:
extern "C" int f(X* p, int i) { return p->f(i); }
```

C programmers have typically assumed that any C header can be used from a C++ program. This has largely been true (after someone adds suitable ‘‘*extern "C"*’’ directives), though headers that use C++ keywords as identifiers have been a constant irritant to C++ programmers (and sometimes a serious practical problem). For example:

```
class X { /* ... */ };                  // not C
struct S { int class; /* ... */ };      // not C++
```

C99 introduces several features that if used in a header will prevent that header from being used in a C++ program (or in a C89 program). Examples include VLAs, ***restrict***ed pointers, ***_Bool***, ***_Complex***, some ***inline*** functions, and macros with variable number of arguments For example:

```
// C99 interface features, not found in C++ or C89:

void f1(int[const]);                    // equivalent to f(int *const);
void f2(char p[static 8]);              // p is supposed to point to at least 8 chars
void f3(double *restrict);
void f4(char p[*]);                     // p is a VLA

inline void f5(int i) { /* ... */ }     // may or may not be C++ also [Stroustrup,2002]

void f6(_Bool);
void f7(_Complex);

#define PRINT(form ...) fprintf(form, __VA_ARGS__)
```

If a C header uses one of those features, mediation code and a C++ header must be provided for the C code to be used from C++.

The ability to share header files is an important aspect of C and C++ culture and a key to performance of programs using both languages: C and C++ programs can call libraries implemented in ‘‘the other language’’ with no data conversion overheads and no (or very minimal) call overhead.

4.2 Thin Bindings

Where language features differ so that very similar functionality is provided in different ways, approaches based on sharing declarations are insufficient to mask the language differences. One approach to dealing with this is to provide ‘‘compatibility headers’’ that, through liberal use of ***#ifdef***s, provide very different definitions for each language but allow user code to look very similar. For example:

```
// my double precision complex

#ifdef __cplusplus
    #include<complex>
    using namespace std;
    typedef complex<double> Cmplx;
    inline Cmplx Cmplx_ctor(double r, double i) { return Cmplx(r,i); }
    //...
#else
    #include<complex.h>
    typedef double complex Cmplx;
    #define Cmplx_ctor(r,i) ((double)(r)+I*(double)(i))
    //...
#endif
```

```
void f(Cmplx z)
{
    Cmplx zz = z+Cmplx_ctor(1,2);
    Cmplx z2 = sin(zz);
    // ...
}
```

Basically, this approach is for the individual programmer or organization to create a new dialect that maps into both languages. This is an example of how a user (or a library vendor) must invent a private language simply to compensate for compatibility problems. The resulting code is typically neither good C nor good C++. In particular, by using this technique the C++ programmer is restricted to use what is easily represented in C. For examples, unless exceptional effort is expended on the C mapping, arrays must be used rather than containers, overloading beyond what is offered by C99's ***<tgmath.h>*** must be avoided, and errors cannot be reported using exceptions. In addition, macros tend to be used much more heavily than a C++ programmers would like. Such restrictions can be acceptable when providing interfaces to other code, but are typically too constraining for a C++ programmer to use them within the implementation. Similarly, a C programmer using this technique is prevented from using C facilities not also supported by C++, such as VLAs and ***restrict***ed pointers.

Real code/libraries will have much larger ''thin bindings'' with many more macros, typedefs, inlines, etc., and more conventions for their use. The likelihood that two such ''thin bindings'' can be used in combination is slim and the effort to learn a new binding is non-trivial. Thus, this approach doesn't scale and fractures the community.

4.3 Competing Programming Models

Interfaces – that is, information in header files – are all that matter to people who see C and C++ as distinct languages that just happen to be able to produce code that can be linked together (like C and Fortran). However, programmers who use both languages, teachers, and implementers must contend with equally intractable compatibilities issues related to the facilities used to express computations.

For users of both languages, the areas where C and C++ provide alternative solutions to similar programming problems become a problem:

[1] An alternative forces programmers to choose between two sets of facilities and their associated programming techniques.

[2] An alternative more than doubles the effort for teachers and students.

[3] Code using separate alternatives can often cooperate only through specially written mediation code.

Consider the problem of manipulating a number of objects where that number is known only at run time. C++ and C99 offer alternative solutions not present in C89. Consider a C89 example:

```
void f89(int n, int m, struct Y* v)  /* C89: v points to m Ys */
{
    struct X* p = malloc(n*sizeof(struct X)); /* not Classic C; not C++ */
    struct Y* q = malloc(m*sizeof(struct Y));
    if (p==NULL || q==NULL) exit(-1);       /* memory exhausted */
    if (3<n && 4<m) p[3] = v[4];
    memcpy(q,v,v+m*sizeof(struct Y));       /* copy */
    /* ... */
    free(q);
    free(p);
}
```

Among the potential problems with this code is that ***v*** might not point to an array with at least ***m*** elements.

The obvious C99 alternative is:

```
void f99(int n, int m, struct Y v[m])    // C99: v points to m Ys
{
    struct X p[n];                          // not C89; not C++
    struct Y q[m];
    if (3<n && 4<m) p[3] = v[4];
    memcpy(q,v,v+m*sizeof(struct Y));       // copy
    // ...
}
```

The nicer syntax makes it less likely that ***v*** does not point to an array with at least ***m*** elements, but that is still possible. Unfortunately, it is undefined what happens if the array definition fails to allocate memory for the ***n*** elements required. The use of arrays automates the freeing of memory, though there could still be a memory leak if ***f99*( )** is exited through a ***longjmp*( )**.

The obvious C++ alternative is:

```
void fpp(int n, vector<Y>& v)  // C++: v holds v.size() Ys
{
    vector<X> p(n);            // not C89; not C99
    if (3<p.size() && 4<v.size()) p[3] = v[4];
    vector<Y> q = v;           // copy
    // ...
}
```

A ***vector*** contains the number of its elements, so the programmer doesn't have to worry about keeping track of array sizes or about freeing the memory used to hold those elements.

The standard library ***vector*** is more general than a VLA. For example, ***vector*** has a copy operation, you can change the size of a ***vector***, and ***vector*** operations are exception safe (see Appendix E of [Stroustrup,2000]). This could imply a performance overhead compared to VLAs on some implementations, but so far I have not found significant overheads.

The key point here is that users have to choose and the users of more than one of these languages have to understand the different programming styles and remember where to apply them. The result is that these differences in the facilities of C and C++ make it significantly more difficult to program in both languages than to program in just one – even though the two languages share a common root.

5 As close as possible ...

The semi-official policy for C++ in regards to C compatibility has always been ''As Close as Possible to C, but no Closer'' [Koenig,1989]. Naturally, wits have answered with ''As Close as Possible to C++, but no Closer,'' but I have never seen that in any official context nor seen any elaboration of what it means.

How close is ''as close as possible to C?'' Traditionally, it has almost been possible to equate that statement with ''compatible with C except where the C++ type system would be compromised.'' Differences such as those for ***void****, C++'s insistence on function prototypes, the use of built-in types for ***bool*** and ***wchar_t***, and even the ***inline*** rules, can be explained that way [Stroustrup,2002].

The ''as close as possible ...'' rules were crafted under the assumption that ''the other language'' was immutable. In reality, it has not been so: Just look as the number of cross borrowings between C and C++ [Stroustrup,2002]. I believe that it would be technically feasible for ''as close as possible'' to be ''identical in the subset supporting traditional C-style programming'' assuming that changes could be made simultaneously to both languages systematically bringing them closer together.

Whatever is (or isn't) done must be considered in light of the fact that the world changes rapidly and that users expect programming languages to evolve to meet new challenges. Thus, compatibility issues must be considered in the wider context of language evolution. I think the most promising approach is to consider C and C++ close to complete in language support for their respective kinds of programming. If that is so, increasing C/C++ compatibility could be seen as part of a consolidation and cleanup of basic facilities. Most ''extensions'' belong in standard and non-standard libraries.

6 References

[Benito,1998] John Benito, the ISO C committee liaison to the ISO C++ committee in response to a request to document C++/C99 incompatibilities in the way C89/C++ incompatibilities are.

[Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Björn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.

[C89] ISO/IEC 9899:1990, Programming Languages – C.

[C99] ISO/IEIC 9899:1999, Programming Languages – C.

[C++98] ISO/IEC 14882, Standard for the C++ Language.

[Kernighan,1978] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ. 1978. ISBN 0-13-110163-3.

[Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-110362-8.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. July 1989.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – the language and its compiler*. Cambridge University Press, Cambridge, England. 1980. ISBN 0-521-21965-5.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[Stroustrup,2000] Bjarne Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] Bjarne Stroustrup: *Sibling Rivalry: C and C++*. AT&T Labs - Research Technical Report TD-54MQZY, January 2002. http://www.research.att.com/~bs/sibling_rivalry.pdf.

[Stroustrup,2002b] Bjarne Stroustrup: *C and C++: A Case for Compatibility*. The C/C++ Users Journal.

[Stroustrup,2002c] Bjarne Stroustrup: *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal.

[Tribble,2001] David R. Tribble: Incompatibilities between ISO C and ISO C++. http://david.tribble.com/text/cdiffs.htm.

C++ Programming Styles and Libraries

Bjarne Stroustrup

AT&T Labs - Research
Florham Park - NJ

ABSTRACT

One of the main aims of C++ has been to make it an excellent tool for writing libraries. Here, I present some points about the role of libraries and of the programming styles that they support and rely on. For lack of space for a thorough treatment of these themes, I refer to books.

1 Introduction

C++ is a multiparadigm programming language. That is, C++ supports several styles of programming:

C-style programming	C++ is a better C, maintaining C's flexibility and run-time efficiency while improving type checking;
Data abstraction	the ability to create types that suit our needs;
Object-oriented programming	the idea of programming with class hierarchies and runtime polymorphism; and
Generic programming	programming using type parameterization of both data types and algorithms.

Importantly, C++ supports the use of combinations of those styles. This is crucial because the most effective programming techniques involve a variety of styles that people often classify as different. Radically different programming styles are often referred to as different paradigms; hence the word ''multiparadigm'' [Stroustrup,2001].

Naturally, this flexibility is viewed as mere complexity by people who think that there is one style of programming that is right for everyone. However, C++ is a general-purpose programming language with a bias towards systems programming and none of the candidates for ''the one right way'' of writing programs adequately supports the range of needs faced by C++ programmers.

One group of programmers in particular need generality, flexibility, and efficiency beyond what most programmers consider normal and reasonable: library writers. Many parts of C++ are best understood as facilities supporting library writers and library users. Think of it this way: Without a good library, most interesting tasks are hard to do in C++; but given a good library, almost any task can be made easy. Here ''easy'' means that the programming language isn't a source of significant complexity so that a programmer can concentrate on the fundamental problems of the task in hand.

2 An example

Here is an example illustrating the power of libraries:

Copyright InformIt. 2002

```
void Decompose(double delt, SymTensor& V, Tensor& R, const Tensor& L)
{
    Symtensor D = Sym(L);
    AntiTensor W = Anti(L);
    Vector z = Dual(V*D);
    Vector omega = Dual(W) - 2.0*Inverse(V-Tr(V)*One)*z;
    AntiTensor Omega = 0.5*Dual(omega);
    R = Inverse(One-0.5*delt*Omega) * (One+0.5*delt*Omega)*R;
    V += delt*Sym(L*V-V*Omega);
}
```

According to [Budge,1992], ‘‘This code is transparent and ... a physicist familiar with the polar decomposition algorithm can make immediate sense of this code fragment without the need for additional documentation.’’

I’m not a physicist, so that code is not obvious to me, but it is clear how that code is trivially translated from what you find in a physics textbook. This example is taken from a pioneering library from Sandia Labs. However, such libraries are now common in various branches of high-energy physics and general vector and matrix libraries supporting them are widely used and available.

I chose physics as my example partly because it is a field where performance is essential. If a program in this field is significantly slower than a conventional Fortran program, it will be discarded. In this area, elegance of programs traditionally takes the back seat compared to performance. The reason is simple: this is a field where results are often limited by available computing resources. C++ and modern programming techniques have met that challenge. Using generic programming techniques, C++ vector and matrix libraries have equaled and exceeded the performance of classical Fortran code (for example, POOMA [POOMA] MTL [MTL] and ROOT [ROOT]). One key to this exceptional performance is the elimination of temporaries through the use of small function objects [Stroustrup,2000§22.4.7]. Another key is (other) function objects representing critical implementation policies. Basically, the average programmer cannot write code that equals the modern libraries in numerical performance. Good programmers often know better than to even try.

Different fields have different requirements. This leads to the use of different programming styles being key to the design and use of different libraries. Numeric libraries tend to rely on generic programming emphasizing conventional notation and efficiency. To contrast, libraries focusing on graphics or user interaction tend to rely heavily on object-oriented techniques involving class hierarchies. When number crunching meets data management and graphics, combinations of programming techniques occur naturally.

The time taken to write a program is at best roughly proportional to the number of lines written, and so is the number of errors in that code. If follows that a good way of writing correct programs is to write short programs. In other words, we need good libraries to allow us to write correct code that performs well. This in turn means that we need libraries to get our programs finished in a reasonable time. In many fields, such C++ libraries exist. The standard library is an excellent way to start getting away from the messier and more error-prone levels of programming [Stroustrup,2000] [Stroustrup,1999]. However, there is much more to C++ than the standard library – that library is just an essential beginning.

3 Education and Libraries

The best an individual or an organization can do to improve their ability to produce good code (that is clean, maintainable, efficient code) is to master modern C++ programming styles and use good libraries. The best best an individual or an organization can do for the general community is to contribute a good, well-documented library for a field where adequate libraries don’t yet exist.

In this context, I’d like to mention some books. These are books on programming style and libraries that I have had a small hand in as an editor. In several cases, I have encouraged the writing of these books because I thought their topic important. In every case, I have nagged the author to try to make the book shorter and more accessible to the reader.

‘‘Accelerated C++’’ [Koenig,2000] is an innovative textbook that teaches Standard C++ from scratch to beginners assumed to have programmed before in some other language. Its presents every feature in a context where the feature is realistically used effectively, relies on the standard library to teach effective programming styles, and doesn’t introduce basic language facilities (such as pointers and arrays) until they are

Copyright InformIt. 2002

needed. It is also a most useful book for C++ programmers who wants to move on to more modern styles of C++ programming. For example, ‘‘Accelerated C++’’ introduces templates several chapters before pointers.

‘‘Exceptional C++’’ [Sutter,2000] explores the uses of exceptions in quite some depth. Similarly, ‘‘Modern C++ Design’’ [Alexandrescu,2001] assumes that a reader understands the basis of templates and generic programming proceeds to demonstrate several advanced and interesting techniques. These are books for people who are not satisfied with introductory-level explanations or simple descriptions of language facilities. They expand into advanced programming techniques and into technical details that we – most of the time – prefer to remain hidden inside libraries. For most C++ programmers, these books combine the intellectual thrill of discovering something new with practical techniques for building and using libraries.

‘‘Modern C++ Design’’serves as a bridge to the second type of books that I want to mention: The documentation of the design and facilities of interesting C++ libraries. ‘‘Modern C++ Design’’ presents Alexandrescu’s ‘‘Loki’’ library for building generic components, many based on classical patterns.

‘‘C++ Network Programming’’ [Schmidt,2002] describes ACE, a very large and very successful library for programming distributed systems. Its roots go back to the early 1990s, and it is in serious industrial use. Much of what people think of doing with threads, sockets, locks, etc., is already being done using ACE. My favorite ACE story happened when a developer was explaining a large industrial system for medical applications at a conference. He emphasized his needs for portability and someone asked the inevitable question: ‘‘so why didn’t you use Java?’’ I had expected one of the usual answers (stressing performance, flexibility, or elegance), but he simply responded: ‘‘because ACE runs on more platforms than Java’’ and carried on with his talk. The ACE style combines classical object-oriented programming with extensive use of templates for type safety and flexibility.

The final book that I’ll mention here, ‘‘The Boost Graph Library’’ [Siek,2002] describes a new library. Its authors contributed the MTL (Matrix Template Library [MTL]) to the numerical computation community and then proceeded to apply the lessons learned to the area of graph algorithms. The result, the BGGL (Boost Generic Graph Library) is a tour de force of generic programming. There is a heavy emphasis on flexibility and the book also reflects many of the lessons learned about how to document generic libraries. It is an example of what you can achieve when you take a fresh look at an area applying the tools of Standard C++ [ISO,1998]. ‘‘Boost’’ is a collection of open source libraries designed to augment the C++ standard library [Stroustrup,2000] and a collection of people writing and documenting them [BOOST].

The main point of this article has been to encourage people to build and use good libraries by giving examples and a few simple arguments. Understanding the needs of a potential user community and matching those with design styles and programming techniques is harder than most programmers are willing to believe. Designing, implementing, and documenting a library for an interesting application domain is a worthy intellectual challenge. Getting a library into widespread use within its intended user community is another major challenge involving teaching at many levels. To succeed as users, designers, and implementers, we need to study examples of what can be done and what has been done. Good luck and have fun!

4 References

[Alexandrescu,2001] A. Alexandrescu: *Modern C++ Design*. Addison-Wesley. ISBN 0-201-70431-5. 2001.

[BOOST] A growing collection of open source libraries designed to augment the C++ standard library. http://www.boost.org.

[Budge,1992] Ken Budge, J.S. Perry, and A.C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, OR. August 1992.

[ISO,1998] ISO/IEC 14882, Standard for the C++ Language.

[Koenig,2000] Koenig and B. Moo: *Accelerated C++*. Addison-Wesley. ISBN 0-201-70353-X. 2000.

[MTL] MTL (Matrix Template Library) from University of Notre Dame. http://www.osl.iu.edu/research/mtl/.

[POOMA] POOMA (Parallel Object-Oriented Methods and Applications) from Los Alamos National Labs. http://www.acl.lanl.gov/Pooma/. This library is used for parallel scientific computation.

Copyright InformIt. 2002

[ROOT] ROOT (from CERN. http://root.cern.ch/root/. This library is used for data analysis of large volumes (for example, many terrabytes) of experimental data.

[Schmidt,2002] D. C. Schmidt and S. D. Huston: *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley. ISBN 0-201-60464-7. 2002.

[Siek,2002] J. Siek, L. Lee, and A. Lumsdaine: *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley. ISBN 0-201-72914-8. 2002.

[Stroustrup,1999] B. Stroustrup: *Learning Standard C++ as a New Language*. C/C++ Users Journal. pp 43-54. May 1999. Also in CVU Vol 12 No 1. January 2000. http://www.research.att.com/~bs/new_learning.pdf.

[Stroustrup,2000] B. Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. ISBN 0-201-70073-5. 2000.

[Stroustrup,2001] B. Stroustrup: *C++ glossary*. http://www.research.att.com/~bs/glossary.html

[Sutter,2000] H. Sutter: *Exceptional C++*. Addison-Wesley. ISBN 0-201-61562-2. 2000.

Copyright InformIt. 2002

“New” Value Terminology

Bjarne Stroustrup

Background

The terms “lvalue” and “rvalue” are deep in C++’s genes. They were introduced by Christopher Strachey for CPL [Strachey,196?], the ancestor to BCPL. Dennis Ritchie used “lvalue” to describe C (e.g. see [K&R,1978]), but left out “rvalue”, considering “lvalue” and “not lvalue” sufficient. I did the same for early definitions of C++ (e.g. see [Stroustrup,1986] and [Ellis,1989]). The terms “lvalue” and “rvalue” are “all over” the draft C++ standard. Clearly, this is not terminology to mess with without serious reason and great care. So – despite appearances and rumors to the contrary – we didn’t. Key terms, such as “lvalue” and “rvalue” retain their conventional meaning in the FCD. Furthermore, I claim that the terminology used in the FCD [Becker,2010] and presented in [Miller,2010a] does not represent “random personal preferences” but is derived from an analysis of fundamental properties and selected from a very constrained set of options. Anyone doing a proper analysis should come to a very similar set of concepts and names for those.

Precise wording for specification is hard, so to aid precision and to resolve some known specification problems (related to rvalue references) Mike Miller wrote a paper with the unpromising title *Rvalue References as ‘Funny’ Lvalues* proposing a changed terminology [Miller,2010]. I did not consider “messing with terminology” late in the standards process a good idea and when Daniel Krügler pointed out that the definition of “rvalue” in that proposal differed from the use of “rvalue” in the standard library, I considered the proposal dead.

The model

However, the majority of the CWG disagreed and insisted that some changed and/or novel terminology was necessary to address known problems and to get the specification consistent. This led to a long (days) confused discussion with many suggestions – some novel and interesting. Eventually, I got sufficiently worried to ask a series of questions about what characterized the various terms, which Steve Adamczyk and others answered. The problems being addressed and their suggested resolutions had already received years of consideration, so my questions had solid answers in the context of the WP. By lunchtime, Wednesday March 13, the whiteboard in the CWG looked like this (the handwriting is mostly Steve’s):

Now I was seriously worried. Clearly we were headed for an impasse or a mess or both. I spent the lunchtime doing an analysis to see which of the properties (of values) were independent. There were only two independent properties:

- "has identity" – i.e. and address, a pointer, the user can determine whether two copies are identical, etc.
- "can be moved from" – i.e. we are allowed to leave to source of a "copy" in some indeterminate, but valid state

This led me to the conclusion that there are exactly three kinds of values (using the regex notational trick of using a capital letter to indicate a negative – I was in a hurry):

- iM: has identity and cannot be moved from
- im: has identity and can be moved from (e.g. the result of casting an lvalue to a rvalue reference)
- Im: does not have identity and can be moved from

The fourth possibility ("IM": doesn't have identity and cannot be moved) is not useful in C++ (or, I think) in any other language.

In addition to these three fundamental classifications of values, we have two obvious generalizations that correspond to the two independent properties:

- i: has identity
- m: can be moved from

This led me to put this diagram on the board (my handwriting):

Please note that the labels on the diagram (representing the three fundamental kinds of values and the two generalizations) are not names; they are abbreviations of lists of properties claimed to be primitive (in the mathematical sense).

Naming

I observed that we had only limited freedom to name: The two points to the left (labeled “iM” and “i”) are what people with more or less formality have called “lvalues” and the two points on the right (labeled “m” and “Im”) are what people with more or less formality have called “rvalues.” This must be reflected in our naming. That is, the left “leg” of the W should have names related to “lvalue” and the right “leg” of the W should have names related to “rvalue.” I note that this whole discussion/problem arise from the introduction of rvalue references and move semantics. These notions simply don’t exist in Strachey’s world consisting of just rvalues and lvalues. Someone observed that the ideas that

- Every value is either an lvalue or an rvalue
- An lvalue is not an rvalue and an rvalue is not an lvalue

are deeply embedded in our consciousness, very useful properties, and traces of this dichotomy can be found all over the draft standard. We all agreed that we ought to preserve those properties (and make them precise). This further constrained our naming choices. I observed that the standard library wording uses “rvalue” to mean “m” (the generalization), so that to preserve the expectation and text of the standard library the right-hand bottom point of the W should be named “rvalue.”

This led to a focused discussion of naming. First, we needed to decide on “lvalue.” Should “lvalue” mean “iM” or the generalization “i”? Led by Doug Gregor, we listed the places in the core language wording where the word “lvalue” was qualified to mean the one or the other. A list was made and in most cases and in the most tricky/brittle text “lvalue” currently means “iM”. This is the classical meaning of lvalue because “in the old days” nothing was moved; “move” is a novel notion in C++0x. Also, naming the top-left point of the W “lvalue” gives us the property that every value is an lvalue or an rvalue, but not both.

So, the top left point of the W is “lvalue” and the bottom right point is “rvalue.” What does that make the bottom left and top right points? The bottom left point is a generalization of the classical lvalue, allowing for move. So it is a “generalized lvalue.” We named it “glvalue.” You can quibble about the abbreviation, but (I think) not with the logic. We assumed that in serious use “generalized lvalue” would somehow be

abbreviated anyway, so we had better do it immediately (or risk confusion). The top right point of the W is less general than the bottom right (now, as ever, called “rvalue”). That point represent the original pure notion of an object you can move from because it cannot be referred to again (except by a destructor). I liked the phrase “specialized rvalue” in contrast to “generalized lvalue” but “pure rvalue” abbreviated to “prvalue” won out (and probably rightly so). So, the left leg of the W is “lvalue” and “glvalue” and the right leg is “prvalue” and “rvalue.” Incidentally, every value is either a glvalue or a prvalue, but not both.

This leaves the top middle of the W: “im”; that is, values that have identity and can be moved. We really don’t have anything that guides us to a good name for those esoteric beasts. They are important to people working with the (draft) standard text, but are unlikely to become a household name. We didn’t find any real constraints on the naming to guide us, so we picked ‘x’ for the center, the unknown, the strange, the xpert only, or even x-rated.

Here is Steve showing off the final product:

Note the classification of current uses of “lvalue” (with various classifications) in the pre-Pittsburgh WP to the left (mostly in Doug’s handwriting).

In Pittsburgh (and a few places later), I had the opportunity to explain this terminology to several people. It can be done in a few minutes. I think it has prvalue (Groan!). Members of the CWG reports that the new terminology has already helped clarify and resolve issues.

Summary

The (supposedly new) terminology in the FCD (and presented in [Miller,2010a]) is logical (derived from fundamental language properties), in accordance with the history of C++ and all of its ancestors, in accordance with the pre-Pittsburgh wording, consistent with the terminology used to specify the standard library, and solves a few specification problems that several members of the CWG deemed important.

References

- [Becker,2010] The FCD.
- [K&R,1978] B. Kernighan and D. Ritchie: *The C programming Language*. Prentice-Hall 1978.
- [Miller,2010] William M. Miller: *Rvalue References as "Funny" Lvalues (Rev.2)*. WG21 10-0020=N3030.
- [Miller,2010a] William M. Miller: *A Taxonomy of Expression Value Categories*.10-0045=N3055
- [Strachey, 1963]: D.W. Barron, et al: *The main features of CPL*. The Computer Journal. 6 (2): 134. (1963)
- [Stroustrup,1986] B. Stroustrup: The C++ Programming Language. Addison-Wesley. 1986.
- [Ellis,1989]: M. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. 1989. ("The ARM").

Abstraction and the C++ machine model

Bjarne Stroustrup
Texas A&M University
(and AT&T Labs – Research)
http://www.research.att.com/~bs

Abstract

C++ was designed to be a systems programming language and has been used for embedded systems programming and other resource-constrained types of programming since the earliest days. This paper will briefly discuss how C++'s basic model of computation and data supports time and space performance, hardware access, and predictability. If that was all we wanted, we could write assembler or C, so I show how these basic features interact with abstraction mechanisms (such as classes, inheritance, and templates) to control system complexity and improve correctness while retaining the desired predictability and performance.

Ideals and constraints

C++ [ISO, 2003] [Stroustrup, 2000] is used in essentially every application areas, incl. scientific calculations, compilers, operating systems, device drivers, games, distributed systems infrastructure, animation, telecommunications, embedded systems applications (e.g. mars rover autonomous driving), aeronautics software, CAD/CAM systems, ordinary business applications, graphics, e-commerce sites, and large web applications (such as airline reservation). For a few examples of deployed applications, see http://www.research.att/~bs/applications.html.

How does C++ support such an enormous range of applications? The basic answer is: “by good use of hardware and effective abstraction”. The aim of this paper is to very briefly describe C++’s basic model of the machine and how it’s abstraction mechanisms map a user’s high-level concepts into that model without loss of time of space efficiency. To put this in context, we must first examine the general ideals for programming that C++ is designed to support:

- Work at the highest feasible level of abstraction

Code that is expressed directly using the concepts of the application domain (such as band diagonal matrices, game avatar, and graphics transforms) is more easy to get correct, more comprehensible, and therefore more maintainable than code expressed using low-level concepts (such as bytes, pointers, data structures, and simple loops). The use of "feasible" refers to the fact that the expressiveness of the programming language used, the availability of tools and libraries, the quality of optimizers, the size of available memory, the performance of computers, real-time constraints, the background of programmers, and many other factors can limit our adherence to this ideal. There are still applications that are best written in assembler or very-low-level C++. This, however, is not the ideal. The challenge for tool builders is to make abstraction feasible (effective, affordable, manageable, etc.) for a larger domain of applications.

By "abstract", I do not mean "vague" or "imprecise". On the contrary, the ideal is one-to-one correspondence between application concepts and precisely defined entities in the source code:

- Represent concepts directly in code
- Represent independent concepts independently in code
- Represent relationships among concepts directly in code
- Combine concepts freely in code when (and only when) combination makes sense

Examples of "relationships among concepts" are hierarchies (as used in object-oriented programming) parameterized types and algorithms (as used in generic programming).

This paper is about applying these ideas to embedded systems programming, and especially to hard-real time and high-reliability embedded systems programming where very low-level programming techniques traditionally have been necessary.

What's special about embedded systems programming? Like so many answers about programming, this question is hard to answer because there is no generally accepted definition of "embedded systems programming". The field ranges from tiny controllers of individual gadgets (such as a car window opener), through stereo amplifiers, rice cookers, and digital cameras, to huge telephone switches, and whole airplane control systems. My comments are meant to address all but the tiniest systems: there can be no ISO C++ on a 4-bit micro-processor, but anything larger than that could potentially benefit from the ideals and techniques described here. The keys from a system design view are

- The system is not just a computer
  - It's a "gadget"/system containing one or more computers
- Correctness
  - "but the hardware misbehaved" is often no excuse
- Reliability requirements
  - Are typically more stringent than for an "ordinary office application"
- Resources constraints
  - Most embedded systems suffer memory and/or time constraints

- Real time constraints
  - Hard or soft deadlines
- No operator
  - Just users of “the gadget”
- Long service life
  - Often a program cannot be updates for the years of life of its gadget
- Some systems can’t be taken down for maintenance
  - Either ever or for days at a time

What does C++ have to offer in this domain that is not offered by assembler and C? In particular, what does the C++ abstraction mechanisms offer to complement the model of the machine that C++ shares with C? For a discussion of the relationship between C and C++, see [Stroustrup, 2002].

Machine model

C++ maps directly onto hardware. Its basic types (such as **char**, **int**, and **double**) map directly into memory entities (such as bytes, words, and registers), most arithmetic and logical operations provided by processors are available for those types. Pointers, arrays, and references directly reflect the addressing hardware. There is no “abstract”, “virtual” or mathematical model between the C++ programmer’s expressions and the machine’s facilities. This allows relatively simple and very good code generation. C++’s model, which with few exceptions is identical to C’s, isn’t detailed. For example, there is nothing in C++ that portably expresses the idea of a 2nd level cache, a memory-mapping unit, ROM, or a special purpose register. Such concepts are hard to abstract (express in a useful and portable manner), but there is work on standard library facilities to express even such difficult facilities (see the ROMability and hardware interface sections of [ISO, 2005]). Using C++, we can get really close to the hardware, if that’s what we want.

Let me give examples of the simple map from C++ types to memory. The point here is not sophistication, but simplicity.

Basic arithmetic types are simply mapped into regions of memory of suitable size. A typical implementation would map a **char** to a byte, an **int** to a word, and a **double** to two words:

The exact map is chosen so as to be best for a given type of hardware. Access to sequences of objects is dealt with as arrays, typically accessed through pointers holding machine addresses. Often code manipulating sequences of objects deal with a pointer to the beginning of an array and a pointer to one-beyond-the-end of an array:

The flexibility of forming such addresses by the user and by the code generators can be important.

User-defined types are created by simple composition. Consider a simple type **Point**:

```
class Point { int x; int y; /* … */ };
Point xy(1,2);
Point* p = new Point(1,2);
```

A **Point** is simply the concatenation of its data members, so the size of the **Point xy** is simply two times the size of an **int**. Only if we explicitly allocate a **Point** on the free store (the heap), as done for the **Point** pointed to by **p,** do we incur memory overhead (and allocation overhead). Similarly, basic inheritance simply involves the concatenation of members of the base and derived classes:

```
class X { int b; }
class Y : public X { int d; };
```

Only when we add virtual functions (C++'s variant of run-time dispatch supplying run-time polymorphism), do we need to add supporting data structures, and those are just tables of functions:

```
class Shape {
public:
        virtual void draw() = 0;
        virtual Point center() const = 0;
```

```
        // …
};

Class Circle : public Shape {
        Point c;
        double radius;
public:
        void draw() { /* draw the circle */ }
        Point center() const { return c; }
        // …
};

Shape* p = new Circle(Point(1,2),3.4);
```

Naturally, this simple picture leaves out a lot, but when it comes to estimating time and space costs it's pretty accurate: What you see is what you get. For more details see [ISO, 2005]. In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for [Stroustrup, 1994]. And further: What you do use, you couldn't hand code any better.

Please note that not every language provide such simple mappings to hardware and obeys these simple rules. Consider the C++ layout of an array of objects of a user-defined type:

```
class complex { double re, im; /* … */ };
complex a[ ] = { {1,2}, {3,4} };
```

The likely size is **4*sizeof(double)** which is likely to be 8 words. Compare this with a more typical layout from a "pure object-oriented language" where each user-defined object is allocated separately on the heap and accessed through a reference:

The likely size is **3*sizeof(reference)+3*sizeof(heap_overhead)+4*sizeof(double)**. Assuming a reference to be one word and the heap overhead to be two words, we get a likely size of 17 words to compare to C++'s 8 words. This memory overhead comes with a run-time overhead from allocation and indirect access to elements. That indirect access to memory typically causes problems with cache utilization and limits ROMability.

Myths and limitations

It is not uncommon to encounter an attitude that "if it's elegant, flexible, high-level, general, readable, etc., it must be slow and complicated". This attitude can be so ingrained that someone rejects essentially every C++ facility not offered by C without feeling the need for evidence. This is unfortunate because the low-level alternative involves more work at a lower level of abstraction, more errors, and more maintenance headaches. Bit, byte, pointer, and array fiddling should be the last resort rather than the first choice. C++ balances costs with benefits for "advanced features", such as classes, inheritance, templates, free store (heap), exceptions, and the standard library. If you need the functionality offered by these facilities, you can rarely (if ever) provide better hand-coded alternatives. The ISO C++ standard committee's technical report on performance [ISO, 2005] is provides data and arguments for that proposition.

Obviously, we should not use every feature of C++ for every problem. In particular, not every feature is suitable for hard real time because their performance is not 100% predictable (that is, we can't state in advance exactly how much an operation cost without knowing how it is used and the/or state of the program when it is used). The operations with this problem are:

- Free store (**new**/**delete**): The time needed for an allocation depends on the amount of available free memory and fragmentation can cause deterioration of performance over time. This implies that for many systems, free store cannot be used or can be used only at startup time (no deallocation implies no fragmentation). Alternatives are static allocation, stack allocation, and use of storage pools.
- RTTI (**dynamic_cast**/**typeid**): This is rarely needed in small embedded systems, so just don't use it for such systems. It is possible to implement **dynamic_cast** to

be fast and predictable [Gibbs, 2005] but current implementations don't implement this refinement.

- Exceptions (**throw**/**catch**): The time needed to handle an exception depends on the distance (measured in function calls) from the throw-point to the catch-point and the number of objects needed to be destroyed on the way. Without suitable tools that's very hard to predict, and such tools are not available. Consequently, I can't recommend exceptions for hard real time; doing so is a research problem, which I expect to be solved within the decade. For now, we must use more conventional error-handling strategies when hard real time is needed, and restrict the use of exceptions to large embedded systems with soft real time requirements.

The rest of C++ (including classes, class hierarchies, and templates) can be used and has been used successfully for hard real time code. Naturally, this requires understanding of the facilities and their mapping to hardware, but that's no different from other language constructs. Writing code for hard-real-time or high-reliability systems also requires caution and a good compiler (see http://www.research.att/~bs/compilers.html). It is worth noting that for many styles of usage, modern exception implementations are within 5% of the performance of non-exception code – and that non-exception code must be augmented with alternative exception-handling code (returning error codes, explicit tests, etc.). For systems where exceptions can be used, I consider them the preferred basis for an error-handling strategy [Stroustrup, 2000].

Compilers used for embedded systems programming have switches to disable features where they are undesirable (e.g. in a hard-real time application). Anyway, their use is obvious from the source code.

Abstraction mechanisms

The main abstraction mechanisms provided by C++ are classes, inheritance of classes, and templates. Here, I'll concentrate on templates because they are the key tool for modern statically type-safe high-performance code. Templates are a compile-time composition mechanism implying no runtime or space cost compared to equivalent hand-written code. Templates allow you to parameterize classes and functions with types and integers. If you like fancy words, they provide parametric polymorphism complementing the ad-hoc polymorphism offered by class hierarchies. Generally, systematic use of templates is called "generic programming" which complements the "object-oriented programming" that you get from systematic use of class hierarchies. Both programming paradigms rely on classes.

I will first present some simple "textbook examples" to illustrate the general techniques and tradeoffs. After that, I'll show some real code from a large marine diesel engine using those same facilities to provide reliability, safety, and performance.

Here is a slightly simplified version of the C++ standard library complex type. This is a template class parameterized by the scalar type used:

```
template<class Scalar>
class complex {
    Scalar re, im;
public;
    complex() { }
    complex(Scalar x) : re(x) { }
    complex(Scalar x, Scalar y) : re(x), im(y) { }

    complex& operator+=(complex z) { re+=z.re; im+=z.im; return *this; }
    complex& operator+=(Scalar x) { re+=x; return *this; }

    // …
};
```

This is a perfectly ordinary class definition, providing data members (defining the layout of objects of the type) and function members (defining valid operations). The **template<class Scalar>** says that **complex** takes a type argument (which it uses as its scalar type). Given that definition – and nothing else – we can write

```
complex<double> z(1,2);     // z.re=1; z.im=2;
complex<float> z2 = 3;      // z2.re=3;

z += z2;                    // z.re=z.re+z2.re; z.im=z.im+z2.im;
```

The comments indicate the code generated. The point is that there is no overhead. The operations performed are at the machine level exactly those required by the semantics. A **complex<double>** is allocated as two **double**s (and no more) whereas a **complex<float>** is allocated as two **float**s. A **complex<int>** would make a rather good **Point** type. No code or space is generated for the template class **complex** itself and since we didn't use the **+=** operation taking a scalar argument, no code is generated for that either. Given a decent optimizer, no code is laid down for the used **+=** operation either. Instead, all the operations are inlined to give the code indicated in the comments.

There are two versions of += to ensure optimal performance without heroic efforts from the optimizer. For example, consider:

```
z+=2;       // z.re+=2
z+=(2,0);   // z.re+=2; z.im+=0;
```

A good optimizer will eliminate the redundant **z.im+=0** in the second statement. However, by providing a separate implementation for incrementing only the real part, we don't have to rely on the optimizer to be that clever. In this way, overloading can be a tool for performance.

We can use the += operation to define a conventional binary +:

```
template<class S>
complex<S> operator+(complex<S> x, complex<S> y)
{
        complex<S> r = x;    // r.re=x.re; r.im=y.im;
        r+=y;                // r.re+=y.re; r.im+=y.im;
}

// define complex variables x and y
complex<double> z = x+y;   // z.re=x.re+y.re; z.im=x.im+y.im;
```

Again the comments indicate the optimal code that existing optimizers generate for this. Basically, the templates map to the implementation model for classes described above to give good use of memory and inlining of simple function calls ensures good use of that memory. By "good" I mean "optimal given a good optimizer" and optimizers that good are not uncommon. The example above might make a simple first test of your compiler and optimizer if you want to see whether it is suitable for an application.

The performance of this code depends on inlining of function calls. It has correctly been observed that inlining can lead to code bloat when a large function is inlined many times (either for many different calls or for s few calls but with different template arguments). However, that argument does not apply to small functions (such as, the += and + defined for **complex**) where the actual operation is smaller and faster than the function preamble and value return. In such cases, inlining provides improvements in both time and space compared with ordinary functions and ordinary function calls. In fact, a popular use of class objects and inline function is to implement parameterization where the parameter can be a single machine instruction, such as < [Stroustrup, 1999].

Inlining a large function is usually a very bad idea. Doing so typically indicates carelessness on behalf of the programmer or a poorly tuned optimizer.

In sharp contrast to the claim that templates cause code bloat, it so happens that templates can be used to save code space. A C++ compiler is not allowed to generate code for an unused template function. This implies that if a program uses only 3 of a template class' 7 member functions, only those three functions will occupy space in memory. The equivalent optimization for non-template classes is not common (the standard doesn't require it) and extremely hard to achieve for virtual functions.

The perfect inlining of small member functions and the guarantee that no code is generated for unused functions is the reason that function objects have become the preferred way of parameterizing algorithms. A function object is an object of a class with the application operator () defined to perform a required action. For example

```
template<class T> struct less {
        bool operator()(const T& a, const T& b) const { return a<b; }
};
```

This function object, **less**, is used by most standard library facilities that need to perform a comparison. The result can be factors of improvement in run time compared to parameterization with a function pointers for algorithms such as **sort()** [Stroustrup, 1999].

Most uses of templates are described as “generic programming” or “template meta-programming”. Both are based on overloading where we let the compiler pick the right implementation based on types (and integer values). The simplest and most familiar example is the compiler choosing the right implementation of + when we add **int**, **double**, **complex**, etc. values. The compiler can pick the right function (or basic operation) based on argument types. Similarly, the compiler will pick the right type for an object based on template arguments.

The selection of types and operations is done at compile time and can lead to major improvements. For example, in an embedded application the indirection through pointers to manipulate device drivers turned out to be the bottleneck. The solution was to replace hand-optimized low-level C with templates parameterized on the device register addresses and object types; a 40% improvement in performance was achieved that way. The resulting code was also much shorter and easier to maintain [O’Riorden, 2004]. Section 5 of [ISO, 2005] contains code illustrating such techniques; the examples there relate to a standard interface to special-purpose registers.

It’s amazing what you can do using these techniques. One place to look for techniques and examples is the STL (the C++ standard library’s framework for containers and algorithms) [Stroustrup, 2000]. Since the STL relies on free store it may not be applicable to your particular embedded application, but the techniques are general. For more advanced/extreme uses labeled “template metaprogramming”, see [Abrahams, 2005] and for lots of examples see the Boost collection of libraries [Boost, 2005].

For generality, it is important that templates can have integer arguments. In particular, you can do arbitrary computations at compile time; compile-time constant folding is just the simplest example.

Code examples

Consider briefly a problem faced by the designers of control and monitoring software for large (100,000Hp+) marine diesel engines at MAN B&W Diesel A/S. These engines simply can’t be allowed to fail (or a huge ship is adrift), the engine computers must potentially work for years without maintenance, and programs must be portable to new generations of computers (since computer generations are shorted than engine generations) [Hansen, 2004].

How can we compute accurately and safely? Using numbers of different accuracies? And detect errors such as dived by zero and overflow? Fast enough for hard-real time? (on rugged hardware based on 25MHz Motorola 68332 processors used for electronic fuel injection). The solution chosen and now running on huge ships on the high seas involves:

- Make a template class for fixed-point arithmetic
  - Fixed point is completely portable
  - Fixed point is most efficient on the relevant processors
- Use template specializations where needed

As expected and required, this solution has zero overhead in time and space.

Consider first an example of a function that performs a critical computation. I have done nothing to this code except adjusting the indentation. I am told that it is easy to read if you understand about the engine. Having seen far worse looking code for far simpler problems, I have no trouble believing that:

```
StatusType<FixPoint16> EngineClass::InternalLoadEstimation(
                                const StatusType<FixPoint16>& UnsigRelSpeed,
                                const StatusType<FixPoint16>& FuelIndex)
{
        StatusType<FixPoint16> sl =UnsigRelSpeed*FuelIndex;

        StatusType<FixPoint16> IntLoad =
          sl*(PointSevenFive+sl*(PointFiveFour-PointTwoSeven*sl))
          - PointZeroTwo*UnsigRelSpeed*UnsigRelSpeed*UnsigRelSpeed;

        IntLoad=IntLoad*NoFuelCylCorrFactor.Get();

        if (IntLoad.GetValue()<FixPoint16ZeroValue)
                IntLoad=sFIXPOINT16_0;

        return IntLoad;
}
```

The 16-bit fixed point type is just an ordinary class:

```
stuct FixPoint16 {
        FixPoint16();
        FixPoint16(double aVal);

        bool  operator==(const FixPoint16& a) const { return val==a.val; }
        bool  operator!=(const FixPoint16&) const;
        bool  operator>(const FixPoint16&) const;
        bool  operator<(const FixPoint16&) const;
        bool  operator>=(const FixPoint16&) const;
        bool  operator<=(const FixPoint16&) const;

        short    GetShort() const;
        float    GetFloat() const;
```

```
        double  GetDouble() const;
private:
        long      val;     // e.g. 16.16
};
```

The real computation (of engine status) takes place on status types (parameterized by arithmetic types, such as **FixPoint16**):

```
template <class T>
struct StatusType {
   StatusType();
   StatusType(const StatusType&);
   StatusType(const T aVal,const unsigned long aStat);

   // Member Compound-assignment operator functions:
   StatusType& operator+=(const StatusType&);

   // Miscellaneous:
   const T& GetValue() const;

   // Access functions for status bits:
   bool isOk() const;
   bool IsValid() const;
private:
   T value;
   unsigned long  fpStatus;  // Bit codes defined by type tagFixPoint16Status
};
```

This template class is designed and implemented using the techniques we saw for **complex**. For its time and space performance, it relies on the same techniques and optimizations. This implies that the techniques (and the tools that supports them) are effective in real-world embedded systems contexts.

The low-level details of the engine and the processor are encoded in constants and encapsulated in functions relying on such constants:

```
template<class T>
inline bool StatusType<T>::IsValid() const
{
   return (bool)((fpStatus & 0x0000FFFF) ==VS_VALID);
}


template <>
StatusType<long>&
        StatusType<long>::operator+=(const StatusType<long>& rhs)
```

```
{
    long sum = value + rhs.value;

    if ((value ^ sum) & (rhs.value ^ sum) & LONG_MSB)  { // overflow
        AppendToStatus(VS_OVERFLOW);
        value = (sum & LONG_MSB ? LONG_MAX : LONG_MIN);
    }
    else {
        value = sum;
    }

    AppendToStatus(rhs.GetStatus());

    return (*this);
}
```

The designers of this software emphasize (my translation from Danish):

- C++ is not just used as ”A better C”
  - Our results far exceeded our outside consultants experience with comparable C-based projects.
- Heavy use of object-oriented techniques
  - Including class hierarchies and virtual functions
- Heavy use of generic programming and templates
  - Essential to avoid code duplication
  - Essential to achieve optimal performance
  - Object-oriented and generic programming used in combination
- A good tool chain is essential

The code does not use exceptions (since it is a hard-real-time program) and free store allocation is only used during startup where memory exhaustion and fragmentation cannot occur.

Acknowledgements

Special thanks to Mogens Hansen and Martin O’Riorden for making their examples available to me and educating me in some newer techniques used in performance-critical and safety-critical embedded systems programming. Also thanks to the members of the ISO C++ standard committee’s Performance working group who collected the information for [ISO, 2005].

References

[Abrahams, 2005] David Abrahams and Aleksey Gurtovoy: “C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond”. Addison Wesley. 2005. ISBN 0-321-22725-5.

[Boost, 2005] www.boost.org.

[Gibbs, 2005] Michael Gibbs and Bjarne Stroustrup: “Fast Dynamic Casting”. Software-Practice&Experience. Wiley. To appear 2005.

[Hansen, 2004] Mogens Hansen: “C++ I embedded systemer”. Elektronik 04. Odense Congress Center. September 2004. And personal communication.

[O’Riorden, 2004] Martin J. O’Riordan: “C++ For Embedded Systems”. And personal communications.

[ISO, 2003] “The C++ Standard” (ISO/IEC 14882:2002). Wiley 2003. ISBN 0 470 84674-7.

[ISO, 2005] “Technical Report on C++ Performance”. ISO.IEC PDTR 18015. (http://www.research.att.com/~bs/performanceTR.pdf).

[Stroustrup, 1994] Bjarne Stroustrup: “The Design and Evolution of C++”. Addison Wesley, 1994. ISBN 0-201-54330-3.

[Stroustrup, 1999] Bjarne Stroustrup: “Learning standard C++ as a new language”. C/C++ Users Journal. May 1999

[Stroustrup, 2000] Bjarne Stroustrup: “The C++ Programming Language”. Addison Wesley. 2000. ISBN 0-201-70073-5.

[Stroustrup, 2002] B. Stroustrup: “C and C++: Siblings”, “C and C++: A Case for Compatibility”, “C and C++: Case Studies in Compatibility”. The C/C++ Users Journal. July, August, and September 2002.

A Brief Look at C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This note describes some key aspects of what C++ is and of how C++ has developed over the years. The perspective is that of an experienced C++ user looking at C++ as a practical tool. No attempts are made to compare C++ to other languages, though I have tried to answer some questions that I have often heard asked by Lisp programmers.

1 Introduction

Like all living languages, C++ has grown and changed over the years. For me, the improvements to C++ have seemed almost glacially slow and inevitable; they are natural developments of C++'s own internal logic and deliberate responses to the experiences of hundred of thousands of users. To many who wanted to use C++ aggressively, this growth has been frustratingly slow and timid. To some who considered C++ only infrequently, the developments have seem like unpredictable lurches into the unknown. To others, C++ has simply been something peripheral about which little concrete was known, but about which a multitude of strange rumors persisted.

However you look at it, C++ has developed significantly since its first appearance. As an example, consider a simple function that sorts a container and then counts the number of entries between `Dahl` and `Nygaard`:

```
template<class C> int cnt(C& v)
{
        sort(c.begin(),v.end());
        C::iterator d = find(v.begin(), v.end(), "Dahl");
        return count(d, find(d, v.end(), "Nygaard"));
}
```

A container is seen as a sequence of elements from `begin()` to `end()`. An `iterator` identifies an element in a container.

This template function will work as described for any container that conforms to the conventions of the C++ standard library with elements that can be compared to string literals. For example:

```
vector<char*> v;      // vector of C-style strings
list<string> lst;     // list of C++ strings
// ...
int i1 = cnt(v);
int i2 = cnt(lst);
```

The types `vector`, `list`, and `string` are parts of the standard C++ library.

Naturally, we need not build the string values `Dahl` and `Nyggard` into our little function. In fact, it is easy to generalize the function to do perform an arbitrary action on a range of values of arbitrary types in a container of arbitrary type.

Clearly, this style of programming is far from traditional C programming. However, C++ has not lost touch with C's primary virtues: flexibility and efficiency. For example, the C++ standard library algorithm `sort()` used here is for many simple and realistic examples several times faster than the C standard library `qsort()` function.

2 The C++ Standard

C++ is a statically-typed general-purpose language relying on classes and virtual functions to support object-oriented programming, templates to support generic programming, and providing low-level facilities to support detailed systems programming. That fundamental concept is sound. I don't think this can be proven in any strict sense, but I have seen enough great C++ code and enough successful large-scale projects using C++ for it to satisfy me of its validity.

By 1989, the number of C++ users and the number of independent C++ implementors and tools providers made standardization inevitable. The alternative was to allow C++ to fracture into dialects. In 1995, the ANSI and ISO C++ standards committees reached a level of stability of the language and standard library features and a degree of precision of the description that allowed a draft standard to be issued [Koenig,1995]. A formal standard is likely in late 1996 or early 1997.

During standardization, significant features and libraries were added to C++. In general, the standards process confirmed and strengthened the fundamental nature of C++ and made it more coherent. A description of the new features and some of the reasoning that led to their adoption can be found in [Stroustrup,1994]. So can discussions of older features and of features that were considered but didn't make it into C++.

2.1 Language Features

Basically Standard C++ is the language described in ''The C++ Programming Language (2nd edition)'' [Stroustrup,1991] with namespaces, run-time type information, plus a few minor features added. Among the many minor improvements, the refinements to the template mechanisms are the most significant.

Here is one of the classical examples of object-oriented programming in C++:

```
class Shape {
        virtual void draw() = 0:
        virtual void rotate(int) = 0;
        // ...
};
```

Class `Shape` is an abstract class; that is, a type that specifies an interface, but no implementation. Specific types that conforms to that interface can be defined. For example, this defines `Circle` to be a kind of `Shape`:

```
class Circle : public Shape {
        Point center;
        int radius;
public:
        Circle(Point, int);  // constructor
        void draw();
        void rotate(int) { }
        // ...
};
```

We can now manipulate all kinds of shapes through their common interface. For example, this function rotates a vector of arbitrary `Shape`s `r` degrees:

```
void rotate_all(vector<Shape*>& v, int r)
{
        for (int i = 0; i<v.size(); i++) v[i]->rotate(r);
}
```

For each `Shape`, the appropriate `rotate()` is called. In particular, if the `Shape` rotated is a `Circle`, `Circle::rotate()` is called.

Consider reading `Shape`s from a stream:

```
void user(istream& ss)
{
        io_obj* p = get_obj(ss); // read object from stream

        if (Shape* sp = dynamic_cast<Shape*>(p)) {  // is it a Shape?
                sp->draw();     // use the Shape
                // ...
        }
        else                    // oops: non-shape in Shape file
                throw unexpected_shape();
}
```

Here, the `dynamic_cast` operator is used to check that the objects really are `Shape`s. Any kind of `Shape`, for example a `Circle`, is acceptable. We throw an exception if an object that is not a `Shape` is encountered.

This example is rather trivial. However, the techniques presented and the language features supporting them have been used in the construction of some of the largest and most demanding applications ever built.

2.2 The Standard Library

The lack of a solid standard library has always been one of C++'s greatest weaknesses. This lack caused a proliferation of incompatible ''foundation libraries,'' and diverted novice C++ programmers from getting real work done into designing basic library facilities. The latter was particularly nasty because it is far harder to design and implement good basic library facilities than using them. This lack of a standard library forced many programmers to deal with advanced C++ features before they had mastered the basics.

The facilities provided by the standard library can be classified like this:

[1] Basic run-time language support (for allocation, RTTI, etc.).

[2] The standard C library (with very minor modifications to minimize violations of the type system).

[3] Strings and I/O streams (with support for international character sets and localization).

[4] A framework of containers (such as, `vector`, `list`, and `map`) and algorithms using containers (such as general traversals, sorts, and merges).

[5] Support for numeric computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization).

The main criteria for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overheads compared to a simpler version of the same facility, and that simple uses should be easy to learn. Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

Every algorithm works with every container without the use of conversions. This framework, conventionally called the STL [Stepanov,1994], is extensible in the sense that users can easily provide containers and algorithms in addition to the ones provided as part of the standard and have these work directly with the standard containers and algorithms.

3 Tools, Environments, and Libraries

The stability resulting from by the near-completion of the standard is causing a boom in work on programming environments, libraries, and tools. Traditionally, much of the effort in the C++ world has been aimed at producing a language that could be used effectively for significant industrial projects even in the absence of advanced tools and environments. This has not stopped excellent C++ tools and environments from appearing, but the evolution of the language has been a significant drag on the C++ implementation and tools communities.

I expect to see really great program development environments to become almost universal over the next few years. Already, features that people had deemed impossible for C++ are available in shipped products. For example, Sun's C++ implementation allows you to stop an executing program at a breakpoint, rewrite a function, and re-start using the new function. This is something that people using interpreted and dynamically-typed languages have taken for granted for decades. However, it is one interesting and significant step towards my goal of a program development environment combining the strengths of a statically-

typed language with the benefits of a sophisticated environment like the ones usually associated with dynamically-typed languages.

One of the benefits of a large user community is the availability of libraries. By now, there is a bewildering variety of C++ libraries, but the development of libraries have been hampered by both the differences between compilers and by the lack of a standard library. The former problem has led to unnecessary segmentation of the community, and to the emergence of libraries especially designed to allow crossplatform development. The latter problem forced library developers to re-invent basic concepts, such as string and list, over and over again. Though it will takes years to work these problems out of the system, we now have an alternative and can get on with more important and interesting tasks.

Automatic garbage collection is possibly the issue over which the C and Lisp communities has traditionally been most at odds. The Lisp community was certain that memory management was far too important to leave to users, and the C community was sure that memory management was far too important to leave to the system. C++ takes an intermediate approach. Automatic garbage collection is possible, but not compulsory in C++. Traditionally, this simply meant that C++ programs didn't use automatic garbage collection, but now both commercial and free garbage collectors for C++ has found their way into non-experimental use. The performance of these collectors is respectable, and in particular, far better than the pessimistic predictions that I have repeatedly heard over the years. Even where a garbage collector isn't used, well-designed C++ programs suffer far less from memory management problems than traditional C programs. Memory management is often encapsulated in user defined types so that users don't have to allocate and deallocate memory directly. In particular, standard containers such as `string`, `vector`, and `list` do their own memory management and provide variable-sized data structures.

4 Programming Styles

C++ is a multi-paradigm language. In other words, C++ was designed to support a range of styles. No single language can support every style. However, a variety of styles that can be supported within the framework of a single language. Where this can be done, significant benefits arise from sharing a common type system, a common toolset, etc. These technical advantages translates into important practical benefits such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages.

For starters, C++ supports traditional C-style. Other styles emphasize the use of classes, abstract classes, class hierarchies, and templates to express concepts and relationships between concepts directly, cleanly, and affordably. For example, §1 used generic programming and §2.1 demonstrated abstract classes and class hierarchies.

Much of the work on styles (for example [Kornig,1995b]) and patterns (for example, [Gamma,1994]) in the C++ community has focussed on finding ways to express ideas from a variety of languages and systems in a way that can be effectively and efficiently be utilized by C++ programmers writing larger production systems. The emphasis is on effective use of C++'s flexible and extensible static type system.

5 Acknowledgements

Thanks to Craig Knoblock for inviting me to write this note.

6 References

[Gamma,1994] Gamma, et.al.: *Design Patterns*. Addison Wesley. 1994. ISBN 0-201-63361-2.

[Koenig,1995] Andrew Koenig (editor): The Working Papers for the ANSI-X3J16 /ISO-SC22-WG21 C++ standards committee.

[Koenig,1995b] Andrew Koenig and Bjarne Stroustrup: *Foundations for Native C++ Styles*. Software − Practice & Experience. 1995.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. ISO Programming language C++ project. Doc No: X3J16/94-0095, WG21/N0482.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd Edition)* Addison Wesley, ISBN 0-201-53992-6. June 1991.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++* Addison Wesley, ISBN 0-201-54330-3. March 1994.

ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221249736

C++ Dynamic Cast in Autonomous Space Systems

Conference Paper · May 2008

DOI: 10.1109/ISORC.2008.20 · Source: DBLP

CITATION
1

READS
43

4 authors, including:

Rabi N. Mahapatra
Texas A&M University
164 PUBLICATIONS **1,492** CITATIONS

SEE PROFILE

Bjarne Stroustrup
Morgan Stanley
157 PUBLICATIONS **8,359** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Thermal Management of Photonic NoC View project

Project Power Management of Photonic NoC View project

All content following this page was uploaded by Bjarne Stroustrup on 29 May 2014.

The user has requested enhancement of the downloaded file.

C++ Dynamic Cast in Autonomous Space Systems

Damian Dechev [1], Rabi Mahapatra [1], Bjarne Stroustrup [1], David Wagner [2]
dechev@tamu.edu, rabi@cs.tamu.edu, bs@cs.tamu.edu, david.a.wagner@jpl.nasa.gov

Texas A&M University [1]
College Station, TX 77843-3112

Jet Propulsion Laboratory, California Institute of Technology [2]
4800 Oak Grove Drive, M/S 301-270, Pasadena, CA

Abstract

The dynamic cast operation allows flexibility in the design and use of data management facilities in object-oriented programs. Dynamic cast has an important role in the implementation of the Data Management Services (DMS) of the Mission Data System Project (MDS), the Jet Propulsion Laboratory's experimental work for providing a state-based and goal-oriented unified architecture for testing and development of mission software. DMS is responsible for the storage and transport of control and scientific data in a remote autonomous spacecraft. Like similar operators in other languages, the C++ dynamic cast operator does not provide the timing guarantees needed for hard real-time embedded systems. In a recent study, Gibbs and Stroustrup (G&S) devised a dynamic cast implementation strategy that guarantees fast constant-time performance. This paper presents the definition and application of a co-simulation framework to formally verify and evaluate the G&S fast dynamic casting scheme and its applicability in the Mission Data System DMS application. We describe the systematic process of model-based simulation and analysis that has lead to performance improvement of the G&S algorithm's heuristics by about a factor of 2.

1 Introduction

ISO Standard C++ [8] has become a common choice for hard real-time embedded systems such as the Jet Propulsion Laboratory's Mission Data System [7]. This is so because ISO C++ offers efficient abstraction model, good hardware use, and predictability. C++'s model of computation has helped engineers deliver more correct, maintainable, and comprehensible software compared to code relying on lower-level programming concepts [19]. However, several C++ features are usually considered unsuitable for programming real-time systems because they do not guarantee predicable constant-time performance [5]. ISO C++ does not provide the necessary timing guarantees for free store (heap) allocation, exception handling, and dynamic casting. In particular, the most common compiler implementations of the dynamic cast operator traverse the representation of the inheritance tree (at run time) searching for a match. Such implementations of dynamic cast are not predictable and are unsuitable for real-time programming. Gibbs and Stroustrup(G&S) [3] describe a technique for implementing dynamic cast that delivers significantly improved and constant-time performance. The key idea is to replace the runtime search through the class hierarchy with a simple (constant-time) calculation, much as the common implementations of the C++ virtual function calls search the class hierarchy at compile time to reduce the runtime action to a simple array subscripting operation. In the G&S scheme, a heuristic algorithm assigns an integer type ID at link time to each class. The type ID assignment rules guarantee that at run time a simple modulo operation can reveal the type information and check the validity of the cast. The requirements for the heuristics assigning the type IDs are that:

(1) They must keep the size of the type ID to a small number of bits. A 64-bit type ID should be sufficient for very large class hierarchies

(2) Avoid conflicts and type ID assignments that create ambiguous or erroneous type resolution at run time

(3) Handle virtual inheritance

There are four heuristic schemes and a few possible optimizations suggested in [3]. However, none of those heuristics guarantee the best solution for every possible class hierarchy. The quality of the type ID assignment heuristics has a critical importance for the performance of the G&S scheme. With better heuristics, a smaller type ID size would be sufficient to facilitate complex and large class hierarchies that would certainly need a significantly bigger type ID size with a poor assignment scheme. The main contribution of this work is to present how the algorithm optimization problem

encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme.

As pointed out by Lowry [10], the increasing complexity of future space missions, such as the Mars Science Laboratory [20] and Project Constellation [17], raises concerns whether it is possible to establish their reliability in a cost-effective manner. Lowry's analysis indicates that at the present moment the verification and certification cost of mission critical software exceeds its development cost. Perrow [13] studies the risk factors in the modern high technology systems. His work identifies two significant hazard dimensions: *interactions* and *coupling*. Complex interactions represent unexpected and unknown sequences and thus cannot be entirely comprehensible at the time of system development. A tightly-coupled system has a number of time-dependent processes that cannot tolerate delays. Perrow classifies space missions in the riskiest category since both hazard factors are present. The dominant paradigms for software development, assurance, and management at NASA rely on the principle "test-what-you-fly and fly-what-you-test". Born out of experience and hindsight, this methodology had been applied in a large number of robotic space missions at the Jet Propulsion Laboratory. For such missions, it has proven suitable in achieving adherence to some of the most stringent standards of man-rated certification such as the DO-178B [14], the Federal Aviation Administration (FAA) software standard. Its Level A requirements demand 100% coverage of all high and low level assurance policies. However, the present certification methodologies are prohibitively expensive for systems of high complexity [16].

In this paper we present a co-simulation framework based on the SPIN model checker [6] to simulate, evaluate, and formally verify the G&S fast dynamic casting algorithm and its application in mission critical code such as the Data Management Services [21] of the Mission Data System. The aim of the Mission Data System is to provide a unified state-based and goal-oriented architecture for building complete data and control systems for autonomous space missions. The framework's state- and model-based methodology and its associated systems engineering processes and development tools have been successfully applied on a number of test systems including the physical rovers Rocky 7 and Rocky 8 and a simulated Entry, Descent, and Landing (EDL) system for the Mars Science Laboratory mission. We use the feedback from the model checker to perform systematic analysis of the G&S scheme and look for improvements to the heuristics for type ID assignment. SPIN is an on-the-fly, linear-time logic model-checking tool that was designed for the formal verification of dynamic systems with asynchronously executed processes. The most recent advances in the state space reduction techniques has made it possible to validate large software applications. Model-checking tools have been widely applied for the verification of a large variety of systems, including flight software [4], network protocols [11], and scheduling algorithms [15]. We are unaware of work suggesting its use for the analysis and optimization of compiler heuristics. Compiler verification usually focuses on seeking a proof on the preservation of the program semantics during the various compiler passes [9]. Our work presents the application of a model-checking tool for the analysis and refinement of the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our co-simulation framework consists of the following components:

(1) An abstract model of the G&S type ID assignment heuristics

(2) An procedure for exhaustive search of the state space discovering the best type ID assignment

The analysis of the heuristics simulation performed in SPIN provides us with ideas of possible improvements to the G&S type ID assignment. We include and evaluate the proposed improvements in the abstract model in order to seek refinement of the G&S type ID assignment scheme. The experiments we have executed show that the G&S priority assignment is not optimal with respect to the best possible type ID assignment where non-virtual multiple inheritance is used. While potentially dangerous if not constructed carefully, such hierarchies happen to be of significant practical importance [18]. Based on our experiments, we suggest optimizations that lead to significant improvement of the G&S heuristics performance. This paper makes the following contributions:

(1) Introduces the use of a co-simulation framework based on model-checking for the analysis and improvement of a compiler-heuristics optimization problem

(2) Verifies and analyzes the G&S C++ fast dynamic casting scheme and its application in mission critical code such as the MDS Data Management Services

(3) Implements optimizations to the G&S heuristics leading to the discovery of optimal type ID assignment in 85% of the class hierarchies, in contrast to 48% for the original G&S algorithm

The rest of the paper is organized as follows: section 2: a brief description of the G&S fast dynamic cast algorithm, section 3: our approach to co-simulation and improvements to the G&S heuristics, section 4: discussion on the challenges of mission critical code and the applicability of the G&S dynamic cast section 5: performance results for the G&S algorithm and the proposed improvements, and section 6: conclusion.

2 Fast Dynamic Casting Algorithm

The G&S fast constant-time implementation of the dynamic cast operator works as follows: at link time, a static integer type ID number, preferably 32 or 64-bit long, is assigned to each class. The ID numbers are selected so that the operation id_a *modulus* id_b yields zero if and only if the class with id_a is derived from the class with id_b. This is done by exploiting the uniqueness of factorization of integers into prime factors. Each class is assigned a *key* prime number. The *type ID* of a class is calculated by multiplying its *key number* with the key numbers of each of its base classes. In the cases where a class contains more than a single copy of a base class, the type ID is computed by taking the square of the corresponding base class ID. The only constraint of the approach is the desire to limit the ID size to fit the machine's built-in integer types. The key primes are not required to be unique and the same prime key can be used for classes that belong to different groups (i.e. do not share common descendants). Gibbs and Stroustrup suggest four approaches for assigning the type IDs in a space-efficient manner. Each method is based on a preliminary computation of the priority factor of each class. The priority reflects the class impact on the growth of the type ID numbers in the hierarchy. Thus, classes with greater number of descendants should receive higher priority and smaller key prime number values respectively. The four possible schemes suggest that:

1 The priority of a class is the *maximum* number of ancestors that any of its descendants has. This scheme was chosen for the initial implementation and testing of the G&S algorithm and also closely followed in the implementation of the abstract model used for our simulation

2 ,3, 4. If a range of primes is assigned to every level with wider levels receiving larger initial values, then each node could be assigned an additional value that is proportional to the logarithm of the (*2. minimum, 3. mean, 4. maximum*) prime in its level. Priorities of hierarchy leaves are computed by taking the sum of these additional values for the leaf itself and all of its ancestor classes

After the priority of each class has been computed, the classes with the highest priority get the smallest prime numbers. According to this scheme, prime numbers can be reused only if there are two classes on the same level of the class hierarchy and only if they do not share common descendants, they are not siblings, and also that none of their parents share a common descendant. According to the ID assignment rules, we know that:

(1) $id_x = k_x \times (k_a)^2 \times k_{a_1} \times (k_b)^2 \times k_{b_1} \times k_c$

(2) $id_y = k_y \times k_c \times k_{c_1} \times (k_d)^2 \times k_{d_1} \times k_b$

Figure 1. A class hierarchy with 11 classes

(3) $id_z = k_z \times k_d \times k_{d_1} \times k_c$

Given a set C with 11 classes in the hierarchy and the set of the first 11 prime numbers *P* = {*2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31*}, we must assign each class V a key $k_v \in P$ such that, the maximum of the set $id_{leaf} = \{id_x, id_y, id_z\}$, the set consisting of the ID numbers of all leaf nodes in C, is minimal. As we already know, prime numbers need not be unique for each class and can be reused in same circumstances.

3 A Co-simulation Framework

The goals of the co-simulation framework are to validate the main invariants of the G&S heuristics, improve its performance, and establish its applicability in mission critical systems. The co-simulation process in the framework (Figure 2) consists of three consecutive stages: verification, evaluation, and analysis. The verification phase is a straightforward application of model checking where an abstract description of the system's behavior is checked against a set of invariants. In the evaluation stage the simulation results from the probabilistic approach are contrasted to the outcome of the deterministic approach. The aim of the analysis stage is to closely examine the instances where the solutions yielded by the two implementations differ. We identify patterns among the inconsistent results that reveal the weaknesses of the probabilistic solution. The framework works by executing two independent models, the G&S model and the exhaustive search model. The first input component to the co-simulation framework (Figure 2) is an abstract model of the G&S fast dynamic casting heuristics, implemented in Promela (SPIN's input language) and the embedded C primitives it allows. The G&S abstract model is subsequently used to verify the main invariants of the G&S heuristics and at the same time provide us with a simulation testbed to examine the heuristics performance. The second component of the framework is the exhaustive search model that simply looks into all possible type ID assignments to discover the optimal solution for a given class hierarchy. We employ SPIN's search engine to perform the exhaustive search. In Algorithm 1 we present the pseudocode of our co-simulation approach. The following sections elaborate in more details on each of the stages of

Figure 2. A Co-Simulation Framework for G&S Improvement and Verification

the framework.

Algorithm 1 Pseudocode of the co-simulation approach.

```
const int MAX_NUMBER_TESTS
VERIFY :
repeat
    Formal Verification (G&S Model) → error report
    if (no errors) then
        goto EVALUATE
    else
        study counter example
        correct G&S
until TRUE
EVALUATE :
count = 0
for (count < MAX_NUMBER_TESTS) do
    Simulation(G&S Model) → G&S solution
    Enumeration(Exhaustive Search Model) → best solution
    if (G&S solution ≠ best solution) then
        add instance to SIS
    count ++
ANALYZE :
for all i ∈ SIS do
    look for a pattern
    modify G&S
    goto EVALUATE
```

3.1 Formal Verification

Every G&S implementation operates under the assumption that when a prime number is reused, it is assigned to non-conflicting classes. In addition, the maximum type ID must fit within the boundaries of a memory word. We check these invariants during the program verification phase. Establishing the validity of the G&S invariants is done by straightforward application of model-checking with SPIN. In SPIN the critical system properties are expressed in the syntax of linear time logic. Based on the G&S abstract specification, the model-checker performs a systematic exploration of all possible states. In case of failure, SPIN provides a counterexample that demonstrates a behavior that has lead to an illegal state. In our model, the invariants are expressed as a *never claim* [6], and are checked just before and after the execution of every statement.

3.2 Evaluation

SPIN has been previously employed to implement solutions of scheduling [1] and discrete optimization [15] problems. The problem we face in the G&S heuristics is a combinatorial optimization problem [12]. Given a finite set I, a collection F of subsets of I, and a real-valued function w defined on I, a discrete optimization problem could be defined as the task of finding a member S of F, such that: $\sum_{e \in S} w(e)$ is as small (or as large) as possible.

Except for the simplest cases, a discrete optimization problem is difficult because its design space is typically disjoint and nonconvex. Therefore, the optimization methods applied to continuous optimization problems cannot be utilized in this case. In a small discrete problem, it would be possible to exhaustively list all possible combinations. As the number of parameters increase, the state explosion makes optimizations difficult. The two general strategies for approaching a discrete optimization problem can be classified as *deterministic* and *probabilistic*. What we do for the G&S exploration in SPIN could be described as applying a deterministic approach for the evaluation of a set of proposed probabilistic methods. The Branch and Bound method [12] guarantees the discovery of the global optimum in the cases when the problem is linear or convex and is the most frequently used discrete optimization method. It is based on the sequential analysis of the discrete tree of each parameter. The branches that can be estimated to reach invalid or unfeasible solutions are consequently eliminated. This simple optimization could also be applied in some limited cases in the SPIN's Fast Dynamic Casting exhaustive search. Let us explore a class hierarchy with three classes A, B, and C, where B is derived from A, and C is derived from both A and B. In this case, we have $C = \{A, B, C\}$, $P = \{2, 3, 5\}$, and $id_{leaf} = \{id_c\}$. The enumeration is given in Table 1. We assume that the computation starts at a state S_0 where all three keys k_a, k_b, and k_c are uninitialized. Then we assign possible values from the set P to the key variables of the classes A, B, and C. The enumeration shown above can be expressed as the computation shown on Figure 3. The graph shows only the valid states of the computation. There are a number of invalid states that are not shown on the graph. For example, according to the rules defined in G&S, it is possible to reuse some of the prime numbers in

$id_c = k_c \times k_b \times (k_a)^2$	k_a	k_b	k_c
60	2	3	5
60	2	5	3
90	3	2	5
90	3	5	2
150	5	2	3
150	5	3	2

Table 1. Enumeration of all solutions

Figure 3. Exhaustive search computation

P. Thus, we can try and add an edge $k_b = 2$ in state S_1, however the reuse of 2 in this case is invalid since A and B are conflicting classes.

The illustrated automation in Figure 3 provides a foundation for the construction of a Promela model for the deterministic solution. Each possible prime number assignment to a given class key is represented by a separate state transition in the exhaustive search model. SPIN initiates the optimum search at state S_0 and visits all possible states. At each end state the value of the minimum of the set of leaves, in this case represented only by id_c, is computed and compared to the current minimum. This approach is similar to the algorithm described by Ruys in [15] and shown in Algorithm 2. For such an application, we use the model checker in a somewhat unusual fashion. In this scenario, the validation property checks whether the value of id_c is greater than the current minimum. Each time this condition is violated, the current minimum is updated and the process is automatically repeated until SPIN confirms that there are no routes violating the specification. Since the solution is deterministic, it is guaranteed to discover the global optimum for type ID assignment. The performance of the G&S heuristics is measured by running a simulation of the G&S model that has been used earlier for verification. Now we are left with only one important task (not automated at this stage), the comparison of the results from the probabilistic and deterministic solutions. Once we identify a set of inconsistent results, we try to find a pattern and refine the G&S heuristics. Then the refined scheme is implemented in the probabilistic model and the evaluation process is reiterated.

Algorithm 2 Finding the global minimum in the state space.

1: $intput: \ Promela\ model\ M$
2: $output: \ the\ optimal\ minimum\ for\ the\ problem\ M$
3: $min = (worst\ case)\ maximum\ value\ for\ id$
4: **repeat**
5: $\quad use\ SPIN\ to\ check\ M\ with\ condition\ (id_c > min)$
6: $\quad$ **if** $(error\ found)$ **then**
7: $\quad\quad min = id_c$
8: **until** $(error\ found)$

3.3 Analysis

The simulation and enumeration models are continuously executed until, if possible, a set of instances with inconsistent solutions can be identified. Thus, each instance in the Set of Inconsistent Solutions(SIS), represents a given class hierarchy for which the deterministic and probabilistic approach have discovered different solutions. The class hierarchies for each test could be guided or created in a random fashion. For the generation of the test data in our experiments we implemented a pseudo random class hierarchy generation algorithm, in a manner similar to the TGFF(Task-Graphs-For-Free) method as described in [2]. We look for patterns among the collected hierarchies in SIS and seek clues that can lead us to improvements of the G&S scheme. Potential improvements are tested by adding them to the G&S model and evaluate their effect. Such scheme modifications are carefully selected since it is possible that they might enhance a given G&S feature and at the same time weaken another. Ideally, the improvements lead to a heuristic scheme that provides the best solutions for a larger number of the test hierarchies and at the same time has a time complexity equal to or less than the earlier heuristic scheme.

Despite the numerous advanced state space reduction techniques utilized by the SPIN model checker, little can be done to further optimize the exhaustive search. The main goal of our experiments is to reach quick and effective optimization of the G&S scheme, thus the class hierarchies considered were not the largest and most complex that our models can handle. The models developed for our experiments are capable of handling class hierarchies of double or triple the size of the ones presented in the paper, and even larger number of classes can be facilitated with increased computational power. In the framework, the exhaustive search is used to identify flaws in the G&S type ID assignment scheme, thus, there is no need to create and simulate much larger hierarchies. In this work our goal is to demonstrate that the current size of the class hierarchies is sufficient to discover significant flaws of the original heuristics.

4 Application in Mission-Critical Software

Modern space mission systems have evolved from simple embedded devices into complex computing platforms with high autonomy and an exceptionally large demand for human-computer interaction. Consequently, such systems require reliable and flexible data systems managing the collection, storage, and transportation of data. The Mission Data System(MDS) is the Jet Propulsion Laboratory's state- and goal-oriented framework for building embedded control systems with a high degree of autonomy. MDS provides the building blocks for the implementation of embedded platforms based on the concepts of state estimation and control. The Data Management Services(DMS) is the MDS component responsible for the production, storage, processing, and transfer of control and scientific data. In [21] Wagner defines the challenges of data management in MDS as the problems of producing and storing data and converting the data to various formats as needed by its consumers. In addition, DMS needs to ensure the secure and lossless transport of the data with limited resources and through unreliable physical medium. To design and relate the data system entities, DMS employs concepts from high-level ISO C++ including templates, object-oriented class encapsulation, and dynamic casting necessary for the conversion of the data formats.

The actual telemetry data objects in MDS communicate with each other via byte streams produced by the transport protocol (e.g. spacecraft to ground communication). The receiver of the telemetry data needs to recreate the data object from the byte stream and thus invoke type casting in numerous occasions. Constant-time dynamic cast is also needed by the MDS Goal Network in the case when a controller or estimator [21] passes a goal via the Coordinating Goal Network(CGN), typically a large dynamic data structure. In CGN the goal is propagated using only its abstract attributes(start and end time, and the associated state variable). The achiever object who eventually picks up the goal needs to reconstruct the data object via dynamic downcasting to the specific type that conveys the state-specific achievement criteria. The application of the common compiler implementation of dynamic cast has proved to be unacceptable due to poor performance and the lack of the timing guarantees.

The G&S scheme was devised as a solution to a real industrial problem related to C++ use for hard real time code. Inquiries in the C++ community revealed that the problem was fundamental and common, rather than isolated: developers simulate dynamic casting with other language features, leading to type-unsafe special-purpose code or the avoidance of best object-oriented practices. Naturally, such workaround code slows down development, complicates maintenance, and increases the need for testing.

5 Results

We applied the co-simulation process described in the previous section to a large number of class hierarchies. The tested hierarchies are not built into our models. Instead, we have followed a pseudo random generation methodology similar to TGFF task graph generation as described in [2] to automatically generate hundreds of possible test cases. For illustration, we show the results from a set of seven pseudo random class hierarchies (Appendix A). The results of the G&S heuristics model and the exhaustive search are shown in Table 2. A brief comparison of the results indicates that the G&S heuristics do not give optimal performance for class hierarchies with non-virtual multiple-inheritance. A closer look at the algorithm reveals that the priority calculation routine takes into account only the number of descendants that each class has. Let us consider the class hierarchy from test case 7. We notice that according to the current scheme, the base classes 0, 1, and 2 all get the same priority rank since they all share the descendant 6. Class 6 is at the lowest level of the hierarchy and has the largest number of ancestors. If we would like to optimize the heuristics, we must find a way to increase the priority of base class 2. Our reasoning is derived from the fact that Class 2 is ambiguous and the leaf Class 6 contains two copies of it. Similarly, let us have a closer look at test case 1. In the optimal solution, Class 5 takes the lower prime number (11) compared to Class 4, despite the fact that its only descendant has less ancestors compared to Class 4. The reason for this result is the fact that the derived Class 3 contains two ambiguous bases while Class 4 contains only one ambiguous base. As a result of our analysis we conclude that higher priority should be given to derived classes and their ancestors who contain more ambiguous base classes. To fix these weaknesses, we extend the G&S heuristics by adding two simple rules:

(1) We count every ambiguous ancestor twice when we determine the number of ancestors to each class

(2) For each base class, we count the number of derived classes that include more than one copy of it, and add that number directly to its priority

We call this enhanced G&S heuristics Fast Dynamic Casting Plus(FDC+). As Table 2 shows, for the initial set of test cases, FDC+ performance is 100% equivalent to the performance of the deterministic approach. In the performed tests, we have generated 127 pseudo random class hierarchies and applied G&S, FDC+, and the exhaustive search to each one of them. The experimental results showed that FDC+ was able to yield the best type ID assignment in 85% of the class hierarchies compared to 48% for the G&S heuristics. The time performance of the three schemes is shown in Figure

Case No	G&S	Exhaustive search	FDC+
Case 1 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 2, 5, 7, 13, 11, 17)	(3, 2, 5, 7, 13, 11, 17)
Case 1(ids of all leaves)	(16380, 16830)	(13860, 13260)	(13860, 13260)
Case 2 (keys)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)	(2, 13, 3, 5, 17, 7, 11)
Case 2 (ids of all leaves)	(1326, 2310)	(1326, 2310)	(1326, 2310)
Case 3 (keys)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)	(2, 3, 13, 5, 7, 17, 11)
Case 3 (ids of all leaves)	(26, 51, 2310)	(26, 51, 2310)	(26, 51, 2310)
Case 4 (keys)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)	(2, 3, 5, 7, 11, 13, 17)
Case 4 (ids of all leaves)	(2310, 1547)	(2310, 1547)	(2310, 1547)
Case 5 (keys)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)	(2, 3, 5, 7, 11, 7, 11)
Case 5 (ids of all leaves)	(42, 66, 70, 110)	(42, 66, 70, 110)	(42, 66, 70, 110)
Case 6 (keys)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)	(2, 3, 5, 11, 13, 7, 17)
Case 6 (ids of all leaves)	(66, 78, 420, 170)	(66, 78, 420, 170)	(66, 78, 420, 170)
Case 7 (keys)	(2, 3, 5, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)	(3, 5, 2, 7, 11, 13, 17)
Case 7 (ids of all leaves)	(2552550)	(1021020)	(1021020)

Table 2. Co-simulation of the seven cases from Appendix A

4. While the time performances of the G&S and FDC+ algorithms are equal and both run in a very low constant-time (the function at 00:01 min on Figure 4), logically the time performance of the exhaustive search increases exponentially with the increase of the number of classes nodes in a given class hierarchy. The analysis of the test results indicated that FDC+ finds a better type ID compared to the G&S approach in 39% of the test scenarios. For the greater part of the test cases, FDC+ matched the optimal type ID assignment computed by the exhaustive search. This efficiency boost is due to the optimized performance of FDC+ in the cases where multiple non-virtual inheritance is present in the class hierarchy. We have also observed that the implementation of these optimizations does not lead to efficiency loss in other scenarios and the performance of FDC+ is always at least as good as the performance of G&S. Our experimental results have indicated that the introduced optimizations in FDC+ have fixed a weakness of the original G&S approach and have improved the success rate in finding the best type ID assignment. The G&S scheme requires a key of a memory size that is a function of the size and shape of a class hierarchy. Thus, the improved heuristics almost double the size of class hierarchies that can be handled by a given key size. Since the scheme gets significantly slower when a key gets too large for a machine word, the improvements to the heuristics address the main limitation of the G&S scheme.

Figure 4. Search time for type ID assignment

6 Conclusion

In this work we applied co-simulation of the deterministic and probabilistic solutions to the combinatorial optimization problem posed by the G&S type ID assignment scheme. Our framework proved successful in verifying and refining the existing G&S heuristics. We demonstrated how we use the simulation results to devise improvements to the G&S algorithm and evaluate them. The results from our experiments indicate that the improved G&S heuristics (FDC+) provide the optimal type ID assignment in 85% of the class hierarchies, compared to 48% for the regular G&S algorithm. The efficiency of the type ID assignment scheme has significant importance for the performance of the fast dynamic casting by Gibbs and Stroustrup [3]. This paper presented a practical approach of how to discover improvements to the type ID assignment scheme in a simple and effective manner. The main advantage of the presented approach is the ease and simplicity of the discovery and test for potential improvements. The improved heuristics that we have described in this work almost doubles the size of class hierarchies that can be handled by a given key size. A more extensive simulation might suggest further improvements to the type ID assignment scheme. Our main goal in this work has been to demonstrated how an algorithm optimization problem encountered has been successfully automated and moreover that its automation has led us to quick but significant improvements of the initial scheme. In the future we intend to utilize a static analysis tool for automatic class hierarchy analysis and extraction.

7 Acknowledgements

We thank Peter Pirkelbauer, Kirk Reinholtz, Gerard Holzmann, and the anonymous referees for their helpful suggestions.

A Appendix

Figure 5. Test Cases 1 and 2

Figure 6. Test Cases 3 and 4

Figure 7. Test Cases 5 and 6

Figure 8. Test Case 7

References

[1] E. Brinksma and A. Mader. Verification and Optimization of a PLC Control Schedule. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 73–92, London, UK, 2000. Springer-Verlag.

[2] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society.

[3] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36(2):139–156, 2006.

[4] R. Gluck and G. Holzmann. Using spin model checker for flight software verification. In *In Proceedings of the 2002 IEEE Aerospace Conference*, 2002.

[5] L. Goldthwaite. Technical Report on C++ Performance. In *ISO/IEC PDTR 18015*, February 2006.

[6] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[7] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada. Engineering Complex Embedded Systems with State Analysis and the Mission Data System. In *In Proceedings of First AIAA Intelligent Systems Technical Conference 2004*, 2004.

[8] ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.

[9] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231, New York, NY, USA, 2003. ACM Press.

[10] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[11] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[12] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.

[13] C. Perrow. *Normal Accidents*. Princeton University Press, September 1999.

[14] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[15] T. C. Ruys. Optimal scheduling using branch and bound with spin 4.0. In T. Ball and S. K. Rajamani, editors, *Model Checking Software, Proceedings of the 10th International SPIN Workshop*, volume 2648 of *Lecture notes in Computer Science*, pages 1–17, Berlin, 2003. Springer Verlag.

[16] J. Schumann and W. Visser. Autonomy Software: V&V Challenges and Characteristics. In *In Proceedings of the 2006 IEEE Aerospace Conference*, 2006.

[17] A. Stoica, D. Keymeulen, A. Csaszar, Q. Gan, T. Hidalgo, J. Moore, J. Newton, S. Sandoval, and J. Xu. Humanoids for lunar and planetary surface operations. In *In Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, October 2005.

[18] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[19] B. Stroustrup. Abstraction and the c++ machine model. In Z. Wu, C. Chen, M. Guo, and J. Bu, editors, *ICESS*, volume 3605 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2004.

[20] R. Volpe. Rover Technology Development and Mission Infusion Beyond Mars Exploration Rover. In *IEEE Aerospace Conference*, March 2005.

[21] D. Wagner. Data Management in the Mission Data System. In *In Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.

View publication stats

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:139–156
Published online 15 September 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.686

Fast dynamic casting

Michael Gibbs[1,*,†] and Bjarne Stroustrup[2]

[1]*Lockheed Martin Aerospace, MZ 6602, P.O. Box 748, Fort Worth, TX 76101, U.S.A.*
[2]*Department of Computer Science, TAMU 3112, Texas A&M University, College Station, TX 77843, U.S.A.*

SUMMARY

We describe a scheme for implementing dynamic casts suitable for systems where the performance and predictability of performance is essential. A dynamic cast from a base class to a derived class in an object-oriented language can be performed quickly by having the linker assign an integer type ID to each class. A simple integer arithmetic operation verifies whether the cast is legal at run time. The type ID scheme presented uses the modulo function to check that one class derives from another. A 64-bit type ID is sufficient to handle class hierarchies of large size at least nine levels of derivation deep. We also discuss the pointer adjustments required for a C++ dynamic_cast. All examples will be drawn from the C++ language. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: dynamic casting; C++; modulo; embedded systems; hard real-time

INTRODUCTION

When writing C++, we often convert a pointer to a derived class (subclass) to a pointer to one of its base classes (superclasses). This is the basis for the object-oriented programming style where we use the interface provided by a base class to access the implementation in the derived class. Occasionally we want to cast such a pointer to a base class back to a pointer to its derived class. C++ supports this functionality with the **dynamic_cast** type conversion operation. Consider

```
Source* ptr = new Actual();
Target* t = dynamic_cast<Target*>(ptr);
```

We create an object of type **Actual**, assign its pointer to **ptr** (where **Source** must be a base of **Actual**), and attempt to cast pointer **ptr** to a pointer to **Target**. If the **Actual** object contains a single public base of class **Target**, the offset to the beginning of the **Target** object is computed and a pointer to the **Target** object returned. If the class **Actual** does not inherit from **Target**, the cast fails and returns **0**. If the **Actual** object contains more than a single base of type **Target** (due to multiple non-virtual derivation),

*Correspondence to: Michael Gibbs, Lockheed Martin Aerospace, MZ 6602, P.O. Box 748, Fort Worth, TX 76101, U.S.A.
†E-mail: james.m.gibbs@lmco.com

Copyright © 2005 John Wiley & Sons, Ltd.

Received 22 March 2004
Revised 17 December 2004
Accepted 11 March 2005

Figure 1. Down-casting and cross-casting in C++.

the cast is ambiguous and returns **0**. The exact specification can be found in section 5.2.7 of the ISO C++ standard [1]. A more tutorial explanation can be found in section 15.4 of [2]. A brief discussion of performance and predictability of **dynamic_cast** can be found in sections 3.2.8 and 4.2.2.1 of [3]. Figure 1 shows the common cases of down-casting and cross-casting.

Dynamic casting is useful in many applications, but current implementations of this functionality are slow compared with other C++ operations. Current implementations of **dynamic_cast** require extra type information to be kept for each class with a virtual function. When **dynamic_cast** is called, the algorithm will traverse a data structure representing all of **Actual**'s base classes, searching for a match for the **Target** type. If the type is not found or is found more than once, **dynamic_cast<Target*>(ptr)** returns **0**.

A FAST, PREDICTABLE ALTERNATIVE

Unfortunately, conventional implementations of **dynamic_cast** render it unsuitable for real-time applications where speed, small memory footprint, and predictable performance are essential. We present an alternative implementation strategy designed to meet those constraints. Our solution is primarily aimed at embedded systems where whole program analysis is typically feasible, but (with some additional effort) it can be extended to systems with dynamic linking. Please see the section entitled 'Dynamically loaded classes' for details.

An alternative to walking the inheritance tree is to assign an integer type ID at link time to each class, most likely kept in the virtual table. This ID needs to be a static constant associated with the class. The linker will assign these integers to the classes such that the following property holds: there is a function **F** which takes two integer arguments and returns an integer such that **F(derived_type, base_type) == 0** if and only if the class represented by the ID **derived_type** is a derived class of the class represented by the ID **base_type**.

Copyright © 2005 John Wiley & Sons, Ltd.

The trick lies in finding a function **F** that satisfies this criterion and in determining the size of the integer necessary to accommodate realistic software projects. We would prefer 32- or 64-bit integers so that the function **F** could be performed efficiently. If the evaluation of the function takes longer than walking the derivation tree, then nothing has been gained except a few bytes of program space.

Our suggestion for an **F** function is integer modulo. To construct the type ID, a different prime number is associated with each class. The type ID for that class will be that prime number times the prime number for each of its base classes. Thus, the type ID for a class will be evenly divisible by the type ID of any of its base classes, and only by its base classes. This is guaranteed by the uniqueness of the factorization of integers into prime factors. Given an arbitrarily large integer type ID, any hierarchy can be represented this way. We wish to determine how large a hierarchy can be represented using a machine's built-in integer types.

This method has the advantage that integer math is very fast for numbers that fit in a machine register and should be much faster than the usual tree-walking routine. Importantly, the check that one class derives from another is performed in a known, constant time. If the full type of the object being cast contains more than one copy of the target class, the cast will be ambiguous. In this case, the ID for the full type will be divisible by the square of the prime for the target class. Performing a second divisibility check determines if the cast can be done unambiguously. If the cast can be performed unambiguously, the address of the returned object must be computed, as described below.

There are a number of heuristic methods for keeping the size of the type ID to a minimum number of bits. To prevent the type IDs from being any larger than necessary, the classes are sorted according to priority. The priority for a class is the maximum number of ancestors that any of its descendants has. The priority reflects the number of prime factors that are multiplied together to form a class' type ID. Classes with the highest priority are assigned the smallest prime numbers.

If we had to assign each class a unique prime number, the type IDs would quickly get very large. However, this is not strictly necessary. While all classes at the root level (those having no base classes) must be assigned globally unique prime numbers, independent hierarchies can use the same primes for non-root classes without conflict. Two classes with a common descendant then cannot have the same prime and none of their children may have the same prime. This also implies that no two children of a class may have the same prime. In all other cases, the primes can be duplicated across a level of the hierarchy. For example, in a tree structure two classes on the same level of the tree never have a common descendant, so they may have identical sub-trees beneath them without a conflict.

CONFLICTING CLASSES

Figure 2 shows an error that an overly simple assignment of prime multipliers could cause. The proposed scheme avoids this problem. Note that the type ID for class **D**, 10, divides the type ID for class **E**, 210, even though class **E** does not derive from class **D**. This has happened because we assigned the same prime multiplier, 2, to classes **C** and **D**, thinking they were independent. They are not independent because classes **A** and **B** share a common descendant, class **E**. In our terminology, classes which share a common descendant are said to be in the same *group*. Class **A** and class **B** are in the same group. This motivates the rule for assigning primes: if any parent of class **F** is in the same group as any parent of class **G**, then class **F** and class **G** must be assigned different prime multipliers. This rule is probably too strict, but we do not see any obvious way to loosen it.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 2. Justification for the conflict rule.

The classes can be divided into levels and each level assigned a distinct set of prime numbers. The level of a class must be greater than the level of any of its base classes and a class with no base classes (i.e. a root class) has a level of 0. One way to assign level numbers is to make the level the maximum number of upward links that need to be traversed to reach a root node. For tree-like inheritance trees, the level number is very intuitive.

EXAMPLES

Figure 3 shows a simple hierarchy of Vehicle classes. (This hierarchy is for illustrative purposes only. This is not intended to be an actual categorization of all possible vehicle types.) In this example, all classes descend from **Vehicle**, so the **Vehicle** class may be assigned a type ID of 1. The type ID for **Jet** is 663 which can be uniquely factored into primes as $3 \times 13 \times 17$. Therefore, **Air** (type ID 3) and **FixedWing** (type ID $39 = 3 \times 13$) are base classes for **Jet** since 663 is evenly divisible by 3 and by 39. As another example, suppose we have a **Vehicle** pointer that actually points to an instance of **Motorcycle**. When we attempt to perform a **dynamic_cast** to a pointer to **Wheeled**, the cast succeeds because the type ID for the **Motorcycle** is 1771 which is evenly divisible by the type ID for **Wheeled**, 77. If we attempt to perform a **dynamic_cast** to **Ship**, the call would return a 0 because 1771 is not evenly divisible by 55.

```
Vehicle*   veh = new Motorcycle();
Wheeled*   whl = dynamic_cast<Wheeled*>(veh); // Succeeds
Ship*      ship = dynamic_cast<Ship*>(veh); // Fails
```

Also note how the prime factors are re-used in different parts of this hierarchy. For example, **RotaryWing**, **Ship**, and **Wheeled** all are represented by the prime factor, 11. Their type IDs are distinguished by having different base classes. Since **Air** is represented by 3 and **Water** by 5, as long as we never use the factors 3 and 5 anywhere below level 1 in the tree, we can never confuse an air vehicle with a water vehicle.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 3. Vehicle hierarchy with prime assignments and type IDs.

Now let us explore some larger structures to determine how large a hierarchy of classes can be represented in a 32-bit or a 64-bit type ID. Here are some examples of type ID assignment for tree structures.

A binary tree 7 levels deep (255 classes) has a maximum type ID of $2 \times 5 \times 11 \times 17 \times 23 \times 31 \times 41 \times 47 = 2\,569\,288\,370$, which will fit in a 32-bit word. The value 1 is not assigned to the root node to account for the possibility that there may be other classes that do not derive from the root node.

Level 0 (1 node) is assigned the prime 2.

Level 1 (2 nodes) is assigned 3 and 5.

Level 2 (4 nodes) is assigned 7 and 11.

Level 3 (8 nodes) is assigned 13 and 17.

Level 4 (16 nodes) is assigned 19 and 23.

Level 5 (32 nodes) is assigned 29 and 31.

Level 6 (64 nodes) is assigned 37 and 41.

Level 7 (128 nodes) is assigned 43 and 47.

Figure 4 is a diagram of levels 0 through 4 of the binary tree, showing the assigned primes above the type IDs for each class. A binary tree 12 levels deep (8191 classes) has a maximum type ID of 5.956×10^{18}, which will fit in a 64-bit integer. The maximum type ID for a binary tree of 21 levels will fit in a 128-bit integer and 37 levels require a 256-bit integer (see Figure 5 for chart).

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 4. Type IDs in a binary tree.

A trinary tree six levels deep (1093 classes) has a maximum type ID of $2 \times 7 \times 17 \times 29 \times 41 \times 53 \times 67 = 1\,004\,869\,082$, which will fit in a 32-bit integer. A trinary tree 11 levels deep (265 720 classes) has a maximum type ID of 1.452×10^{19}, which will fit in a 64-bit integer. The maximum type ID for a trinary tree of 19 levels will fit in a 128-bit integer and 34 levels require a 256-bit integer.

The worst-case hierarchy is one in which a class has a large number of immediate base classes. The curve marked 'Unary' in Figure 5 shows the number of bits required for a given number of base classes. In particular, a class with nine base classes requires a 32-bit integer, 15 base classes require a 64-bit integer, 26 base classes require a 128-bit integer, and 43 base classes require a 256-bit integer.

A tree nine levels deep with branching ratios (100, 100, 100, 10, 6, 5, 4, 3, 2), which has over 12 billion classes, has a maximum type ID of $1 \times 2213 \times 1451 \times 733 \times 113 \times 71 \times 43 \times 23 \times 11 \times 3 = 6.163 \times 10^{17} = 2^{59.096}$, which will fit in a 64-bit word (see Figure 6). This example is inspired by the Visual Component Library (VCL) hierarchy of Borland C++ Builder 6, which has similar branching ratios, but is far from a full tree. From these examples, the primary limiting factor seems to be the depth of the tree, though there is a tradeoff between tree width and tree depth. For the nine-level example (Figure 6), if the trees underneath a 100 branching factor are not all full (as is highly likely), then a significant reduction in maximum type ID may be made by assigning the smallest primes to those branches that have trees underneath them and the larger primes to the branches that terminate early.

A strategy where classes bid on the prime values according to some pre-computed priority is effective in assigning small primes to those classes that have deep trees underneath them.

The type IDs get large when a class has a large number of classes from which it derives. This can either be due to the depth of the tree or from multiple inheritance. The proposed scheme will generate type IDs larger than 64 bits if there are classes with large numbers of base classes (usually more than 10), though a class can have up to 15 base classes if there is only one class with that many base classes.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 5. Depth of tree hierarchy versus number of bits in type ID.

VIRTUAL INHERITANCE AND AMBIGUITIES

Virtual inheritance (also known as repeated inheritance in some languages) complicates the assignment of IDs to classes, but the scheme still works with appropriate modification. Every class has three possible types of ancestor classes: ambiguous, virtual, and unambiguous non-virtual. A base class is ambiguous if more than one copy of it exists in the derived class. A base class is virtual if it is declared virtual in the inheritance tree. A virtual class is included only once in the derived class. Classes which appear exactly once in the base class and are not declared virtual are unambiguous non-virtual. In the previous discussion we have considered only cases with unambiguous non-virtual base classes. Here is an example of a more complex class:

```
class A { . . . };
class V { . . . };
class C : public A, public virtual V { . . . };
class D : public C { . . . };
class E : public C { . . . };
class F : public D, public E { . . . };
```

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 6. A full nine-level tree of classes with branch ratios (100, 100, 100, 10, 6, 5, 4, 3, 2).

In this more complicated case, for each class we must keep track of the type of each of its ancestor classes. If each class **C** has been assigned a prime P_C, the ID for class **D** is the product of P_D, the primes of each of its virtual and its unambiguous ancestors and the square of the primes of its ambiguous ancestors. The type IDs for the above example are shown in Figure 7. The **Value** field listed for each class will be explained in the 'Computing the offset' section.

Given an object of type **F** and a pointer to one of the **A** objects in the **F**, we can cast the **A** pointer to a pointer to the **C** of which the **A** is a part. We cannot cast a pointer to **V** to a pointer to **C** because we would not know which **C** we should choose. In addition, we can cast from either **A** to **D**, **E**, or **F**.

The ID for each class is divisible by the IDs of all classes from which it inherits. When attempting to cast an **F** to a **C**, we first verify that $ID_F \bmod ID_C = 180\,180 \bmod 78 = 0$. This means that **F** contains at least one **C**. Then we check $ID_F \bmod P_C^2 = 180\,180 \bmod 9 = 0$ which means that the cast is ambiguous. Thus a cast from **F** to **C** should return 0.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 7. ID computation with virtual inheritance.

COMPUTING THE OFFSET

In addition to determining if the actual class derives from the target class, we must also compute the return value of the dynamic cast, which is a pointer to an object of the target type. Our initial code example is repeated for reference:

```
Source* ptr = new Actual();
Target* t = dynamic_cast<Target*>(ptr);
```

There are three steps in performing a dynamic cast.

1. Finding the type and address of the **Actual** object given the **Source** pointer (**ptr**).
2. Determining if the **Actual** object derives unambiguously from the **Target** class. If it does not, null is immediately returned.
3. Computing the offset of the **Target** class within the **Actual** class to return a pointer to a **Target** object.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 8. Layout of class **F** showing virtual tables.

Each step exists in every ISO Standard C++ implementation to support virtual function calls and **dynamic_cast**. We will outline a traditional scheme and then describe how to optimize all or parts of steps 2 and 3.

Consider the complex object in Figure 7. There are several ways to lay out such an object, and several different ones are in use. What we describe here is a scheme that is a bit simpler and faster than most because it does not try to match the C++ Application Binary Interfaces (ABIs). Figure 8 shows a plausible layout for an **F** object and its basic structures supporting virtual function calls.

Each **A** and **V** have a separate **vtbl** ('virtual function table'). **F** and **D** and **D::C** share **F::D::C::A**'s **vtbl**. Similarly, **E** and **E::C** share **E::C::A**'s **vtbl**. Such sharing is an almost universal optimization. Each **vtbl** has a pointer to the start of **F** (their joint 'most derived class'). Given a pointer to any base class object, we can find a **vtbl**, and through that **vtbl** find the start of the **F** object.

Step 1 of the implementation of **dynamic_cast** (finding the complete object—as it was created—here **F**) is easy and fast in any implementation. Here, we simply follow a pointer stored in each **vtbl**.

In current implementations, steps 2 and 3 are performed by traversing (base, offset) tables accessed through the **vtbl** until we find an entry where both base and offset match the pointer with which we started (the operand to **dynamic_cast**). Because of the possibility of an ambiguity, we have to search further than the first match.

Copyright © 2005 John Wiley & Sons, Ltd.

The proposed scheme will perform step 2 (checking if the dynamic cast will succeed) with a check that the ID of the actual class is evenly divisible by the ID of the target class. If it is not, null is returned. If it is divisible, it checks if the actual ID is divisible by the square of the target prime. If it is divisible, the cast is ambiguous and null is returned. If it passes both these checks the cast can be performed unambiguously. Clearly, this will typically be faster than any search through a list of base classes. In particular, the modulo test will not need to access memory repeatedly. This optimizes the performance of an unsuccessful cast, making that operation constant time.

It is not uncommon for **dynamic_cast** to be used to compare an object against several possible classes. In such uses, the cast will fail repeatedly. The proposed scheme catches this failure with a single modulo computation, resulting in a significant decrease in the effort to locate the correct class type; we only search (base, offset) tables when we know we will find a base.

However, as long as we do any form of search, the total cost of **dynamic_cast** cannot be constant. Without further improvement our implementation has the cost of a failed **dynamic_cast** constant and successful **dynamic_cast** has a worst-case cost linear with the number of classes in the most derived object (using the same techniques as existing implementations). We can do better.

The complexity of finding the offset depends on the complexity of the class. We can distinguish three cases:

1. single inheritance only;
2. multiple inheritance without any base being replicated;
3. multiple inheritance with one or more bases replicated.

A compiler can trivially distinguish between these three cases and generate different—and optimal—code for each.

The first case (single inheritance) is interesting because no offset modification is needed, so that the problem is solved with close-to-zero added cost in time or space. All that is needed is an indicator in the **vtbl** that no offset calculation is needed—all sub-objects share a single pointer to the ultimate base. Basically, the data structure needed is what we showed in Figure 8 and a **dynamic_cast** is done in constant time, and very fast.

The second case (no base appearing twice) has the property that we can provide a simple (base, offset) map. The simplest implementation would just make that map a vector and search it linearly. Since there is no need for an ambiguity check, the cost will be a maximum of $N + 1$ comparisons, where N is the number of bases in the object, and the average cost $(N + 1)/2$. The $+1$ is to count the most derived object itself. Better performance may be achieved by sorting the vector and using a binary search.

Once the base is found, a simple addition to the pointer to the complete object gives the desired pointer result. The (base, object) map would be attached to or be part of the **vtbl** for the complete object. Note that the offsets can all be relative to the start of the complete object and applied to the pointer to the complete object independently of which base class was pointed to by the **dynamic_cast** argument.

The third case (one or more bases appearing twice) is more complicated. For example, consider trying to obtain **dynamic_cast<C*>(pa)**. If **pa** points to the first **A**, we should get a pointer to the first **C** and if **pa** points to the second **A** we should get a pointer to the second **C**. To do that, we could use a single table with complicated offsets that we can search with the offset of **pa** in the complete object. A simpler approach is to use a table per **vtbl** (Figure 9). In this case, we search the relative offset table

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 9. Layout of class **F** showing virtual tables and object map.

first to see if the target class is derived from our base by single non-virtual derivation. If so, we use the relative offset found; otherwise, we search the absolute offset table of the bases that are unambiguously accessible from the most derived class.

Case 1 is unbeatably cheap and constant time. Case 2 is relatively inexpensive and has an easy-to-calculate upper bound that may be as good as constant time. Case 3 is pretty ugly; it does have an upper bound, but that bound is not trivial for humans to estimate and current implementations show that a **dynamic_cast** implemented roughly this way has a cost comparable with several function calls (about 10 for examples like that of Figure 7).

Even so, the modulo technique provides a significant improvement over traditional techniques: a common use of **dynamic_cast** is to compare an object against several possible class types. In this usage, the cast will fail in most cases. The proposed scheme catches this failure with a single modulo computation, resulting in a significant decrease in the effort to locate the correct class type.

Copyright © 2005 John Wiley & Sons, Ltd.

Also, the knowledge that the cast will succeed if we get to step 3 (determine the offset) simplifies that computation.

When we are not satisfied by a worst case estimate of the cost, cannot afford the elaborate mechanism of case 3, or need an absolute constant-time guarantee, we can optimize by using the modulo function again. We use the Chinese remainder algorithm to compute our offsets. We put all the offsets into an indexed table and assign each class another integer value **V** such that V_D mod P_B gives the index of the offset of **B** within **D** in the offset table. We will illustrate this technique using the hierarchy in Figure 7.

For example purposes, we will assume that **A** is 8 bytes and **V** is 12 bytes. **C** has 16 bytes over and above what it inherits from **A** and **V**, making its total size 36 bytes. **D** has 20 bytes over and above what it inherits from **A**, **C**, and **V**, making its total size 56 bytes. **E** has 32 bytes over and above what it inherits from **A**, **C**, and **V**, making its total size 68 bytes. **F** has 40 bytes over and above what it inherits from **A**, **C**, **D**, **E**, and **V**, making its total size 140 bytes.

Offset of **A** in **C**: 0
Offset of **V** in **C**: 24
Offset of **A** in **D**: 0
Offset of **C** in **D**: 8
Offset of **V** in **D**: 44
Offset of **A** in **E**: 0
Offset of **C** in **E**: 8
Offset of **V** in **E**: 56
Offset of **D** in **F**: 0
Offset of **E** in **F**: 44
Offset of **V** in **F**: 140

Create a table of unique offset values. The offset zero occurs frequently and should be put at index zero:

Index	0	1	2	3	4	5
Offset	0	8	24	44	56	140

Assign primes and values to each class at link time using the Chinese remainder algorithm (these values appear in Figure 7):

Class	**A**	**C**	**D**	**E**	**F**	**V**
Prime	2	3	5	7	11	13
Value	0	2	16	4	395	0

For example, for **F** we need V_F mod $5 = 0$ since **D** has offset 0 in **F** and we need V_F mod $7 = 3$ since **E** has offset 44 in **F**. Note that offset 44 appears at index 3 in the offset table. We also need V_F mod $13 = 5$ since **V** has offset 140 in **F**. We can see that $V_F = 395$ satisfies these constraints.

Copyright © 2005 John Wiley & Sons, Ltd.

Performing a dynamic cast from an object of type **F** to an **E**, first we verify that the cast is legal (it is), then we compute the offset. $V_F \bmod P_E = 395 \bmod 7 = 3$. The offset found at index 3 in the offset table is 44, so **E** is to be found 44 bytes from the beginning of **F**.

The Chinese remainder theorem guarantees that we can find a number that satisfies all the necessary modulo constraints provided all the base classes of a class are given unique prime values. We already have the constraint that no two classes with a common descendant may be assigned the same prime, so we could re-use the primes from the validity check in the offset computation as well. The prime chosen for a class must be larger than any offset index it produces for a descendant class, which provides another constraint on the prime that may be assigned to a class.

We now have a system where each class has three additional integers attached to its virtual table and dynamic casting may be performed in a constant, known time. This modulo method for finding offsets is not as fast as the conventional method for the single inheritance case, but should equal the conventional method for non-repeated bases (at least for classes with many bases) and clearly beat the conventional method in the general case with repeated classes. The optimal combination of techniques will depend on the mix of classes used and the relative cost of machine operations (especially memory access). However, our desire for fast, constant time (or firm upper bound) performance is clearly met.

DYNAMICALLY LOADED CLASSES

The main aim of this paper is to demonstrate a way of implementing **dynamic_cast** that is suitable for embedded systems programming so we will not develop a dynamic linking extension here. However, such an extension appears to be feasible; whether it is practical in real-world systems is another question (which we do not propose to answer here).

The scheme for assigning type IDs assumes the entire hierarchy of classes is available at the time the type IDs are assigned. Creating type IDs for dynamically loaded classes might be done as follows.

A code fragment (CF), that is, a shared library (dynamically linked library in Windows) or a main program, contains all the information it needs to resolve any dynamic cast involving classes it defines. The hierarchy above any new class brought in by the CF is known when the code is compiled. To support dynamic casting, each CF could assign type IDs independently of all others. In order to perform a dynamic cast, the most derived type of the object being cast is determined, and the CF in which it was defined. Within that CF, the type ID of the class to which we are casting is determined. If the class is not known to that CF, then the cast fails and returns null. Otherwise, the modulo check is performed as usual.

This scheme implies that a class will have one type ID for each CF that uses it. It is necessary that each class knows in which code fragment it was defined. The type information for a class will effectively consist of a map from CF ID to type IDs. This could be implemented in several ways.

PRIORITIZING THE CLASSES

We have shown that a 64-bit type ID is sufficient for several large systems; we would like to concentrate on heuristics to keep the type IDs as small as possible, so that as large a hierarchy as possible can be represented with an ID of this size.

Copyright © 2005 John Wiley & Sons, Ltd.

The following terminology will be used in the discussion below:

parent—a direct base class;
child—a direct derived class;
ancestor—a direct or indirect base class;
descendant—a direct or indirect derived class;
sibling—two classes are siblings if they have at least one common parent;
root—a class having no parents;
leaf—a class having no children;
level—the level of a class is the maximum number of links one must follow up the hierarchy from it to reach a root node; root nodes are at level zero;
group—a set of classes on the same level having a common descendant.

Much smaller maximum type IDs will be obtained if care is taken to give smaller prime values to classes that have more levels underneath them. This can be done by assigning a priority to each class, giving classes with higher priorities smaller prime values. The priority value should reflect the expected size of the largest type ID of any of a class's descendants. The largest type ID values always occur at the leaves of the inheritance tree, so once the expected maximum value for each leaf has been computed, the values can be propagated upward through the tree to set the priority values. Here are four possible schemes for computing priority.

1. The priority of each leaf class is the number of ancestor classes it has. This is the primary method used in our test program.
2, 3, 4. Assign a range of primes to each level, with wider levels receiving larger starting values. Each node is given a value proportional to the logarithm of the (minimum, mean, or maximum) prime in its level. The priority for a leaf class is the sum of these values for itself and all its ancestor classes.

In all cases, the priority of a non-leaf class is the maximum of the priorities of all its derived classes.

Here is how to use the priorities to assign prime values: sort all classes by priority, highest priority first. Assign the highest priority class the smallest prime in the set of primes for that level, the next highest priority class the second smallest prime, and so on. We can assign the same prime to two or more classes on a level if none of their parents have a common descendant.

OPTIMIZATIONS

There are several optimizations which will make the numbers smaller, hence able to represent larger class diagrams.

- A class which is the only child of each of its parents can be assigned the prime value 2 if it has only one parent or the value 1 if it has more than one parent. This assumes that the prime 2 is reserved exclusively for this purpose.
- If there is a single root class in the system, from which all other classes derive, it can be assigned a 'prime' value of 1, and thus a type ID of 1.
- Disjoint hierarchies can be broken into levels and assigned type IDs independently of one another, provided that all root classes have unique prime type IDs.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 10. Simple optimization.

- For unbalanced trees, it may be beneficial to give levels non-consecutive sets of primes. For example, if a level has a width of six, but at most three of the siblings in any family have deep trees underneath them, then the level could take three small primes and three large primes. This can be implemented by having levels bid on the primes according to the priority of their classes.

The first two of these optimizations are easy to implement. For cases where there is a single derived class of another class, an 'only child', we can extend the hierarchy quite deep using these techniques. These two techniques are incompatible with the Chinese remainder offset scheme and can break if another class is added to the hierarchy. Breaking the hierarchy into disjoint pieces should be unnecessary if the last optimization (bidding) is used.

Figure 10 illustrates these first two optimizations. A derived class must always have a type ID larger than any of its base classes so that the algorithm never identifies the base class as deriving from the derived class. However, we require only that the derived class is divisible by all its base classes and distinguishable from all its siblings.

The bidding strategy is quite helpful for unbalanced trees and can be implemented as follows. Each level needs a set of prime numbers assigned to it. First, every class is assigned a priority by method 1 as described above. The classes are sorted by priority, highest priority first. For classes with equal priority, those with smaller level number come first. The classes are examined one by one, in order of priority. The class is compared to all other classes on its level that have already been assigned primes, checking for conflicts. The smallest available (non-conflicting) prime in the level is assigned to the class, adding a new prime to the level if necessary. The primes are thus dealt out to the levels starting with 3, 5, 7, 11, etc. The value 2 is reserved for only-child classes.

Copyright © 2005 John Wiley & Sons, Ltd.

Figure 11. For hierarchies with 8192 classes, the number of ancestors of the class with the largest ID versus the number of bits in the largest ID.

SIMULATION

We wrote a simulation that randomly generates a class hierarchy with specified statistics, then assigns type IDs and reports the largest type ID used. We used the bidding strategy with the optimizations discussed above. It ran in under a second for 1024 classes and in about 18 s for 8192 classes on a 533 MHz Pentium IV PC. We suspect that most of this time is due to the clustering algorithm and could be optimized substantially.

We ran the simulation for various numbers of classes and various statistics. The results are shown in Figure 11. On average, for a hierarchy with 8192 classes, the type ID of a class with 11 ancestors will just fit in a 64-bit integer (see chart in Figure 5). A class with 26 ancestors is expected to require a 128-bit ID. The more convoluted class hierarchies required more bits to represent them, as expected.

Copyright © 2005 John Wiley & Sons, Ltd.

NOTES AND FUTURE DIRECTIONS

Only classes that have their type ID explicitly accessed or are of a class that is the source or target of a dynamic cast need to have a type ID computed. This may be a small fraction of the total number of classes in the system.

The size of the integer needed could be dynamically changed by the linker according to the size of the problem, or a variable-sized integer class could be used for the type IDs.

It may be possible to devise other strategies for assigning type ID integers, such as a hashing function that returns 0 when the two integer type IDs satisfy a particular pattern. This may allow the system to cover greater numbers of classes than is possible with the modulo function. A hashing system would have some probability of generating a false positive, believing that one class was derived from another when it was not. The linker would have to verify that this did not happen for the particular set of classes involved and start over with a different hashing function if the first did not work.

SUMMARY

We have demonstrated that it is possible for a linker to generate integer type IDs for classes such that it may be verified by a simple integer modulo computation that one class derives from another in an object-oriented language. When combined with a suitable way of adjusting offsets, this method provides for a fast, constant-time dynamic casting algorithm. A 64-bit type ID is capable of representing quite large class hierarchies containing thousands of classes and at least nine levels deep.

ACKNOWLEDGEMENTS

We would like to thank Dr Gabriel Dos Reis, who made several constructive comments on earlier drafts of this paper.

REFERENCES

1. Koenig A (ed.). *The C++ Standard* (2nd edn). IOC/IEC 14882:2003. Wiley: New York, 2004.
2. Stroustrup B. *The C++ Programming Language*. Addison-Wesley: Reading, MA, 2000.
3. Goldthwaite L (ed.). *Technical Report on C++ Performance*. ISO/IEC PDTR 18015, 2003.

Copyright © 2005 John Wiley & Sons, Ltd.

Model-based Product-Oriented Certification

Damian Dechev
dechev@tamu.edu
Texas A&M University
College Station, TX 77843-3112

Bjarne Stroustrup
bs@cs.tamu.edu
Texas A&M University
College Station, TX 77843-3112

Abstract

Future space missions such as the Mars Science Laboratory and Project Constellation suggest the engineering of some of the most complex man-rated software systems. The present process-oriented certification methodologies employed by NASA are becoming prohibitively expensive when applied to systems of such complexity. The process of software certification establishes the level of confidence in a software system in the context of its functional and safety requirements. Providing such certification evidence may require the application of a number of software development, analysis, and validation techniques. We define product-oriented certification as the process of measuring the system's reliability and efficiency based on the analysis of its design (expressed in models) and implementation (expressed in source code). In this work we introduce a framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis. We describe a classification of the certification artifact types, the development and validation tools and techniques, the application domain-specific factors, and the levels of abstraction. We demonstrate the application of our certification platform by analyzing the process of model-based development of the parallel autonomic goals network, a critical component of the Jet Propulsion Laboratory's Mission Data System (MDS). We describe how we identify and satisfy seven critical certification artifacts in the process of model-driven development and validation of the MDS goal network. In the analysis of this process, we establish the relationship among the seven certification artifacts, the applied development and validation techniques and tools, and the level of abstraction of system design and development.

1 Introduction

[1] In this work we introduce a framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis by the utilization of advanced programming techniques and tools. As opposed to process-oriented certification (as suggested by DO-178B [24]), the product-oriented methodology[7] relies on the application of safety concerns directly on implementation source code and its formal models. The product-oriented approach is inherently more flexible by offering the developers the freedom to follow a variety of software development life-cycle paradigms. In addition, the certification authority itself has the ability to collect all required artifacts for the system's safety and quality assurance. As suggested in [29], the rationale for source code enhancement is to seek an effective alternative to domain-specific programming languages for high-performance computing systems. A language enhancement can be achieved by supersetting a programming language by a library defining domain-specific concepts and algorithms and at the same time employing program analysis and validation tools to ensure the correctness of the introduced domain-specific notions. Source code enhancement allows a programmer to reach a high level of expressiveness and at the same time rely on the tool-chain of a mainstream programming language. We demonstrate the use of our certification platform by describing its application to the process of model-based development and verification of the autonomic goals network [6], a critical component of the Jet Propulsion Laboratory's Mission Data System (MDS). MDS provides an experimental goal- and state- based platform for model-driven testing and development of autonomous real-time flight applications [21]. The process of software certification establishes the level of confidence in a software system in the context of its *functional* and *safety* requirements. A software certificate contains the evidence required for the system's independent assessment by an authority having minimal knowledge and trust in the technology and tools employed [7]. Providing such certification evidence may require the application of a number of software development, analysis, verification, and validation techniques [18]. Our approach offers a classification of and establishes the relationship among the certification artifact types, the development and validation tools and techniques, the application domain-specific factors, and the levels of abstraction. We further suggest the concept of semantic enhancement of the source code and the application of a number of program analysis and transformation techniques to achieve reliabil-

[1] This is the authors' version of the work. It is posted here by permission of the publisher. Not for redistribution. The definitive version is published in Proceedings of 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS 2009), San Francisco, California, April 2009.

ity and efficiency in the system implementation. Our case study in Section 4 illustrates the practical application of our approach. We describe how we identify and satisfy seven critical certification artifacts in the process of model-driven development and validation of the MDS goal network. In the analysis of this process, we establish the relationship among the seven certification artifacts, the applied development and validation techniques and tools, and the level of abstraction of system design and development.

2 Background and Previous Work

Future space exploration projects such as the Mars Science Laboratory (MSL) [31] and Project Constellation [27] suggest the engineering of some of the most complex man-rated software systems. As stated in the Columbia Accident Investigation Board Report [3], the inability to thoroughly apply the required certification protocols had been determined to be a contributing factor to the loss of STS-107, Space Shuttle Columbia. Schumann and Visser's discussion in [25] suggests that the current process-oriented certification methodologies are prohibitively expensive for systems of such complexity. A detailed analysis by Lowry [18] indicates that at the present moment the certification cost of mission-critical space software exceeds its development cost. The challenges of certifying and re-certifying avionics software has led NASA to initiate a number of advanced experimental software development and testing platforms, such as the Mission Data System (MDS) [21], as well as a number of program synthesis, modeling, analysis, and verification platforms, such as The JavaPathFinder [2], the CLARAty project [30], Project Golden Gate [10], The New Millenium Architecture Prototype (NewMAAP) [9].

2.1 Complex Systems Analysis

In [20] Perrow studies the risk factors in the modern high technology systems. His work identifies two significant sources of complexity in modern systems: *interactions* and *coupling*. The systems most prone to accidents are those with *complex* interactions and *tight* coupling. With the increase of the size of a system, the number of functions it has to serve, as well as its interdependence with other systems, its interactions become more incomprehensible to human and machine analysis and this can cause unexpected and anomalous behavior. Tight coupling [20] is defined by the presence of time-dependent processes, strict resource constraints, and little or no possible variance in the execution sequence. Perrow classifies space missions in the riskiest category since both hazard factors are present.

2.2 Parallelism and Complexity

In a parallel application, there are a number of challenges that are not known in sequential programming: most importantly to correctly manipulate data where multiple threads access it. The most commonly applied technique for controlling the interactions of the concurrent processes is the application of mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system and diminish its parallelism. For the majority of applications, the problem with locks is one of difficulty of providing correctness more than one of performance. The application of mutually exclusive locks poses significant safety hazards and incurs high complexity in the testing and validation of mission-critical software. Mutual exclusion locks can be optimized in some scenarios by utilizing fine-grained locks [14] or context-switching. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [18]. Even for efficient locks, the interdependence of processes implied by the use of locks, introduces the dangers of deadlock, live-lock, and priority inversion. The incorrect application of locks is hard to determine with the traditional testing procedures and a program can be deployed and used for a long period of time before the flaws cause anomalous behavior.

As discussed by Lowry [18], in July 1997 The Mars Pathfinder mission experienced a number of anomalous system resets that caused an operational delay and loss of scientific data. The follow-up study identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the high-priority bus management process. The investigation furthermore revealed that it would have been impossible to detect the problem with the black box testing applied at the time to derive the certification artifacts. A more appropriate priority inversion inheritance algorithm had been ignored due to its frequency of execution, the real-time requirements imposed, and its high cost incurred on the slower flight-qualified computer hardware. The efficient and reliable control of the subtle interactions in the concurrent applications of the modern aerospace autonomous systems is of critical importance to the overall system's safety and correct operation. Despite the challenges in debugging and verification of the system's concurrent components, the existing certification process [24] does not provide guidelines at the level of detail reaching the development, application, and testing of concurrent programs. This is largely due to the process-oriented nature of the current certification protocols and the complexity and high level of specialization of the aerospace autonomous embedded applications. In the near future, NASA plans to deploy a number of diverse vehicles, habitats, and support-

ing facilities for its imminent missions to the Moon, Mars and beyond. According to [21] some of the most significant challenges for such systems are managing a large number of tightly-coupled components, performing operations in uncertain remote environments, ability to respond and recover from anomalies, guaranteeing the system's correctness and reliability, and the effective communication across the system's components.

3 Principles of Model-based Product-Oriented Certification

We define *product-oriented certification* as the process of establishing the system's reliability and efficiency based on the analysis of its design (expressed in models) and implementation (expressed in source code). In this section we describe a classification of the *certification artifact types*, the *development and validation tools and techniques*, the *application domain-specific factors*, and the *levels of abstraction*. In our framework a *certification artifact type* can be one of the following:

(1) Invariant (η): a critical property or assumption that is constant (does not change throughout the transformation of program/system states) and must hold at all time to ensure the validity and correct operation of a program or a system.*Example a.*: the values stored in a given shared vector must be word-sized pointers. *Example b.*: a graph of temporal constraints [6] must contain no cycles.

(2) Guarantee (γ): a goal or condition that needs to be satisfied. Unlike invariants, goals can be defined differently at different moments of the lifecycle of a system. *Example:* an event E_a must precede an event E_b in the autonomic operation of a robot.

(3) Constraint (κ): a physical and resource constraint that need to be observed. *Example:* the physical memory available to store a graph of autonomic goals is 7168KB.

(4) Performance artifact (ϵ): an artifact describing the quality of operation and degree of optimization. *Example:* Complexity and space efficiency of a particular propagation scheme in a network of temporal constraints.

(5) Comprehension artifact (σ): an artifact measuring the human understanding of the interactions, coupling, and behavior of a system. *Example:* a list of all concurrent interleavings in a goal network leading to a state of inconsistency.

(6) Re-certification and maintenance artifact (μ): an artifact demonstrating the ability to re-establish the validity of the system upon its evolution and re-use. *Example:* an automated program analysis tool that checks for the separation of data and algorithms and thus can demonstrate the validity of the graph implementation upon the replacement of the constraint propagation algorithm.

3.1 Development

Satisfying the certification artifacts in a software system requires the application of a combination of software development, modeling, formal verification, and analysis techniques and tools. Expressing as well as checking the certification requirements is enabled and directly dependent on the following software development dimensions:

(1) Model of Computation (Δ_{MC}): the computing architecture defined by the hardware and the operating system. It defines the sequential or parallel memory model as well as the available basic machine-level instructions and atomic primitives. *Example:* an embedded multi-core platform with eight processing cores supporting only single-word atomic primitives, such as the single-word Compare-And-Swap (CAS).

(2) Programming Language (Δ_{PL}): programming constructs, libraries, and techniques available. *Example:* The availability of a nonblocking vector [5] that can allow safe and lock-free access to shared data (and thus eliminate the hazards of deadlock, livelock, and priority inversion).

(3) Modeling Tools (Δ_{MT}): expressing design notions, automated code generation, and formal verification. *Example:* the application of the SPIN model checker [13] to exhaustively search all concurrent interleavings.

(4) Analysis Techniques (Δ_{AT}): program static and dynamic analysis. *Example:* the application of static analysis utilizing a high-level program representation to guarantee high performance in parallel systems [29].

(5) Software Architecture (Δ_{SA}): defines the most significant design notions such as system states, system goals, and modes of communication. *Example:* The Mission Data System defines a unified model-driven architecture for testing and development of autonomous flight software based on the notions of system goals and states.

3.2 Application Domains

The application-domain factors have a direct impact in defining the certification requirements and the development process. We identify the following significant application-specific properties for mission critical software:

(1) Real-time (Rt): the system must achieve a goal or provide a response in a time-constrained manner. *Example:* The real-time operation of a robot demands a *system guarantee* that the meteorological process must complete prior to the initiation of communication with mission control.

(2) Safety-critical (Sc): establishes that a failure would lead to a catastrophic or hazardous consequences to the entire system. The DO-178B offers a hazards analysis process to assess the risk level upon a module or sub-system failure. *Example:* if the autonomous obstacle avoidance scheme fails, the rover

might crash. Thus, the system *invariants* and *guarantees* assuring the correct operation of the system are *safety-critical*.

(3) Embedded (Em): since the system is designed and optimized according to a set of pre-defined goals, its software must often control the hardware, consider strict resource constraints, and handle failures and events that may occur in the physical world. *Example:* the embedded nature and limited memory availability of the rover places the *constraint* that a goal network should not exceed 7168KB of memory space.

(4) Autonomous (Au): the system must achieve a set of goals with little or no human interaction, meanwhile possibly responding to the conditions and events in its environment. *Example:* the autonomy of the meteorological and bus management processes requires the *invariant* that the system is free of the hazards of priority inversion.

3.3 Levels of Abstraction

We classify the system's safety concerns according to their rank in the abstraction hierarchy:

(1) Physical and Hardware (Φ): related to constraints in the hardware resources, organization, and architecture and the conditions in the physical environment. *Example:* the lack of complex atomic primitives on the flight-qualified hardware requires all nonblocking code to rely on the single-word Compare-And-Swap (CAS) atomic primitive. This demands the specification of an *invariant* that the system must eliminate the possibility of occurrence of the ABA problem [5].

(2) Algorithms and Procedures (Θ): invariants of a particular computational routine or algorithm. *Example:* the complexity of Floyd-Warshall's all-pairs-shortest-path algorithm [4] is $O(N^3)$. Due to the frequent execution of the constraint propagation scheme in a goal network the direct application of the algorithm can be prohibitively expensive. To meet the *performance* requirements a propagation scheme should execute with complexity of at most $O(N^2)$.

(3) Libraries (Λ): domain-specific concerns on a set of algorithms that are grouped in a standard or custom language extension. *Example:* a library of CAS-based nonblocking algorithms must guarantee its ABA-freedom.

(4) Modules and Sub-Systems (Ψ): guarantees and quality of service provided by the individual components and sub-systems. *Example:* The rover's module performing atmospheric experiments must coordinate its execution with the bus management and the communication systems. Such a coordination might lead to a number of safety-critical invariants and guarantees (such as no priority inversion).

(5) System (Ω): goals critical for the successful completion of the mission. *Example:* the rover's goal is to autonomously navigate the surface of Mars, perform scientific exploration of the planet's atmosphere and geology, and communicate results back to mission control. Meeting these goals impacts the guarantees defined by all of the robot's sub-systems.

(6) Framework (Ξ): conditions related to the principle organization and design of the software development. *Example:* The Mission Data System defines the notions of states and goals. Their definition and requirements are described (independently from the implementation of a particular mission) in a number of MDS framework papers such as [21].

As emphasized by Stroustrup in [28], the concept of higher-level systems programming is of significant importance to systems of high complexity and size. Higher-level systems programming implies that while low-level efficiency is important, the emphasis is placed towards the design, maintenance, and validation of the larger system. With respect to the system implementation, it is the programming language facilities for data abstraction and representation of domain-specific concerns that directly address this issue. As defined by Stroustrup [28]:

> A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of *concepts* for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is 'close to the machine', so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is 'close to the problem to be solved', so that the concept of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.

The application of C++ in a framework for complex, autonomous, and embedded flight software, such as the Mission Data System, further illustrates and emphasized the significance of the ability of C++ to excel in providing both, instructions 'close to the machine' and facilities that are 'close to the problem to be solved'. Language facilities allowing the definition of high-level design concepts and domain-specific concern are often provided by language libraries. Such libraries enhance the language semantic model by defining notions and guarantees that belong to the problem domain.

Modeling and formal verification tools such as SPIN [13], Alloy Analyzer [15], and Eclipse [26] are used to express and validate high-level domain-specific and design concerns. The challenges associated with the application of modeling and formal verification tools in the development process are:

(1) Bridging the implementation source and the software models.

  (a) from implementation to models: as an abstraction and simplification of the software implementation, a model represents an aspect of the software solution based on a number of assumptions and rules. Defining these assumptions as well as the verification invariants, and es-

tablishing whether the model is trustworthy with respect to the source are some of the most challenging tasks.

(b) from models to implementation: the application of program synthesis techniques such as AutoFilter [8] have been applied successfully in a number or flight applications. However, the certification of the produced software is challenged by the strict FAA requirement of having the program synthesis meet the same certification requirements as the produced flight software.

(2) Limited state space and heavy computational complexity: despite the advanced state space reduction techniques in many modern formal verification tools, the main limitations for their applicability arise from the heavy computational complexity imposed and the state space explosion problem. Program simplification and abstract interpretation techniques are often necessary to reduce the explored state space. However, according to the FAA certification standards, it is required to establish the preservation of the program semantics upon the application of any program transformation and abstract interpretation techniques.

(3) Project Scheduling: the application of formal logic can often be as demanding to the software developers as the engineering of the system implementation itself.

The semantic enhancement of the implementation can allow for the direct validation of some software invariants and guarantees and thus reduce the state space and the computational complexity required in the process of formal verification. In addition, the increased expressiveness and abstraction level of the implementation source can ease the manual or automated transition to and from the software models. Stroustrup and Dos Reis [29] present the notion of *Semantically Enhanced Library Languages(SELLs)*. As defined by the authors, a SELL is a domain-specific language derived from a general-purpose programming language by extending it with libraries defining the concepts and functionalities of the problem domain and then applying an analysis tool to guarantee the higher-level semantic invariants. The main advantages of defining and applying a SELL are founded in the availability of the maintenance, training, and tool chain of the general-purpose language that had served as its base. At the same time, a SELL's main purpose is to deliver a special-purpose language tailored to the ideals and concepts of a specific application domain. The notion of SELL is fundamental for the application of our model-based product-oriented framework for software certification.

The following section presents a case study describing the details of how we extend the semantics of ISO C++ with the libraries defining Temporal Constraint Networks and Shared Containers with Nonblocking Synchronization. Furthermore, we demonstrate how the applied programming and modeling techniques, formal verification, program transformation, and static analysis (in the process of validation and automatic parallelization of goal networks) relate to our classification framework.

4 A Case Study: Automatic Parallelization of Goal Networks in MDS

Mission Data System (MDS) [21] is the Jet Propulsion Laboratory's framework for designing and implementing complete end-to-end data and control autonomous flight systems. The framework focuses on the representation of three main software architecture principles (defining the highest Δ_{SA} level of development in the certification framework):

(1) System control: a state-based control architecture with explicit representation of controllable state [11]

(2) Goal-oriented operation: control intent is expressed by defining a set of goal and a goal network [1]

(3) Layered data management: an integrated data management and transport protocols [32]

In MDS a state variable provides access to the data abstractions representing the physical entities under control over a continuous period of time, spanning from the distant past to the distant future. In other words, a state variable is a programming (Δ_{PL}) and modeling (Δ_{MT}) representation of a set of constraint (κ) certification variables ($S_{\kappa_{sv}}$) expressed in the library level of abstraction (Λ). As explained by Wagner [32], the implementation's intent is to define a goal timeline overlapping or coinciding with the state variables timeline. This means that the implementation must rely on an algorithm (abstraction level Θ) that transforms the engineers intent together with $S_{\kappa_{sv}}$ into a set of invariants $S_{\eta_{sv}}$ and a set of guarantees $S_{\gamma_{sv}}$ and establish the validity and consistency of all $\eta_i \in S_{\eta_{sv}}$ and all $\gamma_i \in S_{\gamma_{sv}}$ so that the system's operations corresponds with its Rt, Sc, Em, and Au behavior.

Computing the invariants (a set of $S_{\eta_{gi}}$) necessary for achieving a goal (any $\gamma_i \in S_{\gamma_{sv}}$) might require the lookup of past states as well as the computation of projected future states. MDS employs the concept of goals to represent control intent. Goals are expressed as a set of temporal constraints (Section 4.1). Each state variable is associated with exactly one state estimator whose function is to collect all available data and compute a projection of the state value and its expected transitions. Control goals are considered to be those that are meant to control external physical states. Knowledge goals are those goals that represent the constraints on the software system regarding a property of a state variable. Not all states are known at all time. The most trivial knowledge goal is the request for a state to be known, thus enabling its estimator. A data state is defined as the information regarding the available state and goal data and its

storage format and location. The MDS framework considers data states an integral part of the control system rather than a part of the system under control. There are dedicated state variables representing the data states. In addition, data states can be controlled through the definition of data goals. A data state might store information such as location, formatting, compression, and transport intent and status of the data. A data state might not be necessary for every state variable. In a simple control system where no telemetry is necessary, the state variable implementation might as well store the information regarding the variable's value history and its extrapolated states.

The representation of the data states and the data management in MDS is implemented in the Data Management Service module [32]. The problem of data management in an embedded control system (often requiring the satisfaction of Em, Rt, Au, and Sc -driven certification requirements at the same time) is one of translating the intent of remote operations into actions and then safely returning the observed information. The system should be robust to the extent of overcoming possible communication loss, hardware failures or a system reboot. The resource constraint of an embedded control systems (its collection of κ variables) dictate that command-oriented control systems typically do not retain information specific to the intent of the observations. In addition, telemetry systems process and transport data in unlabeled packages where the scientific data is often mixed with other data. In this context, Wagner argues that it is of significant importance to address the challenges of providing uniform models for managing the flow of observation and control data. The MDS Data Management Service Library implements a Catalog for organizing the storage of physical observations in terms of storage products. Its functionality is responsible for the remote transport of data products with respect to the behavior of other spacecraft components. According to the current lock-based Catalog design, locks are applied in a complex manner within the inheritance hierarchy that leads to an exponential increase of the verification state space.

To achieve higher reliability (expressed as a set of safety invariants $S_{\eta_{safety}}$) and enhance the performance (measured in terms of speed of execution in the ϵ_{exe} variable), we consider the application of *lock-free synchronization*. As defined by Herlihy [12], a concurrent object is *nonblocking* (lock-free) if it guarantees that *some* process in the system will make progress in a *finite* amount of steps. Nonblocking algorithms do not apply mutually exclusive locks and instead rely on a set of atomic primitives supported by the hardware architecture. This means that a nonblocking technique represents an algorithmic (Θ) or library (Λ) solution to an important Sc problem, namely avoidance of the hazards of deadlock, livelock, and priority inversion (expressed as three separate Sc invariants: η_{lvlock}, η_{delock}, and η_{pinv}) while at the same time offering a significant performance boost (measured in ϵ_{exe}). The application of a library of nonblocking algorithms shifts the complexity of engineering shared data access from the user's source into the lock-free library implementation. Thus lock-free programming techniques can often help increase the comprehensibility of the concurrent interactions in the user's implementation. In the process of creating a parallel network of temporal constraints (by utilizing nonblocking synchronization), we measure the increased simplicity of the code (in contrast to the application of mutual exclusion) with the certification variable σ_{ptcn}.

Lock-free systems typically utilize CAS in order to implement an optimistic speculation on the shared data. A contending process attempts to make progress by applying one or more writes on a local copy of the shared data. Afterwards, the process attempts to swap (CAS) the global data with its updated copy. Such an approach guarantees that from within a set of contending processes, there is at least one that succeeds within a finite number of steps (expressed as the nonblocking invariant η_{nb}). The system is non-blocking at the expense of some extra work performed by the contending processes. Linearizability [12] is an important correctness condition for concurrent nonblocking objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time between the time t_1 of its invocation and the time t_2 of its completion (expressed as the linearizability invariant η_{lin}). The consistency model implied by the linearizability requirements is stronger than the widely applied Lamport's sequential consistency model [16]. According to Lamport's definition, sequential consistency requires that the results of a concurrent execution are equivalent to the results yielded by *some* sequential execution (given the fact that the operations performed by each individual processor appear in the sequential history in the order as defined by the program).

Practical nonblocking programming techniques stand at a development level Δ_{PL} and when used properly help in assuring safe and efficient access to shared data (in a concurrent system defined by the system's Δ_{MC}) and their semantics and implementation is directly related to the atomic primitives available by the system's Φ level (such as the availability of atomic primitives like Compare-And-Swap (CAS) or Double-Compare-And-Swap (DCAS)). In hardware platforms that do not provide complex atomic primitives (involving the atomic update of more than a single-word location), the implementation of lock-free algorithms is CAS-based. Such systems impose yet another important invariant η_{aba} where the programmer must eliminate the possibility of occurrence of the ABA problem [5].

The ABA problem [19] is fundamental to all CAS-based systems. There are two particular instances that create ABA hazards: 1. the user intends to store a memory address value

A multiple times, and 2. the memory allocator reuses the address of an already freed object. In these scenarios a CAS speculation can succeed despite the fact that the speculating process has experienced an interrupt and the value about to be updated had been modified by other processes. The completion of such a CAS speculation can lead to ABA and can seriously affect the semantics of the nonblocking algorithm. One approach to eliminating ABA is to strictly define the semantic usage pattern of a nonblocking algorithm (meaning that not all operations might be total at all time). Such usage rules are another example of a transformation of an invariant (η_{aba}) into a set of guarantees ($S_{\gamma_{aba}}$) variables that need to be satisfied. One possibility is the application of static analysis [29] that can check for the exclusion of interleavings leading to ABA hazards. In such a scenario the ABA problem (η_{aba}) is resolved by the application of ΔAT development tools.

In our case study we describe the integration of a nonblocking vector in a parallel implementation of the Mission Data System's Temporal Constraint Network Library (TCN) in order to achieve higher thread safety and boost the performance of the MDS Goal Networks component.

4.1 Temporal Constraint Networks

A Temporal Constraint Network (TCN) defines the goal-oriented operation of a control system in the context of a system under control. The Temporal Constraint Networks (TCN) application is at the core of the Jet Propulsion Laboratory's Mission Data System (MDS) [21] state-based and goal-oriented unified architecture for testing and development of mission software. A TCN consists of a set of temporal constraints (TCs) and a set of time points (TPs). In this model of goal-driven operation, a time point is defined as an interval of time when the configuration of the system is expected to satisfy a property predicate. The width of the interval corresponds to the temporal uncertainty inherent in the satisfaction of the predicate. Similarly, temporal constraints have an associated interval of time corresponding to the acceptable bounds on the interactions between the control system and the system under control during the performance of a specific activity. A TCN graph topology represents a snapshot at a given time of the known set of activities the control system has performed so far, is currently engaged in, and will be performing in the near future up to the horizon of the elaborated plan initially created as a solution for a set of goals. The topology of a temporal constraint network must satisfy a number of invariants ($S_{\eta_{tcn}}$).

(a) A TCN is a directed acyclic graph where the edges represent the set of all time points (S_{tps}) and the vertices the set of all temporal constraints (S_{tcs})

(b) For each time point $TP_i \in S_{tps}$, there is a set of temporal constraints that are immediate successors (S_{succ_i}) of TP_i and a set, S_{pred_i}, consisting of all of TP_i's immediate predecessors

(c) Each temporal constraint $TC_j \in S_{tcs}$ has exactly one successor TP_{succ_j} and one predecessor TP_{pred_j}

(d) For each pair $\{TP_i, TC_j\}$, where $TP_i \equiv TC_{succ_j}$, $TC_j \in S_{pred_i}$ must hold. The reciprocal invariant must also be valid, namely for each pair of $\{TP_i, TC_j\}$ such that $TP_i \equiv TC_{pred_j}$, $TC_j \in S_{succ_i}$

(e) The firing window of a time point $TP_i \in S_{tps}$ is represented by the pair of time instances $\{TP_{min_i}, TP_{max_i}\}$. Assuming that the current moment of time is represented by T_{now}, then $TP_{min_i} \leq T_{now} \leq TP_{max_i}$, for every $TP_i \in S_{tps}$.

General-purpose programming languages lack the capabilities to formally specify and check domain-specific design constraints. Direct representation and verification of the TCN invariants ($S_{\eta_{tcn}}$) in the implementation source code would result in a slow and cumbersome solution. However, any implementation (in C++, Java or another programming language) must operate under the assumptions that the basic TCN invariants are satisfied. Thus, prior to implementing a solution to the TCN constraint propagation problem, it is necessary to guarantee the correctness and consistency of the topology of the goal network.

4.2 TCN Verification and Automatic Parallelization

In this section we describe the design and application of a model-based framework for verification and parallelization of real-time C++ within JPL's MDS Framework (Figure 1). The main goal of the framework is to provide a model-driven design of a parallel temporal constraint propagation scheme where the following seven critical certification variables are satisfied: $S_{\eta_{safety}}$, ϵ_{exe}, η_{nb}, η_{lin}, η_{aba}, $S_{\eta_{tcn}}$, and σ_{ptcn}. The input to the framework is the MDS mission planning and execution module that is based on the definition of temporal constraint networks. At the core of the most recent implementations at JPL of this critical module is an optimized iterative algorithm for the real-time propagation of temporal constraints, developed and described by Lou in [17]. Constraint propagation poses performance challenges and speed bottlenecks due to the algorithm's frequent execution and the necessary real-time update of the goal network's topology. The end goal of the presented tool-chain is, given the implementation of the optimized iterative propagation scheme and the topology of a particular goal network, to establish the correctness of the core TCN semantic invariants (see Section 4.1) and *automatically* derive an implementation that can be executed concurrently. Our approach for achieving concurrent execution is based

on the idea of identifying Time Phases within a goal network, which allow the parallelization of the constraint propagation algorithm. A fundamental component of our model-driven design is the construction and execution of a formal verification model in Alloy[15] that represents the implementation's core semantics and functionality. We refine a formal modeling and analysis methodology [23] that helps us analyze the logical properties of the goal network model and automatically derive a meta-model for our parallel solution.

Figure 1. TCN Verification and Parallelization

4.3 Constraint Propagation

A classic solution to the problem of constraint propagation in TCN is the direct application of Floyd-Warshall's all-pairs-shortest-path algorithm[4], offering a complexity of $O(N^3)$, where N is the number of time points in the TCN topology. Since, by definition, the goal of the TCN propagation algorithm is to compute the real-time values of the network's temporal constraints, the algorithm is frequently executed and, given the massive scale of a real world goal network, can cause significant bottleneck for the overall system's performance. In [17], Lou describes an innovative and effective TCN propagation scheme with a complexity close to linear. Lou's TCN propagation is based on the concept of alternating forward and backward propagation passes. A forward pass updates the time interval at each time point by considering only its incoming temporal constraints. Similarly, a backward pass recomputes the time windows at each time point by considering only its outgoing temporal constraints. The scheme utilizes a shared container, named a *propagation queue*, to keep track of all time points whose successor time points' windows are about to be updated next (during a forward pass) and all time points whose predecessor time points' windows are about to be updated next (during a backward pass). A forward pass begins by selecting all time points with no predecessors and inserts them into the propagation queue. A backward pass begins by selecting all time points with no successors and inserts them into the propagation queue. Each iteration is carried out until:

(a) An iteration completes without updating any temporal constraints (thus indicating that there are no more updates to be performed during the pass). In this case, the TCN topology is considered to be *temporally consistent*.

(b) The iteration has stumbled upon a time window of negative value and the algorithm terminates with the outcome of having a temporally inconsistent network.

4.4 Model-based Development and Certification

Alloy [15] is a lightweight formal specification and verification tool for the automated analysis of user-specified invariants on complete or partial models. We use the Alloy specification language [15] to formally represent and check the TCN main invariants ($S_{\eta_{tcn}}$). In our C++ goal networks implementation we have applied generic programming techniques and concepts [22], so that we can maintain a higher level of expressiveness. As a result we have achieved a significant similarity in the way the main TCN notions and invariants are expressed in our actual implementation and the Alloy verification models. In addition, we utilize the Alloy Analyzer to implement our parallelization approach. Our method for parallelization of the goal network is based on the observation that in a topology we can identify groups of time points that would allow the concurrent execution of the propagation passes. A possible criterion for identifying such groups would be to identify the time points in a topology that allow disjoin-access to the shared data. Given the method used to compute the time window $[TP_{min_i}, TP_{max_i}]$ for each $TP_i \in S_{tps}$, we have observed that the functionally-independent time points are the time points that are equidistant (with respect to the longest path) from the root of the graph. Thus, in our methodology, we define a *Time Phase* Tph_i as the set of the time points (S_{Tph_i}) in a topology that are equidistant, with respect to the longest path, from the root of the graph. In such a way, by definition, the computations of $[TP_{min_a}, TP_{max_a}]$ and $[TP_{min_b}, TP_{max_b}]$ for every pair of $\{TP_a, TP_b\}$, such that $TP_a \in S_{Tph_i}$ and $TP_b \in S_{Tph_i}$, are mutually independent and allow disjoin-access to the shared data. With the support of Alloy Analyzer we define

and identify the time phases in a goal network graph. Having identified the time phases in our temporal constraint network specification in Alloy, the aim of the rest of our toolchain is to *automatically* derive the C++ implementation of the parallel solution through a number of code transformation techniques. Following Rouquette's methodology [23] for model transformation through the application of the Object Constraint Language (OCL) and the Eclipse Modeling Framework (EMF), we are able to automatically derive an intermediary XML and XSD representations of the graph's topology and the TCN semantic notions, respectively. We apply an XML parser (XercesC) and a CodeSynthesis XSD transformation tool to deliver the C++ implementation of the goal network and our parallel propagation method.

4.5 Analysis

Table 1 provides a summary of the applied development tools that help us satisfy the seven critical certification variables in the process of TCN verification and parallelization. Each non-empty cell indicates the level of abstraction of the applied development tool. Empty cells are designated by the $\emptyset$ symbol. Below we briefly explain each entry in the table:

Cert. Artifact	Δ_{MC}	Δ_{PL}	Δ_{MT}	Δ_{AT}	Δ_{SA}
$S_{\eta_{safety}}$	$\emptyset$	Θ, Λ	$\emptyset$	$\emptyset$	$\emptyset$
ϵ_{exe}	$\emptyset$	Θ, Λ	$\emptyset$	$\emptyset$	$\emptyset$
η_{nb}	Φ, Θ	Θ	$\emptyset$	$\emptyset$	$\emptyset$
η_{lin}	Φ, Θ	Θ	$\emptyset$	$\emptyset$	$\emptyset$
η_{aba}	Φ, Θ	Θ	$\emptyset$	Λ	$\emptyset$
$S_{\eta_{tcn}}$	$\emptyset$	$\emptyset$	Λ	Λ	Ξ
σ_{ptcn}	$\emptyset$	Λ	Ψ	Θ, Λ	$\emptyset$

Table 1. Linking Certification Artifacts, Development Tools, and Levels of Abstraction

(1) $S_{\eta_{safety}}$: to eliminate the dangers of deadlock (η_{delock}), livelock (η_{lvlock}), and priority inversion (η_{pinv}) we have relied on the use of a library of nonblocking algorithms that allow the fast and safe implementation of shared data access of the C++ STL vector. Thus our approach to deliver safe concurrent interactions is based on the application of innovative algorithms (Θ) and language library extensions (Λ).

(2) ϵ_{exe}: as described in detail in [5], when used under contention a nonblocking shared vector can deliver a significant performance boost (by a factor of 10 or more) when compared with the application of the most recent and optimized mutual exclusion schemes. In the scenarios when the shared data structure access patterns show less contention, the nonblocking techniques provide a scalable and efficient solution with performance better or equal to the most optimal mutual exclusive schemes [6]. Achieving better performance and scalability of our parallel goal network is also based on the application of programming techniques at the algorithms/library level.

(3) η_{nb}: the careful application of CAS-based speculation on single-word memory locations allows us to guarantee that among a set of contending processes trying to manipulate the shared vector, there is at least one that is guaranteed to progress. To construct our library of nonblocking algorithms we have relied on the atomic primitives provided by the hardware architecture (Φ) and a set of practical lock-free programming techniques (Θ).

(4) η_{lin}: some operations in a shared vector require the update (in a linearizable fashion) of two or more memory locations. Such operations are $push_back$ (need to update the tail and the size of the vector) and $resize$ (need to update the size and copy all elements). Implementing such operations in a linearizable fashion with the support of only single-word atomic primitives is notoriously difficult. We have employed a set of practical lock-free programming techniques to guarantee that the vector's operations are linearizable (such a technique is the use of Barne's-style announcement [5]).

(5) η_{aba}: The ABA problem is fundamental to all CAS-based systems and can affect the semantics of the nonblocking algorithms. In systems allowing complex atomic primitives such as CAS2 or DCAS, ABA can be easily avoided by attaching a version counter to each value. In such a case we would have had an algorithmic solution with a strong support from the hardware architecture. We cannot assume the availability of such complex atomic primitives in the hardware architecture of the flight-qualified embedded hardware. Our solution to the ABA problem is the application of a library for program analysis [29] that can help us eliminate hazardous interleavings.

(6) $S_{\eta_{tcn}}$: to guarantee the correct operation of our autonomous goal-driven application, we have build a framework (Figure 1) that relies on modeling, program analysis, and program transformation programming techniques.

(7) σ_{ptcn}: we have increased the comprehensibility of our parallel goal network implementation by: a. shifting the complexity of allowing safe and efficient concurrent operations into a library of nonblocking containers (Λ), b. used the Alloy modeling notation to express the software architectural and design notions (Ψ), and c. applied program analysis and transformation techniques to automatically derive the implementation source. Any further evolution of the system would rely on high-level models expressed in simpler design-level domain-specific terms.

5 Conclusion

We introduced an innovative framework for model-based product-oriented certification founded on the concept of source code enhancement and analysis. We offered a classification of the certification artifact types, the development and validation tools and techniques, the application domain-specific factors, and the levels of abstraction used in our

certification platform. We used our certification platform to analyze the model-driven development of a parallel propagation scheme of the MDS temporal constraint network module. In our analysis we identified seven critical certification artifact: 1. providing the safety of the concurrent interactions (by eliminating the hazards of deadlock, livelock, and priority inversion), 2. achieving better scalability and overall system performance, 3. allowing nonblocking synchronization, 4. having linearizable operations on the shared data, 5. eliminating the possibility of ABA corrupting the concurrent operations' semantics, 6. establishing the correctness of the core TCN graph invariants, and 7. having simpler to analyze and maintain parallel processes. In our discussion we explained the relationships among these seven certification artifacts and the underlying hardware architecture, the applied programming techniques, and program analysis, modeling, and transformation techniques. Our certification framework helped us formulate, express, and analyze the process of product-oriented certification for a complex computer-based system, such as the model-driven development tool-chain of parallel autonomous goal networks.

References

[1] A. Barett, R. Knight, J. Morris, and R. Rasmussen. Mission Planning and Execution Within the Mission Data System. In *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004.

[2] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington, and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. In *Formal Methods in Systems Design Journal*, September 2005.

[3] Columbia Accident Investigation Board. Columbia Accident Investigation Board Report Volume 1.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[5] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In A. A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2006.

[6] D. Dechev, N. Rouquette, P. Pirkelbauer, and B. Stroustrup. Verification and Semantic Parallelization of Goal-Driven Autonomous Software. In *Proceedings of ACM Autonomics 2008: 2nd International Conference on Autonomic Computing and Communication Systems*, 2008.

[7] E. Denney and B. Fischer. Software Certification and Software Certification Management Systems. In *SoftCement05. Proceedings of the 2005 ASE Workshop on Software Certificate Management*, 2005.

[8] E. Denney, B. Fischer, J. Schumann, and J. Richardson. Automatic Certification of Kalman Filters for Reliable Code Generation. In *Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

[9] D. Dvorak. Challenging encapsulation in the design of high-risk control systems. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, 2002.

[10] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reiinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *IEEE ISORC*, 2004.

[11] D. Dvorak, R. Rasmussen, and T. Starbird. State Knowledge Representation in the Mission Data System. In *Proceedings of IEEE Aerospace Conference*, 2002.

[12] M. Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[13] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[14] Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.

[15] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes programs, September 1979.

[17] J. Lou. An Efficient Algorithm for Propagation of Temporal Constraint Networks. *NASA Tech Brief Vol. 26 No. 4 from JPL New Technology Report NPO-21098*, April 2002.

[18] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[19] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[20] C. Perrow. *Normal Accidents*. Princeton University Press, September 1999.

[21] R. Rasmussen, M. Ingham, and D. Dvorak. Achieving Control and Interoperability Through Unified Model-Based Engineering and Software Engineering. In *AIAA Infotech at Aerospace Conference*, 2005.

[22] G. D. Reis and B. Stroustrup. Specifying C++ Concepts, ISO WG21 N1886, 2005.

[23] N. Rouquette. Analyzing and verifying UML models with OCL and Alloy. *EclipseCon 2008*, 2008.

[24] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[25] J. Schumann and W. Visser. Autonomy Software: V&V Challenges and Characteristics. In *Proceedings of the 2006 IEEE Aerospace Conference*, 2006.

[26] M. Sherriff and L. Williams. DevCOP: A Software Certificate Management System for Eclipse. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 375–384, Washington, DC, USA, 2006. IEEE Computer Society.

[27] A. Stoica, D. Keymeulen, A. Csaszar, Q. Gan, T. Hidalgo, J. Moore, J. Newton, S. Sandoval, and J. Xu. Humanoids for lunar and planetary surface operations. In *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, October 2005.

[28] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[29] B. Stroustrup and G. D. Reis. Supporting SELL for High-Performance Computing. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, LCPC 2005*, 2005.

[30] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty Architecture for Robotic Autonomy. In *IEEE Aerospace Conference*, March 2001.

[31] R. Volpe and S. Peters. Rover Technology Development and Mission Infusion for the 2009 Mars Science Laboratory Mission. In *7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, May 2003.

[32] D. Wagner. Data Management in the Mission Data System. In *Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.

Open Multi-Methods for C++

Peter Pirkelbauer Yuriy Solodkyy Bjarne Stroustrup

Texas A&M University,
College Station, TX 77843-3112
peter.pirkelbauer@tamu.edu, {yuriys, bs}@cs.tamu.edu

Abstract

Multiple dispatch – the selection of a function to be invoked based on the dynamic type of two or more arguments – is a solution to several classical problems in object-oriented programming. Open multi-methods generalize multiple dispatch towards open-class extensions, which improve separation of concerns and provisions for retroactive design. We present the rationale, design, implementation, and performance of a language feature, called open multi-methods, for C++. Our open multi-methods support both repeated and virtual inheritance. Our call resolution rules generalize both virtual function dispatch and overload resolution semantics. After using all information from argument types, these rules can resolve further ambiguities by using covariant return types. Great care was taken to integrate open multi-methods with existing C++ language features and rules. We describe a model implementation and compare its performance and space requirements to existing open multi-method extensions and workaround techniques for C++. Compared to these techniques, our approach is simpler to use, catches more user mistakes, and resolves more ambiguities through link-time analysis, runs significantly faster, and requires less memory. In particular, the runtime cost of calling an open multimethod is constant and less than the cost of a double dispatch (two virtual function calls). Finally, we provide a sketch of a design for open multi-methods in the presence of dynamic loading and linking of libraries.

Categories and Subject Descriptors D [3]: 3

General Terms Design, Languages, Performance

Keywords multi-methods, open-methods, multiple dispatch, object-oriented programming, generic programming, C++

1. Introduction

Runtime polymorphism is a fundamental concept of object-oriented programming (OOP), typically achieved by late binding of method invocations. "Method" is a common term for a function chosen through runtime polymorphic dispatch. Most OOP languages (e.g.: C++ [34], Eiffel [26], Java [3], Simula [6], and Smalltalk [20]) use only a single parameter at runtime to determine the method to be invoked ("single dispatch"). This is a well-known problem for operations where the choice of a method depends on the types of two or more arguments ("multiple dispatch"), such as the binary method problem [8]. Another problem is that dynamically dispatched functions have to be declared within class definitions. This is intrusive and often requires more foresight than class designers possess, complicating maintenance and limiting the extensibility of libraries. Open-methods provide an abstraction mechanism that solves these problems by separating operations from classes.

Workarounds for both of these problems exist for single-dispatch languages. In particular, the visitor pattern (double dispatch) [18] circumvents these problems without compromising type safety. Using the visitor pattern, the class-designer provides an accept method in each class and defines the interface of the visitor. This interface definition, however, limits the ability to introduce new subclasses and hence curtails program extensibility [12]. In [37] Visser presents a possible solution to the extensibility problem in the context of visitor combinators, which make use of RTTI.

Providing dynamic dispatch for multiple arguments avoids the restrictions of double dispatch. When declared within classes, such functions are often referred to as "multi-methods". When declared independently of the type on which they dispatch, such functions are often referred to as open class extensions, accessory functions [39], arbitrary multi-methods [28], or "open-methods". Languages supporting multiple dispatch include CLOS [32], MultiJava [12, 27], Dylan [30], and Cecil [10]). We implemented and measured both multi-methods and open-methods. Since open-methods address a larger class of design problems than multi-methods and are not significantly more expensive in time or space, our discussion concentrates on open-methods.

Generalizing from single dispatch to open-methods raises the question how to resolve function invocations when no overrider provides an exact type match for the runtime-types of the arguments. Symmetric dispatch treats each argument alike but is subject to ambiguity conflicts. Asymmetric dispatch resolves conflicts by ordering the argument based on some criteria (e.g.: an argument list is considered left-to-right). Asymmetric dispatch semantics is simple and ambiguity free (if not necessarily unsurprising to the programmer), but it is not without criticism [9]. In addition, asymmetric dispatch differs radically from C++'s symmetric function overload resolution rules.

We derive our design goals for the open-method extension from the C++ design principles outlined in [33]. For open-methods, this would mean the following. Open-methods should address several specific problems, be more convenient to use than all workarounds (e.g. the visitor pattern) as well as outperform them in both time and space. They should neither prevent separate compilation of translation units nor increase the cost of ordinary virtual function calls. Open-methods should be orthogonal to exception handling in order to be considered suitable for hard real-time systems (e.g. [25]), and parallel to the virtual and overload resolution semantics.

The contributions of this paper include:

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 6th international conference on Generative programming and component engineering, 2007. http://doi.acm.org/10.1145/1289971.1289993

GPCE'07, October 1–3, 2007, Salzburg, Austria.
Copyright © 2007 ACM 978-1-59593-855-8/07/0010…$5.00

- A design of open-methods that is coherent with C++ call-resolution semantics.
- An efficient implementation and performance data to support its practicality.
- A first known consideration of repeated and virtual inheritance for multi-methods.
- A novel idea of harnessing covariance of the return type for ambiguity resolution.
- A discussion of handling open-methods in the presence of dynamic linking.

2. Application Domains

Whether open-methods address a sufficient range of problems to be a worthwhile language extension is a popular question. We think they do, but do not consider the problem one that can in general be settled objectively, so we just present examples that would benefit significantly. We consider them characteristic for larger classes of problems.

2.1 Shape Intersection

An intersect operation is a classical example of multi-methods usage [33]. For a hierarchy of shapes, intersect() decides if two shapes intersect. Handling all different combinations of shapes (including those added later by library users) can be quite a challenge. Worse, a programmer needs specific knowledge of a pair of shapes to use the most specific and efficient algorithm.

Using the multi-method syntax from [33], with **virtual** indicating runtime dispatch, we can write:

```
bool intersect(virtual Shape&, virtual Shape&); // open-method
bool intersect(virtual Rectangle&, virtual Circle&); // overrider
```

We note that for some shapes, such as rectangles and lines, the cost of double dispatch can exceed the cost of the intersect algorithm itself.

2.2 Data Format Conversion

Consider an image format library, written for domains such as image processing or web browsing. Conversion between different representations is not among the core concerns of an image class and a designer of a format typically can't know all other formats. Designing a class hierarchy that takes aspects like this into account is hard particularly when these aspects depend on polymorphic behavior. In this case, generic handling of formats by converting them to and from a common representation in general gives unacceptable performance, degradation in image quality, loss of information, etc. Optimal conversion between different formats requires knowledge of exact source and destination types, therefore it is desirable to have open-class extensions in the language, like open-methods. Here is the top of a realistic image format hierarchy:

A host of concrete image formats such as RGB24, JPEG, and planar YUY2 will be represented by further derivations. The optimal conversion algorithm must be chosen based on a source-target pair of formats [22] [41].

2.3 Compiler Pass over an AST

High-level compiler infrastructures ([35], [29]) typically use abstract syntax trees (AST) to represent programs. OOP enables modeling the language constructs in an oo-hierarchy. Then, storing pointers to a base class allows the use of several classes in the same context. For example, a statement class would be the common base for selective (e.g.: **if** or **switch**) and iterative constructs e.g.: **for** or **while**. Analysis or transformation passes that would take the semantics of the statement into account (e.g.: dataflow framework) need to uncover the real type. This is typically implemented with visitors or type-tags used to cast to a derived class. Open-methods are a non-intrusive technique to write these compiler passes. They guarantee type-safety, and allow retroactive extension of the oo-hierarchy to support new language features or dialects.

3. Definition of Open-methods

Open-methods are dynamically dispatched functions, where the callee depends on the dynamic type of one or more arguments. ISO C++ supports compile-time (static) function overloading on an arbitrary number of arguments and runtime (dynamic) dispatch on a single argument. The two mechanisms are orthogonal and complementary. We define open-methods to generalize both, so our language extension must unify their semantics. Our dynamic call resolution mechanism is modeled after the overload resolution rules of C++. The ideal is to give the same result as static resolution would have given had we known all types at compile time. To achieve this, we treat the set of overriders as a viable set of functions and choose the single most specific method for the actual combination of types.

We derive our terminology from virtual functions: a function declared virtual in a base class (super class) can be overridden in a derived class (sub class):

- an *open-method* is a non-member function with one or more parameters declared virtual
- an *overrider* is an open-method that refines another open-method according to the rules defined in §3.1
- an open-method that does not override another open-method is called a *base-method*.

For example:

```
struct A { virtual ~A(); } a;
struct B : A {} b;

void print(virtual A&, virtual A&); // (1)
void print(virtual B&, virtual A&); // (2)
void print(virtual B&, virtual B&); // (3)
```

Here, both (2) and (3) are overriders of (1), allowing us to resolve calls involving every combination of A's and B's. For example, a call print(a,b) will involve a conversion of the B to an A and invoke (1). This is exactly what both static overload resolution and double dispatch would have done.

To introduce the role of multiple inheritance, we can add to that example:

```
struct X { virtual ~X(); };
struct Y : X, A {};

void print(virtual X&, virtual X&); // (4)
void print(virtual Y&, virtual Y&); // (5)
```

Here (4) defines a new open-method print on the class hierarchy rooted in X. Y inherits from both A and X, and since both print open-methods have the same signature, – (5) is an overrider for both (4) and (1).

3.1 Overriding

DEFINITION 1. *An open-method is considered an overrider (or) for an open-method (om) in the same translation unit if it has the same name, the same number of parameters, covariant virtual parameter types, invariant non-virtual parameter types, and a possibly covariant return-type.*

A base-method must be declared before any of its overriders. This restriction parallels other C++ rules and greatly simplifies compilation. As shown in the previous example, an overrider can be associated with more than one base-method.

For every overrider and base-method pair, the compiler checks, if the exception specifications and covariant return type (if present) comply with the semantics used for virtual functions.

DEFINITION 2. *An open-method that is not an overrider and an overrider that introduces a covariant return type are considered a* base-method *for a translation unit.*

DEFINITION 3. *A* Dispatch table (DT) *maps the type-tuple of the base-method's virtual parameters to actual overriders that will be called for that type-tuple.*

Millstein and Chambers show in [28] that open-methods cannot be modularly type checked if the language (like C++) supports multiple implementation inheritance. Therefore, we split our call resolution mechanism into three distinct stages:

- Overload resolution
- Ambiguity resolution
- Runtime dispatch

The goal of overload resolution is to find at compile time a unique base-method, through which the call can be dispatched. This base-method determines a dispatch table through which the call will be made, the necessary casts of the arguments, and the expected return type. The actual overrider to handle the call is determined at runtime.

The C++ overload resolution rules [23] are unchanged: the viable set includes both open-methods and regular functions and treats open-methods like any other freestanding functions. Following the static rules exactly would imply that a base-method is used only if an open-method is the best match. We can do slightly better; that is, we can resolve some static ambiguities at link time (where more information is available): if a set of best matches consists of open-methods only and the intersection of their base-methods has a single element - overload resolution does not report an ambiguity. We demonstrate with an example:

```
struct X;
struct Y;
struct Z;

void foo(virtual X&, virtual Y&); // (1)
void foo(virtual Y&, virtual Y&); // (2)
void foo(virtual Y&, virtual Z&); // (3)

struct XY : X, Y {} xy;
struct YZ : Y, Z {} yz;

void foo(virtual XY&, virtual Y&); // (4)
void foo(virtual Y&, virtual YZ&); // (5)
```

Open-methods 1,2 and 3 are three independent base-methods defined on different class hierarchies. Because XY and YZ are parts of several hierarchies, overriders 4 and 5 refine several base-methods. In particular, 4 is an overrider for 1 and 2, while 5 is an overrider for 2 and 3.

A call `foo(xy,yz);` is now ambiguous according to the standard overload resolution rules as both 4 and 5 are equally good matches. Our relaxed rule, however, does not reject this call as ambiguous at compile time, because these overriders have a unique base-method through which the call can be dispatched – 2.

At link time, when all the overriders have been seen, we check the overriders for return type consistency, perform ambiguity resolution and build the dispatch tables.

Runtime dispatch simply looks up the entry in the dispatch table that corresponds to the dynamic types of the arguments and dispatches to that function.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [21]. Further, the use of open-methods (as opposed to ordinary virtual functions and multi-methods) can be seen as adding a runtime dimension to generic programming [4].

3.2 Ambiguity Resolution

C++ supports single-, repeated-, and virtual inheritance:

Note that to distinguish repeated and virtual inheritance, this diagram represents sub-object relationships, not just sub-class relationships. We must handle all ambiguities that can arise in all these cases. By "handle", we mean resolve or detect as errors.

Our ideal for resolving open-method calls combines the ideals for virtual functions and overloading:

- virtual functions: the same function is called independently of which class in an inheritance hierarchy is used in the call.
- overloading: a call is considered unambiguous if (and only if) every parameter is at least as good a match for the actual argument as the equivalent parameter of every other candidate function and that it has at least one parameter that is a better match than the equivalent parameter of every other candidate function.

This implies that a call of a single-argument open-method is resolved equivalently to a virtual function call. The rules described in this paper closely approximate this ideal. As mentioned, the static resolution is done exactly according to the usual C++ rules. The dynamic resolution is presented as the algorithm for generating dispatch tables in §3.4. Before looking at that algorithm, we present some key motivating examples.

3.2.1 Single Inheritance

In object models supporting single inheritance (§3.2) ambiguities can only occur with open-methods taking at least two virtual parameters. Ambiguities in this case have to be resolved by introducing a new overrider. The resolution of an open-method with one argument is identical to that of a virtual function. Thus, open-methods provide an unsurprising mechanism for expressing non-intrusive ("external") polymorphism. This eliminates the need to complicate a class hierarchy just to support the later addition of additional "methods" in the form of visitors.

3.2.2 Repeated Inheritance

Consider the repeated inheritance case (§3.2) together with this set of open-methods visible at a call site to `foo(d1,d2)`:

```
void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
```

Even tough the compiler can determine a unique base-method foo(A&, A&) through which the call can be dispatched, the call with two arguments of type D gets rejected at compile-time. The problem in this case, is that there are multiple pathes from D to sub-objects of type A.

To resolve that conflict, a user can either add an overrider foo(D&,D&) visible at the call site or explicitly cast arguments to either the B or C sub-object. Making an overrider for foo(D&,D&) available at the call site, renders the choice of the path to the subobject irrelevant. It would always be dispatched to the same overrider.

If the (B,C)-vs.-(C,B) conflict is resolved by casting, a question remains on how the linker should resolve a call with two arguments of type D? We know at runtime (by looking into the virtual function table's open-method table (see §4) which "branch" of a D object (either B or C) is on. Thus, we can fill our dispatch table appropriately; that is, for each combination of types, there is a unique "best match" according to the usual C++ rules:

	A	**B**	**C**	**D/B**	**D/C**
A	AA	AA	AA	AA	AA
B	AA	BB	BC	BB	BC
C	AA	CB	CC	CB	CC
D/B	AA	BB	BC	BB	BC
D/C	AA	CB	CC	CB	CC

This depicts the dispatch table for the repeated-inheritance hierarchy in §3.2 and the set of overriders above. Since the base method is foo(A&,A&) and A occurs twice in D, each dimension has two entries for D: D/B meaning "D along the B branch". This resolution exactly matches our ideals.

3.2.3 Virtual Inheritance

Consider the virtual inheritance class hierarchy from §3.2 together with the set of open-methods from §3.2.2: In contrast to repeated inheritance, a D has only one A part, shared by B, C, and D. This causes a problem for calls requiring conversions, such as foo(b,d); is that D to be considered a B or a C? There is not enough information to resolve such a call. Note that the problem can arise is such a way that we cannot catch it at compile time:

```
C& rc = d;
foo(b,rc);
B& rb = d;
foo(b,rb);
```

Using static type information to resolve either call would violate the fundamental rule for virtual function calls: use runtime type information to ensure that the same overrider is called from every point of a class hierarchy. At runtime, the dispatch mechanism will (only) know that we are calling foo with a B and a D. It is not known whether (or when) to consider that D a B or a C. Based on this reasoning (embodied in the algorithm in §3.4) we must generate this dispatch table:

	A	**B**	**C**	**D/A**
A	AA	AA	AA	AA
B	AA	BB	BC	??
C	AA	CB	CC	??
D/A	AA	??	??	??

We cannot detect the ambiguities marked with ?? at compile time, but we can catch them at link time when the full set of overriders are known.

3.3 Covariant Return Types

Covariant return types are a useful element of C++. If anything, they appear to be more useful for operations with multiple arguments than for single argument functions. Covariant return types complicate the use of workaround techniques. In case of the visitor, it would require even more foresight and lead to a proliferation of accept methods that have to be replicated in each derived class.

As an example for using covariant return type, consider a class Symmetric derived from Matrix:

```
Matrix& operator+(Matrix&, Matrix&);
Symmetric& operator+(Symmetric&, Symmetric&);
```

It follows that we must generalize the covariant return rules for open-methods. Doing so turns out to be useful because covariant return types help resolve ambiguities.

In single dispatch, covariance of a return type implies covariance of the receiver object. Consequently, covariance of return types for open-methods implies an overrider (or) - base-method (bm) relationship between two open-methods. Liskov's substitution principle [24] guarantees that any call type-checked based on bm can use or's covariant result without compromising type safety.

This can be used to eliminate what would otherwise have been ambiguities. Consider the class hierarchies $A \leftarrow B \leftarrow C$ and $R1 \leftarrow R2 \leftarrow R3$ together with this set of open-methods:

```
R1* foo(virtual A&, virtual A&);
R2* foo(virtual A&, virtual B&);
R3* foo(virtual B&, virtual A&);
```

A call foo(b,b) appears to be ambiguous and the rules outlined so far would indeed make it an error. However, choosing R2* foo(A&,B&) would throw away information compared to using R3* foo(B&,A&): An R3 can be used wherever an R2 can, but R2 cannot be used wherever an R3 can. Therefore, we prefer a function with a more derived return type and for this example get the following dispatch table:

	A	**B**	**C**
A	AA	AB	AB
B	BA	BA	BA
C	BA	BA	BA

At first glance, this may look useful, but ad hoc. However, an open-method with a return type that differs from its base method becomes a new base method and requires its own dispatch table (or equivalent implementation technique). The fundamental reason is the need to adjust the return type in calls. Obviously, the resolutions for this new base method must be consistent with the resolution for its base method (or we violate the fundamental rule for virtual functions). However, since R2* foo(A&,B&) will not be part of R3* foo(B&,A&)'s table, the only consistent resolution is the one we chose.

If the return types of two overriders are siblings, then there is an ambiguity in the type-tuple that is a meet of the parameter-type tuples. Consider for example that $R3$ derives directly from $R1$ instead of $R2$, then none of the existing overriders can be used for (B,B) tuple as its return type on one hand has to be a subtype of $R2$ and on the other a subtype of $R3$. To resolve this ambiguity, the user will have to provide explicitly an overrider for (B,B), which must have the return type derived from both $R2$ and $R3$.

Using the covariant return type for ambiguity resolution also allows the programmer to specify preference of one overrider over another when asymmetric dispatch semantics is desired.

To conclude: covariant return types do not only improve static type information, but also enhance our ambiguity resolution mechanism. We are unaware of any other multi-method proposal using a similar technique.

3.4 Algorithm for Dispatch Table Generation

Let us assume we have a multi-method $rf(h_1, h_2, ..., h_k)$ with k virtual arguments. Class h_i is a base of hierarchy of the ith argument. $H_i = \{c : c <: h_i\}$ is a set of all classes from the hierarchy rooted at h_i. $X = H_1 \times H_2 \times \cdots \times H_k$ is the set of all possible argument type-tuples of f. Set $Y = \{(y_1, y_2, \cdots, y_k)\} \subseteq X$ is the set of argument type-tuples, on which the user defined overriders f_j for f. The set $O_f = \{f_0, \cdots, f_{m-1}\}$ is the set of those overriders ($f_0 \equiv f$). A mapping $F : Y \leftrightarrow O_f$ is a bijection between type-tuples on which overriders are defined and the overriders themselves.

Because different derivation paths may get different entries in the dispatch table, we assume that x_i in the type-tuple $x = (x_1, \cdots, x_k)$ identifies not only the concrete type, but also a particular derivation path for it (see [38] for formal definitions). Under this assumption, we define $B(x_i)$ to be a direct ancestor (base-class) of x_i in the derivation path represented by x_i. For example, for the repeated inheritance hierarchy from §3.2, $B(D/B) = B, B(D/C) = C, B(C) = A$, while for the virtual inheritance hierarchy $B(D/A) = A, B(B) = A, B(C) = A$.

For the sake of convenience, we define:

$$B_i(x) \equiv (x_1, \cdots, B(x_i), \cdots, x_k).$$

With it, we extend the definition of B to type-tuples as follows:

$$B(x) \equiv \{B_1(x), B_2(x), \cdots, B_k(x)\}.$$

$P(X, <) : (x_1, ..., x_k) <_P (y_1, ..., y_k) \Leftrightarrow \forall i : x_i <: y_i \wedge \exists j : y_j \not<: x_j$ defines a partial ordering that models ordering of viable functions for overload resolution as defined in [23]. $max_set(S) = \{x \in S \subseteq X : \nexists y \in S : x < y\}$ is a set of maximal elements of S with respect to the partial ordering P.

Dispatch table DT is a mapping $DT : X \rightarrow O_f$ that maps various combinations of argument types to the overriders used to handle that combination.

For any combination of argument types $x \in X$, we recursively define entries of the dispatch table DT as following:

$$DT[x] = \begin{cases} F(x), x \in Y \\ DT[max_set(B(x))], |max_set(B(x))| = 1 \\ Ambiguity, otherwise \end{cases}$$

The above recursion exhibits optimal substructure and has overlapping sub-problems, which lets us use dynamic programming [14] to create an efficient algorithm for generation of dispatch table, shown in Algorithm 1.

To analyze its performance, we first note that comparison of two type-tuples from X can be done in time $O(k)$. If $n = max(|H_i|, i = 1, k)$ and $r = max(r_i, i = 1, k)$ (where r_i is a maximum number of times h_i is used as non-virtual base class in any class of hierarchy H_i) then $|X| <= (n * r)^k$ and the amount of edges for topological sort is less then $k * (n * r)^k$. Therefore the complexity of topologically sorting X is $O(k * n^k)$. The second loop has complexity $O(k^2 * n^k)$ so the overall complexity is $O(n^k)$ since k is a constant defining the amount of virtual arguments. This means that the algorithm is linear in the size of the dispatch table.

3.5 Alternative Dispatch Semantics

Our open-method semantics strictly corresponds to virtual member function semantics in ISO C++ but does not entirely reflect overload resolution semantics. The reason is that less information is available for compile-time resolution than for link-time or runtime resolution. For example, consider the repeated inheritance class hierarchy from §3.2 with a virtual function added:

Algorithm 1 Dispatch Table Generation

```
S ← topological_sort(X)
for all x ∈ S do
  if x ∈ Y then
    DT[x] ← F(x)
  else
    max_set = {B_1(x)}
    for i ← 2, k do
      dominated ← false
      for all e ∈ max_set do
        if F^{-1}(DT[e]) <_P F^{-1}(DT[B_i(x)]) then
          max_set ← max_set − {e}
        else if F^{-1}(DT[B_i(x)]) <_P F^{-1}(DT[e]) then
          dominated ← true
          break
      if not dominated then
        max_set ← max_set ∪ {F^{-1}(DT[B_i(x)])}
    if |max_set| = 1 then
      DT[x] ← F(max_set)
    else
      Report ambiguity for x
```

```
struct A { virtual void foo(); };
struct B : A {};
struct C : A { virtual void foo(); };
struct D : B, C {};

void bar(A&); // conventional overloading
void bar(C&);

void foobar(virtual A&); // open−method
void foobar(virtual C&); // open−method

D d;
B& db = d; // B part of D
C& dc = d; // C part of D

// (runtime) Virtual Member Function Semantics:
b.foo(); // calls A::foo
c.foo(); // calls C::foo
d.foo(); // error: ambiguous

// (compile time) Overload Resolution Semantics:
bar(db); // calls bar(A&)
bar(dc); // calls bar(C&)
bar(d); // calls bar(C&) (why not ambiguous?)

// (runtime) open−method Semantics:
foobar(db); // calls foobar(A&)
foobar(dc); // calls foobar(C&)
foobar(d); // error: ambiguous
```

The difference between the ordinary virtual function (foo) calls and the ordinary overloaded resolution for (bar) is odd and depends on pretty obscure rules that may be more historical than fundamental. Calls to the open-method foobar follow the virtual function resolution.

Further differences emerge in cases where a different resolution become possible in cases where additional information from other translation units may become available to resolve open-methods (see §5). This parallels decisions for related parts of the language. For example, the resolution of **static_cast** and **dynamic_cast** can

differ even given identical arguments: **dynamic_cast** can use more information than **static_cast**.

Another difference to overloading is that the return type of the overriders is bounded by the return type of the base-method.

4. Implementation

We implemented open-methods as described here by modifying the EDG compiler front-end [15].

4.1 Changes to Compiler & Linker

Our mechanism extends ideas presented in [16, 39] as to compiler and linker model. We adopted the multi-method syntax proposed in [33], which in turn was inspired by an earlier idea by Doug Lea. One or more parameters of a non-static freestanding function can be specified to be **virtual**. Overloading functions based only on the virtual specifier is not allowed.

A virtual argument must be a reference or pointer to a polymorphic class (that is, a class containing at least one virtual function). For example:

```
struct A { virtual ~A(); };

void print(virtual A&); // ok
void print(int, virtual A&); // ok

void dump(virtual Shape); // compiler error
void dump(virtual int); // compiler error
```

Open-methods are generic freestanding functions, which do not have the access privileges of member functions. If an open-method needs access to non-public members of a class, that class must declare it a friend. There are no abstract (pure virtual) open-methods; that is every open-method must be defined. Consider a (dynamic) library D1 that introduces a new class and a second (dynamic) library D2 that defines a new abstract open-method. Then, the presence of an overrider for the class in D1 could not be guaranteed. The alternative would be runtime "method not defined" errors (reported as exceptions), but that solution would be inconsistent with the rest of C++ and would limit the use of open-methods in embedded systems.

For each translation unit, the compiler generates an *open-method description* (OMD) file that stores the data needed to generate the runtime data structure discussed in §4.2. This includes the names of all classes, their inheritance relationships, and the kind of inheritance. Open-methods are represented by name, return type, and their parameter list. Finally, the OMD-file also contains definitions of all user-defined types that appear in signatures of open-methods (both as virtual and regular parameters). These definitions are necessary to regenerate prototype declarations for the open-methods, which pass data through by value.

The pre-linker uses Coco/R [40] to parse the OMD-files. Then, the pre-linker synthesizes the OMD-data, associates all overriders with their base-methods, generates dispatch tables, issues link-errors for ambiguities, determines the indices necessary to access the open-method and dispatch-table, as well as defines and interlinks the om-tables of each sub-object type according to §4.2.

4.2 Changes to the Object Model

We augment the IA-64 C++ object model [13] by four elements to support constant time dispatching of open-methods. First, for each base-method there will be a dispatch table containing the function addresses. Second, the v-table of each sub-object contains an additional pointer to the *open-method table (om-table)*. Finally, the indices used for the open-method-table offsets are stored as global variables.

Figure 1 shows the layout of objects, v-tables, om-tables and dispatch-tables for repeated (left) and virtual (right) inheritance. Our extensions to the object-model are shown with grey background. From left to right the elements in each diagram represent the object, v-table, om-table, and dispatch table(s) for the class hierarchy in §3.2. From top to the bottom, the objects are of type A, B, C, and D respectively.

An open-method can be declared after the declarations of the classes used in its virtual parameters. Therefore, the compiler cannot reserve v-table entries to store the data related to open-method dispatch immediately in a class's virtual function table. Hence, we always extend every v-table by one pointer referencing the om-table, which can be laid down later by the pre-linker.

The om-table reserves one position for each virtual parameter of each base-method, where objects of this type can be passed as arguments. This position stores an index into corresponding dimension of the dispatch table. Since the size of the om-tables is not known at compile time, our technique relies on a literal for each open-method and virtual parameter position (called foo_1st, foo_2nd in Figure 1) that determines the offset within the om-tables.

Note that Figure 1 depicts our actual implementation, where entries for first argument positions already resolve one dimension of the table lookup. Entries for all other argument positions store the byte offset within the table.

In the presence of multiple-inheritance, a this-pointer shift might be required to pass the object correctly. In this case, we replace the address of the overrider by an address of a thunk that takes care of correctly adjusting the this-pointer. As described in §3.2.2 in case of repeated inheritance different bases can show different dispatch behavior depending on the sub-object to which the this-pointer refers. As a result, different bases may point to different om-tables. In case of virtual inheritance, the open-method dispatch entries are only stored through the types mentioned in the base-method. Hence, in the virtual inheritance case, all open-method calls are dispatched through the virtual base type.

4.3 Alternative Approaches

We considered a few other design alternatives to explore their trade-offs in extensibility and performance.

Multi-Methods: Multi-methods differ from open-methods in that the base-method has to be declared in the class definition of its virtual parameters. This allows the offset within the v-table be known at compile time, which saves two indirections per argument of a function call (one for the om-table, and one to read the index within the om-table). For a call with k virtual arguments, open-methods need $4k + 1$, while multi-methods need only $2k + 1$ memory references to dispatch a call. The downside of multi-methods is that existing classes cannot easily be extended with dynamically dispatched functions.

Chinese Remainders: In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [19]. The proposed scheme circumvents the necessity for open-method tables by moving all the necessary information from the class to the dispatch table.

Suppose that for every multi-method f there is a function $I_f : T \times N \rightarrow N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns index of type t in the nth dimension of the f's dispatch table. If such function is reasonably fast (preferably constant time) and its range is small (preferably from 0 to maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging rows and columns accordingly to the indices returned by I_f. As in [19] we use the Chinese reminder theorem [14] to generate function I_f.

Despite its elegance, this approach is rather theoretical because it is hard to use for large class hierarchies. The reason is that we need to assign different prime numbers to classes and perform

Figure 1. Object Model for repeated (left) and virtual (right) inheritance

computations on numbers that are bound by the product of these primes. Such product can fit into 32-bit integers for only the 9 first primes and into 64-bit integers for only the 15 first primes. Table compression techniques [2] or use of minimal perfect hash functions [14] instead, can help overcome the problem.

5. Dynamic Linking

Outside embedded systems, dynamically linked libraries are almost universally used with C++. Thus, a design for open-methods that does not allow for DLLs is largely theoretical. We do not currently have an implementation supporting dynamic linking, but we can outline a design addressing the major issues for open-methods in a dynamically linked library. It guarantees that the most specialized overrider available at runtime that preserves type-safety of a call will be used to dispatch a call.

Dynamic modules, compiled independently, may have different sets of overriders defined at the time of compilation. Furthermore, there could be new classes added to a hierarchy in one of the modules and objects of those classes may be passed into code of other modules. This is not a problem for regular virtual functions as their v-tables are found in the module where the class was defined. In case of open-methods, the dispatch table generated within a particular module can be simply unaware of a class, defined somewhere else. To account for this, the dynamic loader will have to update dynamically dispatch tables of each open-method as new modules are loaded and more information becomes available.

Covariant return introduces subtleties when dynamic linking is used. Consider a two-class hierarchy $A \leftarrow B$ and another two-class hierarchy $R1 \leftarrow R2$. The base-method R1 foo(**virtual** A&, **virtual** A&) is defined in a header visible by two dynamically linked modules D_1 and D_2 that do not know anything about each other. Module D_1 introduces overrider R2 foo(A&, B&) and module D_2 introduces overrider R1 foo(B&, B&). Each of the dynamically linked modules perfectly type-checks and links with foo() resolved through the dispatch table (a superscript in a cell denotes the type that is returned by an overrider e.g. AB^2 denotes R2 foo(A&, B&)):

$AA^1 in D_1$	**A**	**B**	$AA^1 in D_2$	**A**	**B**
A	AA^1	AB^2	**A**	AA^1	AA^1
B	AA^1	AB^2	**B**	AA^1	BB^1

When both modules are loaded together, we get the dilemma of how to resolve a call with both arguments of type B. On one side foo(B&,B&) from D_2 is more specialized, but on the other side foo(A&,B&) from D_1 imposes the additional requirement that the return type of whatever is called for (B,B) should be a subtype of R2, which R1 is not. Such scenario would have been rejected should it happen at compile/link time, however at load time we do not have this option anymore.

Keeping all dispatch tables of a particular open-method consistent on the overrider that will be called for a particular combination of types will force us to choose between suboptimal and type unsafe alternatives. What is worse - is that there may not be a unique type-safe alternative.

Imagine for example that a module D_3 introduces an overrider R3 foo(B&, A&) where $R1 \leftarrow R3$, so $R2$ and $R3$ are siblings. When D_1 and D_3 are loaded together, neither R2 foo(A&, B&) nor R3 foo(B&, A&) can be used to resolve a call with both arguments of type B - both alternatives are type unsafe for the other overrider.

To deal with this subtlety, we propose to weaken for the DLL case the requirement that the same overrider should be called for the same tuple of dynamic types regardless of the static types used at the call site. We require that the same overrider be used only if it is type-safe for the caller. Strictly speaking R1 foo(B&,B&) is not an overrider of R2 foo(A&, B&) as defined in §3.1, because its return type is not changing covariantly in respect to the types of arguments. Therefore, it cannot be considered for the dynamic resolution of calls made statically through the base-method R2 foo(A&, B&).

Taking the above into account, we propose that the dynamic linker fills in the dispatch table of every base-method independently. This results in:

AA^1	**A**	**B**	AB^2	**B**	BA^3	**A**	**B**
A	AA^1	AB^2	**A**	AB^2			
B	BA^3	BB^1	**B**	AB^2	**B**	BA^3	BA^3

It looks as if the dispatch table for the base-method R1 foo(A&,A&) now violates covariant consistency, but in reality it does not because all the return types in it are cast back through thunks to R1, which is the type statically expected at the call site.

As can be seen, this logic may result in different functions being called for the same type tuple depending on the base-methods seen at the call site. We note, however, that *the call is always made to the most specialized overrider that is type-safe for the caller*.

Even when covariant return types are not used, dynamic linking can introduce ambiguities in the dispatch tables. The simplest example would be to think of two different modules that both provide overrider for the same pair of dynamic types.

Let us consider a plausible scenario involving three DLLs:

```
// dll−1
struct GuiButton { virtual ~GuiButton(); };
struct GuiEvent { virtual ~GuiEvent(); };

void handleEvent(virtual GuiButton&, virtual GuiEvent&);

// dll−2
#include<dll1>
struct MyButton : GuiButton { };

void handleEvent(virtual MyButton&, virtual GuiEvent&);

// dll−3
#include<dll1>
struct SpecialEvent : GuiEvent {};

void handleEvent(virtual GuiElement&, virtual SpecialEvent&);
```

The first DLL defines a class GuiButton, a class GuiEvent, and a base-multi-method handleEvent. Internally, a second DLL derives a new type MyButton from GuiButton and introduces a new overrider for handleEvent. Likewise, the third DLL derives a new internal class SpecialEvent from GuiEvent and introduces a new overrider. The second and third DLL could stem from different vendors that do not know about each other.

Now a call of handleEvent with a MyButton and a SpecialEvent is ambiguous. The writer of the total system (the "system integrator") should in principle have considered that possibility and dealt with it. Therefore, one solution would be to terminate the program or to throw an exception [31]. However, such problems are hard to predict and design for. Relaxed Multi-Java [27] resolves these conflicts by introducing glue methods (to glue DLL2 and DLL3) that the system-integrator provides. While this might be a viable solution for software developers integrating several libraries, it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without direct request of a developer. This, for example, is the case with various component object models when application may ask the system to create an object with a particular name and operating system will locate and load the module it is resided in.

We emphasize that in current C++, it is not possible to write programs against both types without getting an ambiguity. We report such an ambiguity when it is detected at compile or link time, however, at load time we do not have such an option as it is detected on a user's machine.

One way to handle such a scenario will be to treat both classes as their base classes and dispatch appropriately. We note, however, that in the plug-in like usage scenarios modules may have only seen the interface: the base-multi-method and the roots of the hierarchies on which it is defined. Nevertheless, such use-cases would expect more refined overriders to handle calls on derived classes.

In principle, *both* handleEvent functions should correctly handle the event; that is, each handleEvent function must assume that its argument type is just base class and the actual argument could be of an unknown derived class. Consequently, each handleEvent function must be written in a way that is generic on its argument (probably using virtual functions on the individual argument). This implies that as long as an event handler's code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, it can be replaced by the other event handler. Even a non-deterministic selection of the overrider would produce a correct result! However, developers and testers strongly prefer deterministic choices, so we consistently choose one of the alternatives. Furthermore, as we have already mentioned, two DLLs may provide different implementations for the same overrider, in which case we have to make a choice.

With this said we propose to resolve ambiguities at load time as following. For each base-method of a given open-method:

- If there is a unique best match among type-safe overriders that can handle a particular combination of argument types – use it.
- If there is no unique best match, – make a deterministic choice among all best matches.

We chose an unspecified determinism over non-determinism to guarantee that the same method will always be selected and to keep the dynamic dispatch symmetric.

6. Related Work

Programming languages can support multi-methods either through built-in facilities, pre-processors, or through library extensions. Naturally, tighter language integration enjoys a much broader design space for type checking, ambiguity handling, and optimizations compared to libraries. In this section, we will first review both library and non-library approaches for C++ and then give a brief overview of multi-methods in other languages.

6.1 Cmm

Cmm [31] is a preprocessor based prototype implementation for an open-method C++ extension. It takes a translation unit and generates C++ dispatch code from it. Cmm is available in two versions. One of them uses RTTI to recover the dynamic type of objects to identify the best overrider. The other approach achieves constant time dispatch by relying on a virtual function overridden in each class. Dispatch ambiguities are signalled by throwing runtime exceptions. Cmm allows dynamically linked libraries to register and unregister their open-methods at load and unload time. In addition to open-method dispatch, Cmm also provides call-site virtual dispatch. Cmm does not provide special support for multiple inheritance and therefore its dispatch technique does not entirely conform with virtual function semantics.

6.2 DoubleCpp

DoubleCpp [5] is another preprocessor based approach for multi-methods dispatching on two virtual parameters. It essentially translates these multi-methods into the visitor pattern. For doing so, DoubleCpp requires access to the files containing the class definitions in order to add the appropriate accept and visit methods. DoubleCpp, like any other visitor-based approach, does not report but quietly resolve ambiguities.

6.3 Accessory Function

The accessory functions papers [16, 39] allow open method dispatch based on a single virtual argument and discuss ideas to extend the mechanism for multiple dispatch. The compilation model they

describe uses, like our approach, a compiler and linker cooperation to perform ambiguity resolution and dispatch table generation. However, the accessory functions are integrated into the regular v-tables of their receiver types, which requires the linker to not only generate the dispatch table but also to recompute and resolve the v-table index of any other virtual member function. Both papers do not provide a detailed discussion of the intricacies when multiple inheritance is involved. The authors do not refer to a model or prototype implementation to which we could compare our approach.

6.4 Loki

Loki [1], based on Alexandrescu's template programming library with the same name, provides several different dispatchers that balance between speed, flexibility, and code verbosity. Currently, it supports multi-methods with two arguments only, except for the constant-time dispatcher that allows more arguments. The static dispatcher provides call resolution based on overload resolution rules, but requires manual linearization of the class hierarchy in order to uncover the most derived type of an object first. All other dispatchers do not consider hierarchical relations and effectively require explicit resolution of all possible cases.

6.5 Other Languages

One of the first widely known languages to support multi-methods was CLOS [32]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity conflicts. Cecil [9, 10] views silent ambiguity resolution as a potential source for programming errors. Therefore, it uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at compile-time. In [28], Millstein and Chambers discuss the trade-offs between multi-methods and modular type-checking in languages with neither a total order of classes nor asymmetric dispatch semantics. Ranging from globally type-checked programs to modularly type-checked units, the models embrace or restrict the expressive power of the language to different degrees. Based on these findings, MultiJava [12] implements a model that allows separate compilation and eliminates the need for a link-time type-checker but also curtails extensibility. Relaxed MultiJava [27] re-introduces a link-time type checker and relies on the system integrator to resolve ambiguities by providing new overriders (glue-methods). Parasitic methods model multi-methods as class members and give the receiver precedence over other arguments. Implementations for Java [7] and Smalltalk [17] exist. An example of a language adding multi-methods through a library is Python [36]. Chambers and Chen [11] present an alternative implementation technique based on a lookup DAG. Their work generalizes multiple dispatch to be a subset of predicate-based dispatch.

7. Results

In order to discuss time and space performance, we compare a number of handcrafted implementations with EDG/Omm, our implementation described in §4, Cmm, DoubleCpp, and the Loki library. The handcrafted approaches include multi-methods and open-methods, model implementations used to initially assess the performance trade-offs, a Chinese Remainder (§4.3) based implementation, and the visitor pattern.

We wrote 20 classes (representing shapes, etc.) which can intersect each other. Overall, this results in 400 combinations for binary dispatch functions. We implemented 40 specific intersect functions to which all of the 400 combinations are dispatched to. In order to get a reliable timing of the function invocation, these 40 intersect functions only increment a counter. Since not all techniques we use support multiple inheritance, these 20 classes only use single inheritance. The actual test consists of a loop that randomly chooses 2 out of 32 objects and invokes the intersect method. We implemented a table-based random number generator that is simple and does not contain any floating-point calculations or integer-divisions. We ran the loop twice with the same random numbers: The first run allows implementations, which build the dispatch data structure on the fly to warm up and load data/code into the cache. The second loop was timed. The clock-cycle based timer takes the time before and after the loop and we calculate the average number of clock-cycles per loop to compare the results.

7.1 Implementations

We tested the approaches on a Pentium D, 2.8 Ghz running CentOS Linux and a Core2Duo running Mac OSX. The code was compiled with g++ 4.0 (Linux) and gcc 4.0.1 (OSX) with optimization level set to -O3. EDG/Omm generates source code lowered to C, which was compiled with the corresponding gcc versions and linked to the pre-linker generated dispatch tables.

Using the Chinese Remainder approach, the number associated with the dispatch table grows exponentially with the number of types. Therefore the test is limited to eight types instead of 20 and the size of the executable is omitted. We could only implement a simplified version that can handle eight types instead of 20. Hence, we omit the size of the program executable.

For Loki, we only tested the static dispatcher because the other require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of manually allocated array of functions through which calls are made. However, as we indicated before, the dual nature of multi-methods require them to provide both dynamic dispatch and automatic resolution mechanism.

7.2 Results & Interpretation

Our experimental results can be summarized in terms of execution time and program size:

Approach	**Size (bytes)** Linux	**Cycles/Loop** Pentium-D	**Cycles/Loop** Core2Duo
Virtual function	n/a	75	55
C++ Multi-method	19 547	78	60
C++ Open-method	19 725	82	63
EDG/Omm	70 647	82	64
Double Cpp	20 859	120	82
C++ Visitor	35 289	132	82
Chinese Remainders	n/a	175	103
Cmm (constant time)	112 250	415	239
Cmm	111 305	1 320	772
Loki Library	34 908	3 670	2 238

Executable size: We present the executable size obtained on the Linux system. The size of dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as programming language feature [39]. However, our results reveal that the visitor that is based on a brute force implementation is 80% bigger than the multi-method approach. With the visitor, each shape class has intersect methods for all 20 shapes of the hierarchy. A somewhat smarter approach would be to remove redundant intersect overriders. However, removing specific overriders is tedious and difficult to maintain, since the dispatch would be based on the static type information of the base class. Even an optimized approach would be at best able to match the multi-method implementation, simply because each type contains 20 intersect entries in the v-table. Multiplying this with the number of shapes, 20, results in 400, exactly the number of entries found in the dispatch table. The executable generated by EDG/Omm is bigger because it is statically linked to EDG's C++ runtime library. We do not discuss the program size of

the two Cmms and Loki, since they use additional header files such as <typeinfo> and <stdexcept> that distort a direct comparison.

Execution time: Multi-Methods, Open-Methods, and EDG/Omm are (as expected) roughly comparable to a single virtual function dispatch, which needs 75 (55 on the Core2Duo) cycles per loop. Hence, the better performance compared to the visitors is not surprising. However, the fact that multi-methods reduce the runtime to 62% (73%) of the reference implementation using the visitor is noteworthy. We conjecture this is an effect of the size of the class hierarchy and that the time to double dispatch depends on the number of overriders. On the Pentium D, two observations support our conjecture: firstly, the DoubleCpp-based visitor has no redundant overriders and runs slightly faster. Secondly, we simulated an analysis pass dispatching over AST-objects of 20 different types and counting the category to which they belong (type, declaration, expression, statement, other). In this case, the double dispatch has only 20 leaf-functions instead of 400 and our dispatch test runs 78 cycles instead of 132. The open-method approach requiring only five overriders, is still faster and needs 68 cycles.
The difference between multi-methods and open-methods (EDG/Omm) is within the expected range. Three more indirections require 4 (4) more clock cycles on the Pentium and 3 (4) more on the Core2Duo. Although significantly slower, Cmm (constant time) performs better than expected, since its author estimates the dispatch cost as 10 times a regular virtual function call. As expected the two non-constant time approaches perform worst.

Significance of performance: The performance numbers comes from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application the improved performance might or might not be significant. For the image conversion example, gains in execution speed are negligible compared to time spent in the actual conversion algorithm. In other cases, such as the evaluation of expressions using user-defined arithmetic types, traversal of abstract syntax trees, and some of the most frequent shape intersect examples, the speed differences among the double dispatch approaches appear to be notable.

Contrary to much "popular wisdom", our experiments revealed that for many applications the use of dispatch tables for open-methods and multi-methods actually reduce the program size compared to brute-force and work-around techniques.

8. Conclusions and Future Work

We presented a novel approach to dispatching open multi-methods that is in line with the multiple inheritance semantics of the current C++ object model and the C++ overload resolution rules. This implies compile-time or link-time detection of ambiguities. By considering covariant return type in the ambiguity resolution, we reduce the number of potential conflicts. We have discussed an implementation based on modifications to the EDG compiler front-end and have described a mechanism that supports the integration of several translation units. Our evaluation of different approaches to implementing open-methods in C++ shows that our approach is significantly better (in time and space) than current alternatives. Indeed, it is almost as efficient as single dispatch. Since the dispatch is constant time and does not rely on exceptions to signal ambiguities, it is applicable in embedded and hard real-time systems.

Future plans to extend our work include:

8.1 Virtual Function Templates

Virtual function templates are a powerful abstraction mechanism not part of C++. Generating v-tables for virtual function templates requires a whole-program view and C++ traditionally relies almost exclusively on separate compilation of translation units. The pre-linker technique described here should be able to synthesize v-tables for virtual function templates as it does for open-methods.

8.2 Function Pointers to Open-methods

Pointers to open-methods would generalize pointer to member-functions. This allows to separate n-ary open-methods from their arguments and write code abstracted from a concrete operation.

8.3 Calling a Base Implementation

C++ provides a syntax to call a particular base implementation of a virtual member function directly, avoiding dynamic dispatch. This is often used to call the function in the base class. To do this, C++ requires the user to use a fully qualified name of virtual member function: e.g.: p−>MyClass::foo(); It is likely that similar functionality will be required for open-methods.

8.4 Space Optimizations

With large class hierarchies, the size of dispatch tables can become significant, especially when we consider support for covariant return types. The implementation of space optimizations techniques to compressing and reusing of dispatch tables [2] would reduce the memory footprint.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.

[2] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA '94*, pages 244–258, New York, NY, USA, 1994. ACM Press.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, 3rd edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] L. Bettini, S. Capecchi, and B. Venneri. Double dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.

[6] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula BEGIN*. Auerbach Press, Philadelphia, 1973.

[7] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for Java. In *OOPSLA '97*, pages 66–76, New York, NY, USA, 1997. ACM Press.

[8] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, 1995.

[9] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, pages 33–56, London, UK, 1992. Springer-Verlag.

[10] C. Chambers. The Cecil language: Specification and rationale. 3.2. Technical report, Department of Computer Science and Engineering. University of Washington, 2004.

[11] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *OOPSLA '99*, pages 238–255, New York, NY, USA, 1999. ACM Press.

[12] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[13] Codesourcery.com. The Itanium C++ ABI. Technical report, 2001.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[15] Edison Design Group. C++ Front End, March 2006.

[16] C. B. Flynn and D. Wonnacott. Reconciling encapsulation and dynamic dispatch via accessory functions. Technical Report 387, Rutgers University Department of Computer Science, 1999.

[17] B. Foote, R. Johnson, and J. Noble. Efficient Multimethods in a Single Dispatch Language. *Proceedings of the European Conference on Object-Oriented Programming, Glasgow, Scotland, July*, 2005.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36(2):139–156, 2006.

[20] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[21] D. Gregor, J. Jarvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming. *OOPSLA'06*, 2006.

[22] International Organization for Standardization. *ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines*. 1994.

[23] ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.

[24] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87*, pages 17–34, New York, NY, USA, 1987. ACM Press.

[25] Lockheed Martin. *Joint Strike Fighter, Air Vehicle, C++ Coding Standard*. Lockheed Martin, December 2005.

[26] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[27] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA '03*, pages 224–240, New York, NY, USA, 2003. ACM Press.

[28] T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP '99*, volume 1628 of *LNCS*, pages 279–303, London, UK, 1999. Springer-Verlag.

[29] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03*, volume 2789 of LNCS, pages 214–223. Springer-Verlag, August 2003.

[30] A. Shalit. *The Dylan Reference Manual. 2nd edition*. Apple Press, 1996.

[31] J. Smith. Draft proposal for adding multimethods to C++. Technical Report N1463, JTC1/SC22/WG21 C++ Standards Committee, 2003.

[32] G. L. Steele Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[33] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[34] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[35] B. Stroustrup and G. Dos Reis. Supporting SELL for high-performance computing. In *LCPC'05*, volume 4339 of LNCS, pages 458–465. Springer-Verlag, October 2005.

[36] G. van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.

[37] J. Visser. Visitor combination and traversal control. In *OOPSLA '01*, pages 270–282, New York, NY, USA, 2001. ACM Press.

[38] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06*, pages 345–362, New York, NY, USA, 2006. ACM Press.

[39] D. Wonnacott. Using accessory functions to generalize dynamic dispatch in single-dispatch object-oriented languages. In *COOTS*, pages 93–102. USENIX COOTS, 2001.

[40] A. Wöß, M. Löberbauer, and H. Mössenböck. LL(1) conflict resolution in a recursive descent compiler generator. In *JMLC'03*, volume 2789 of LNCS, pages 192–201. Springer-Verlag, 2003.

[41] www.fourcc.org. Video codec and pixel format definition, February 2007.

Learning Standard C++ as a New Language

Bjarne Stroustrup

AT&T Labs

ABSTRACT

To get the most out of Standard C++ [C++,1998], we must rethink the way we write C++ programs. An approach to such a "rethink" is to consider how C++ can be learned (and taught). What design and programming techniques do we want to emphasize? What subsets of the language do we want to learn first? What subsets of the language do we want to emphasize in real code?

This paper compares a few examples of simple C++ programs written in a modern style using the standard library to traditional C-style solutions. It argues briefly that lessons from these simple examples are relevant to large programs. More generally, it argues for a use of C++ as a higher-level language that relies on abstraction to provide elegance without loss of efficiency compared to lower-level styles.

1 Introduction

We want our programs to be easy to write, correct, maintainable, and acceptably efficient. It follows that we ought to use C++ – and any other programming language – in ways that most closely approximate this ideal. It is my conjecture that C++ community has yet to internalize the facilities offered by Standard C++ so that major improvements relative to the ideal can be obtained from reconsidering our style of C++ use. This paper focuses on the styles of programming that the facilities offered by Standard C++ support – not the facilities themselves.

The key to major improvements is a reduction of the size and complexity of the code we write through the use of libraries. Below, I demonstrate and quantify these reductions for a couple of simple examples such as might be part of a introductory C++ course.

By reducing size and complexity, we reduce development time, ease maintenance, and decrease the cost of testing. Importantly, we also simplify the task of learning C++. For toy programs and for students who program only to get a good grade in a nonessential course, this simplification would be sufficient. However, for professional programmers efficiency is a major issue. Only if efficiency isn't sacrificed can we expect our programming styles to scale to be usable in systems dealing with the data volumes and real-time requirements regularly encountered by modern services and businesses. Consequently, I present measurements that demonstrate that the reduction in complexity can be obtained without loss of efficiency.

Finally, I discuss the implications of this view on approaches to learning and teaching C++

2 Complexity

Consider a fairly typical second exercise in using a programming language:

```
write a prompt "Please enter your first name"
read the name
write out "Hello <name>"
```

In Standard C++, the obvious solution is:

```
#include<iostream>   // get standard I/O facilities
#include<string>     // get standard string facilities
```

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
int main()
{
    using namespace std;      // gain access to standard library

    cout << "Please enter your first name:\n";
    string name;
    cin >> name;
    cout << "Hello " << name << ´\n´;
}
```

For a real novice, we need to explain the ‘‘scaffolding:’’ What is ***main*( )**? What does ***#include*** mean? What does ***using*** do? In addition, we need to understand all the ‘‘small’’ conventions, such as what ***\n*** does, where semicolons are needed, etc.

However, the main part of the program is conceptually simple and differs only notationally from the problem statement. We have to learn the notation, but doing so is relatively simple: ***string*** is a string, ***cout*** is output, **<<** is the operator we use write to output, etc.

To compare, consider a traditional C-style solution†:

```
#include<stdio.h>     // get standard I/O facilities

int main()
{
    const int max = 20;        // maximum name length is 19 characters
    char name[max];

    printf("Please enter your first name:\n");
    scanf("%s", name);         // read characters into name
    printf("Hello %s\n", name);

    return 0;
}
```

Objectively, the main logic here is slightly – but only slightly – more complicated than the C++-style version because we have to explain about arrays and the magic ***%s***. The main problem is that this simple C-style solution is shoddy. If someone enters a ‘‘first name’’ that is longer than the magic number 19 (the stated number 20 minus one for a C-style string terminating zero), the program is corrupted.

It can be argued that this kind of shoddiness is harmless as long as a proper solution is presented ‘‘later on.’’ However, that line of argument is at best ‘‘acceptable’’ rather than ‘‘good.’’ Ideally, a novice user isn’t presented with a program that brittle.

What would a C-style program that behaved as reasonably as the C++-style one look like? As a first attempt we could simply prevent the array overflow by using ***scanf*( )** in a more appropriate manner:

```
#include<stdio.h>     // get standard I/O facilities

int main()
{
    const int max = 20;
    char name[max];

    printf("Please enter your first name:\n");
    scanf("%19s", name);       // read at most 19 characters into name
    printf("Hello %s\n", name);

    return 0;
}
```

There is no standard way of directly using the symbolic form of the buffer size, ***max***, in the ***scanf*( )** format string, so I had to use the integer literal. That is bad style and a maintenance hazard. The expert-level alternative is not one I’d care to explain to novices:

† For aesthetic reasons, I use C++ style symbolic constants and C++ style **//**-comments. To get strictly conforming ISO C programs, use ***#define*** and **/* */** comments.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
char fmt[10];
sprintf(fmt, "%%%ds", max-1);   // create a format string: plain %s can overflow
scanf(fmt, name);               // read at most max-1 characters into name
```

Furthermore, this program throws ‘‘surplus’’ characters away. What we want is for the string to expand to cope with the input. To achieve that, we have to descend to a lower level of abstraction and deal with individual characters:

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>

void quit()                                     // write error message and quit
{
    fprintf(stderr, "memory exhausted\n");
    exit(1);
}

int main()
{
    int max = 20;
    char* name = (char*)malloc(max);            // allocate buffer
    if (name == 0) quit();

    printf("Please enter your first name:\n");

    while (true) {                              // skip leading whitespace
        int c = getchar();
        if (c == EOF) break;                    // end of file
        if (!isspace(c)) {
            ungetc(c, stdin);
            break;
        }
    }

    int i = 0;
    while (true) {
        int c = getchar();
        if (c == ´\n´ || c == EOF) {            // at end; add terminating zero
            name[i] = 0;
            break;
        }
        name[i] = c;
        if (i==max-1) {                         // buffer full
            max = max+max;
            name = (char*)realloc(name, max);   // get a new and larger buffer
            if (name == 0) quit();
        }
        i++;
    }

    printf("Hello %s\n", name);
    free(name);                                 // release memory
    return 0;
}
```

Compared to the previous versions, this seems rather complex. I feel a bit bad adding the code for skipping whitespace because I didn’t explicitly require that in the original problem statement. However, skipping initial whitespace is the norm and the other versions of the program skip whitespace.

One could argue that this example isn’t all that bad. Most experienced C and C++ programmers would – in a real program – probably (hopefully?) have written something equivalent in the first place. We might even argue that if you couldn’t write that program, you shouldn’t be a professional programmer. However, consider the added conceptual load on a novice. This variant uses nine different standard library functions, deals with character-level input in a rather detailed manner, uses pointers, and explicitly deals with free

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

store. To use ***realloc*()** while staying portable, I had use ***malloc*()** (rather than ***new***). This brings the issues of sizes and casts† into the picture. It is not obvious what is the best way to handle the possibility of memory exhaustion in a small program like this. Here, I simply did something obvious to avoid the discussion going off on another tangent. Someone using the C-style approach would have to carefully consider which approach would form a good basis for further teaching and eventual use.

To summarize, to solve the original simple problem, I had to introduce loops, tests, storage sizes, pointers, casts, and explicit free-store management in addition to whatever a solution to the problem inherently needs. This style is also full of opportunity for errors. Thanks to long experience, I didn't make any of the obvious off-by-one or allocation errors. Having primarily worked with stream I/O for a while, I initially made the classical beginner's error of reading into a ***char*** (rather that into an ***int***) and forgetting to check for ***EOF***. In the absence of something like the C++ standard library, it is no wonder that many teachers stick with the ''shoddy'' solution and postpone these issues until later. Unfortunately, many students simply note that the shoddy style is ''good enough'' and quicker to write than the (non-C++ style) alternatives. Thus they acquire a habit that is hard to break and leave a trail of buggy code behind.

This last C-style program is 41 lines compared to 10 lines for its functionally equivalent C++-style program. Excluding ''scaffolding,'' the difference is 30 lines vs 4. Importantly, the C++-style lines are also shorter and inherently easier to understand. The number and complexity of concepts needed to be explained for the C++-style and C-style versions are harder to measure objectively, but I suggest a 10-to-1 advantage for the C++-style version.

3 Efficiency

Efficiency is not an issue in a trivial program like the one above. For such programs, simplicity and (type) safety is what matters. However, real systems often have parts where efficiency is essential. For such systems, the question becomes ''can we afford a higher level of abstraction?''

Consider a simple example of the kind of activity that occurs in programs where efficiency matters:

```
read an unknown number of elements
do something to each element
do something with all elements
```

The simplest specific example I can think of is a program to find the mean and median of a sequence of double precision floating-point numbers read from input. A conventional C-style solution would be:

```
// C-style solution:

#include<stdlib.h>
#include<stdio.h>

int compare(const void* p, const void* q)          // comparison function for use by qsort()
{
    register double p0 = *(double*)p;              // compare doubles
    register double q0 = *(double*)q;
    if (p0 > q0) return 1;
    if (p0 < q0) return -1;
    return 0;
}

void quit()                                        // write error message and quit
{
    fprintf(stderr,"memory exhausted\n");
    exit(1);
}
```

† I know that C allows this to be written without explicit casts. However, that is done at the cost of allowing unsafe implicit conversion of a ***void**** to an arbitrary pointer type. Consequently, C++ requires that cast.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
int main(int argc, char* argv[])
{
    int res = 1000;                          // initial allocation
    char* file = argv[2];

    double* buf = (double*)malloc(sizeof(double)*res);
    if (buf==0) quit();

    double median = 0;
    double mean = 0;
    int n = 0;                               // number of elements

    FILE* fin = fopen(file,"r");             // openfile for reading
    double d;
    while (fscanf(fin,"%lg",&d)==1) {        // read number, update running mean
        if (n==res) {
            res += res;
            buf = (double*)realloc(buf,sizeof(double)*res);
            if (buf==0) quit();
        }
        buf[n++] = d;
        mean = (n==1) ? d : mean+(d-mean)/n; // prone to rounding errors
    }

    qsort(buf, n, sizeof(double), compare);

    if (n) {
        int mid = n/2;
        median = (n%2) ? buf[mid] : (buf[mid-1]+buf[mid])/2;
    }

    printf("number of elements = %d, median = %g, mean = %g\n",n,median,mean);

    free(buf);
}
```

To compare, here is an idiomatic C++ solution:

```
// Solution using the Standard C++ library:

#include<vector>
#include<fstream>
#include<algorithm>

using namespace std;

int main(int argc, char* argv[])
{
    char* file = argv[2];
    vector<double> buf;

    double median = 0;
    double mean = 0;

    fstream fin(file,ios::in);                          // open file for input
    double d;
    while(fin>>d) {
        buf.push_back(d);
        mean = (buf.size()==1) ? d : mean+(d-mean)/buf.size();   // prone to rounding errors
    }

    sort(buf.begin(),buf.end());

    if (buf.size()) {
        int mid = buf.size()/2;
        median = (buf.size()%2) ? buf[mid] : (buf[mid-1]+buf[mid])/2;
    }
```

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
        cout << "number of elements = " << buf.size()
             << ", median = " << median << ", mean = " << mean << '\n';
}
```

The size difference is less dramatic than in the previous example (43 vs 24 non-blank lines). Excluding, irreducible common elements such as the declaration of ***main*()** and the calculation of the median (13 lines) the difference is 20 lines vs 11. The critical input-and-store loop and the sort are both significantly shorter in the C++-style program (9 vs 4 lines for the read-and-store loop, and 9 lines vs 1 line for the sort). More importantly, their logic is far simpler in the C++ version – and therefore far easier to get right.

Again, memory management is implicit in the C++-style program; a ***vector*** grows as needed when elements are added using ***push_back*()**. In the C-style program, memory management is explicit using ***realloc*()**. Basically, the ***vector*** constructor and ***push_back*()** in the C++-style program does what ***malloc*(), *realloc*(),** and the code tracking the size of allocated memory does in the C-style program. In the C++ style program, I rely on the exception handling to report memory exhaustion. In the C-style program, I added explicit tests to avoid the possibility of memory corruption.

Not surprisingly, the C++ version was easier to get right. I constructed this C++-style version from the C-style version by cut-and-paste. I forgot to include ***<algorithm>***, I left ***n*** in place rather than using ***buf.size*()** twice , and my compiler didn't support the local *using-directive* so I had to move it outside ***main*()**. One the other hand, after fixing these four errors, the program ran correctly first time.

To a novice, ***qsort*()** is ''odd.'' Why do you have to give the number of elements? (because the array doesn't know it). Why do you have to give the size of a ***double***? (because ***qsort*()** doesn't know that it is sorting ***double***s). Why do you have to write that ugly function to compare ***double***s? (because ***qsort*()** needs a pointer to function because it doesn't know the type of the elements that it is sorting). Why does ***qsort*()**'s comparison function take ***const void**** arguments rather than ***char**** arguments? (because ***qsort*()** can sort based on non-string values). What is a ***void**** and what does it mean for it to be ***const***? (''Eh, hmmm, we'll get to that later''). Explaining this to a novice without getting a blank stare of wonderment over the complexity of the answer is not easy. Explaining, ***sort*(*v.begin*(), *v.end*())** is comparatively easy: ''Plain ***sort*(*v*)** would have been simpler in this case, but sometimes we want to sort part of a container so it's more general to specify the beginning and end of what we want to sort.''

To compare efficiencies, I first determined how much input was needed to make an efficiency comparison meaningful. For 50,000 numbers the programs ran in less than half a second each, so I choose to compare runs with 500,000 and 5,000,000 input values:

Read, sort, and write floating-point numbers						
	unoptimized			optimized		
	C++	C	C/C++ ratio	C++	C	C/C++ ratio
500,000 elements	3.5	6.1	**1.74**	2.5	5.1	**2.04**
5,000,000 elements	38.4	172.6	**4.49**	27.4	126.6	**4.62**

The key numbers are the ratio; a ratio larger than one means that the C++-style version is faster. Comparisons of languages, libraries, and programming styles are notoriously tricky, so please do not draw sweeping conclusions from these simple tests. The numbers are averages of several runs on an otherwise quiet machine. The variance between different runs of an example was less than 1%. I also ran strictly ISO C conforming versions of the C-style programs. As expected there were no performance difference between those and their C-style C++ equivalents.

I had expected the C++-style program to be only slightly faster. Checking other C++ implementations, I found a surprising variance in the results. In some cases, the C-style version even outperformed the C++-style version for small data sets. However, the point of this example is that a higher level of abstraction and a better protection against errors can be affordable given current technology: The implementation I used is widely available and cheap – not a research toy. Implementations that claim higher performance are also available.

It is not unusual to find people being willing to pay a factor of 3, 10, or even 50 for convenience and better protection against errors. Getting the benefit together with a doubling or quadrupling of speed is spectacular. These figures should be the minimum that a C++ library vendor would be willing to settle for.

To get a better idea of where the time was spent, I ran a few additional tests:

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

500,000 elements						
	unoptimized			optimized		
	C++	C	C/C++ ratio	C++	C	C/C++ ratio
read	2.1	2.8	**1.33**	2.0	2.8	**1.40**
generate	.6	.3	**.5**	.4	.3	**.75**
read&sort	3.5	6.1	**1.75**	2.5	5.1	**2.04**
generate&sort	2.0	3.5	**1.75**	.9	2.6	**2.89**

Naturally, ‘‘read’’ simply reads the data and ‘‘read&sort’’ reads the data and sorts it but doesn’t produce output. To get a better feel for the cost of input, ‘‘generate’’ produces random numbers rather than reading.

5,000,000 elements						
	unoptimized			optimized		
	C++	C	C/C++ ratio	C++	C	C/C++ ratio
read	21.5	29.1	**1.35**	21.3	28.6	**1.34**
generate	7.2	4.1	**.57**	5.2	3.6	**.69**
read&sort	38.4	172.6	**4.49**	27.4	126.6	**4.62**
generate&sort	24.4	147.1	**6.03**	11.3	100.6	**8.90**

From other examples and other implementations, I had expected streamio to be somewhat slower than stdio. That was actually the case for a previous version of this program where I use ***cin*** rather than a ***fileestream***. It appears that on some C++ implementations, file I/O is much faster than ***cin***. The reason is at least partly poor handling of the tie between ***cin*** and ***cout***. However, these numbers demonstrate that C++-style I/O can be as efficient as C-style I/O.

Changing the programs to read and sort integers instead of floating-point values did not change the relative performance – though it was nice to note that making that change was much simpler in the C++ style program (2 edits as compared to 12 for the C-style program). That is a good omen for maintainability.

The differences in the ‘‘generate’’ tests reflect a difference in allocation costs. A ***vector*** plus ***push_back***( ) ought to be exactly as fast as an array plus ***malloc***( ) / ***free***( ), but it wasn’t. The reason appears to be failure to optimize away calls of initializers that do nothing. Fortunately, the cost of allocation is (always) dwarfed by the cost of the input that caused the need for the allocation.

As expected, ***sort***( ) was noticeably faster than ***qsort***( ). The main reason is that ***sort***( ) inlines its comparison operations whereas ***qsort***( ) must call a function.

It is hard to choose an example to illustrate efficiency issues. One comment I had from a colleague was that reading and comparing numbers wasn’t realistic. I should read and sort strings. So I tried this program:

```
#include<vector>
#include<fstream>
#include<algorithm>
#include<string>

using namespace std;

int main(int argc, char* argv[])
{
    char* file = argv[2];        // input file name
    char* ofile = argv[3];       // output file name

    vector<string> buf;

    fstream fin(file, ios::in);
    string d;
    while(getline(fin, d)) buf.push_back(d);  // add line from input to buf

    sort(buf.begin(), buf.end());
```

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
        fstream fout(ofile, ios::out);
        copy(buf.begin(), buf.end(), ostream_iterator<string>(fout, "\n")); // copy to output
    }
```

I transcribed this into C and experimented a bit to optimize the reading of characters. The C++-style code performs well even against hand-optimized C-style code that eliminates copying of strings. For small amounts of output there is no significant difference and for larger amounts of data ***sort*( )** again beats ***qsort*( )** because of its better inlining:

Read, sort, and write strings					
	C++	C	C/C++ ratio	C (no string copy)	optimized-C/C++ ratio
500,000 elements	8.4	9.5	**1.13**	8.3	**.99**
2,000,000 elements	37.4	81.3	**2.17**	76.1	**2.03**

I used 2 million strings because I didn't have enough main memory to cope with 5 million strings without paging.

To get an idea of what time was spent where, I also ran the program with the ***sort*( )** omitted:

Read and write strings					
	C++	C	C/C++ ratio	C (no string copy)	optimized-C/C++ ratio
500,000 elements	2.5	3.0	**1.20**	2.0	**.80**
2,000,000 elements	9.8	12.6	**1.29**	8.9	**.91**

The strings were relatively short (seven characters on average).

Note that ***string*** is a perfectly ordinary user-defined type that just happens to be part of the standard library. What we can do efficiently and elegantly with a ***string***, we can do efficiently and elegantly with many other user-defined types.

Why do I discuss efficiency in the context of programming style and teaching? The styles and techniques we teach must scale to real-world problems. C++ is – among other things – intended for large-scale systems and systems with efficiency constraints. Consequently, I consider it unacceptable to teach C++ in a way that leads people to use styles and techniques that are effective for toy programs only; that would lead people to failure and to abandon what was taught. The measurements above demonstrate that a C++ style relying heavily on generic programming and concrete types to provide simple and type-safe code can be efficient compared to traditional C styles. Similar results have been obtained for object-oriented styles.

It is a significant problem that the performance of different implementations of the standard library differ dramatically. For a programmer who wants to rely on standard libraries (or widely distributed libraries that are not part of the standard), it is often important that a programming style that delivers good performance on one system give at least acceptable performance on another. I was appalled to find examples where my test programs ran twice as fast in the C++ style compared to the C style on one system and only half as fast on another. Programmers should not have to accept a variability of a factor of four between systems. As far as I can tell, this variability is not caused by fundamental reasons, so consistency should be achievable without heroic efforts from the library implementors. Better optimized libraries may be the easiest way to improve both the perceived and actual performance of Standard C++. Compiler implementors work hard to eliminate minor performance penalties compared with other compilers. I conjecture that the scope for improvements is larger in the standard library implementations.

Clearly, the simplicity of the C++-style solutions above compared to the C-style solutions was made possible by the C++ standard library. Does that make the comparison unrealistic or unfair? I don't think so. One of the key aspects of C++ is its ability to support libraries that are both elegant and efficient. The advantages demonstrated for the simple examples hold for every application area where elegant and efficient libraries exist or could exist. The challenge to the C++ community is to extend the areas where these benefits are available to ordinary programmers. That is, we must design and implement elegant and efficient libraries for many more application areas and we must make these libraries widely available.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

4 Learning C++

Even for the professional programmer, it is impossible to first learn a whole programming language and then try to use it. A programming language is learned in part by trying out its facilities for small examples. Consequently, we always learn a language by mastering a series of subsets. The real question is not ‘‘Should I learn a subset first?’’ but ‘‘Which subset should I learn first?’’

One conventional answer to the question ‘‘Which subset of C++ should I learn first?’’ is ‘‘The C subset of C++.’’ In my considered opinion, that’s not a good answer. The C-first approach leads to an early focus on low-level details. It also obscures programming style and design issues by forces the student to face many technical difficulties to express anything interesting. The examples in §2 and §3 illustrate this point. C++’s better support of libraries, better notational support, and better type checking are decisive against a ‘‘C first’’ approach. However, note that my suggested alternative isn’t ‘‘Pure Object-Oriented Programming first.’’ I consider that the other extreme.

For programming novices, learning the programming language should support the learning of effective programming techniques. For experienced programmers who are novices at C++, the learning should focus on how effective programming techniques are expressed in C++ and on techniques that are new to the programmer. For experienced programmers, the greatest pitfall is often to concentrate on using C++ to express what was effective in some other language. The emphasis for both novices and experienced programmers should be concepts and techniques. The syntactic and semantic details of C++ are secondary to an understanding of design and programming techniques that C++ supports.

Teaching is best done by starting from well-chosen concrete examples and proceeding towards the more general and more abstract. This is the way children learn and it is the way most of us grasp new ideas. Language features should always been presented in the context of their use. Otherwise, the programmer’s focus shifts from producing systems to delight over technical obscurities. Focussing on language-technical details can be fun, but it is not effective education.

On the other hand, treating programming as merely the handmaiden of analysis and design doesn’t work either. The approach of postponing actual discussion of code until every high-level and engineering topic has been thoroughly presented has been a costly mistake for many. That approach drives people away from programming and leads many to serious underestimate the intellectual challenge in the creation of production-quality code.

The extreme opposite to the ‘‘design first’’ approach is to get a C++ implementation and start coding. When encountering a problem, point and click to see what the online help has to offer. The problem with this approach is that it is completely biased towards the understanding of individual features and facilities. General concepts and techniques and not easily learned this way. For experienced programmers, this approach has the added problem of reinforcing the tendency to think in a previous language while using C++ syntax and library functions. For the novice, the result is a lot of if-then-else code mixed with code snippets inserted using cut-and-paste from vendor-supplied examples. Often the purpose of the inserted code is obscure to the novice and the method by which it achieves its effect completely beyond comprehension. This is the case even for clever people. This ‘‘poking around approach’’ can be most useful as an adjunct to good teaching or a solid textbook, but on its own it is a recipe for disaster.

To sum up, I recommend an approach that

- proceeds from the concrete to the abstract,
- presents language features in the context of the programming and design techniques that they exist to support,
- presents code relying on relatively high-level libraries before going into the lower-level details (necessary to build those libraries),
- avoids techniques that do not scale to real-world applications,
- presents common and useful techniques and features before details, and
- focus on concepts and techniques (rather than language features).

No. I don’t consider this particularly novel or revolutionary. Mostly, I see it as common sense. However, common sense often gets lost in heated discussion about more specific topics such as whether C should be learned before C++, whether you must write Smalltalk to really understand Object-Oriented programming, whether you must start learning programming in a pure-OO fashion (whatever that means), and whether a thorough understanding of the software development process before trying to write code.

Fortunately, there is some experience with approaches that meet my criteria. My favorite approach is to

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

start teaching the basic language concepts such as variables, declarations, loops, etc. together with a good library. The library is essential to enable to students to concentrate on programming rather than the intricacies of, say C-style strings. I recommend the use of the C++ standard libraries or a subset of those. This is the approach taken by the Computer Science Advanced Placement course taught in American high schools [Horwitz,1999]. A more advanced version of that approach aimed at experienced programmers has also proved successful; for example, see [Koenig,1998].

A weakness of these specific approaches is the absence of a simple graphics and graphical user interfaces early on. This could (easily?) be compensated for by a very simple interface to commercial libraries. By ''very simple,'' I mean usable by students on day two of a C++ course. However, no such simple graphics and graphical user interface C++ library is widely available.

After the initial teaching/learning that relies on libraries, a course can proceed in a variety of ways based on the needs and interests of the students. At some point, the messier and lower-level features of C++ will have to be examined. One way of teaching/learning about pointers, casting, allocation, etc. is to examine the implementation of the classes used to learn the basics. For example, the implementation of ***string***, ***vector***, and ***list***
classes are excellent contexts for discussions of language facilities from the C subset of C++ that are best left out of the first part of a course.

Classes, such as ***vector*** and ***string***, that manage variable amounts of data require the use of free store and pointers in their implementation. Before introducing those, classes that doesn't require that (concrete classes), such as a ***Date***, a ***Point***, and a ***Complex*** types can be used to to introduce the basics of class implementation.

I tend to present abstract classes and class hierarchies after the discussion of containers and the implementation of containers, but there are many alternatives here. The actual ordering of topics should depend on the libraries used. For example, a course using a graphics library relying on class hierarchies will have to explain the basics of polymorphism and the definition of derived classes relatively early.

Finally, please remember there is no one right way to learn and teach C++ and its associated design and programming techniques. The aims and backgrounds of students differ and so does the backgrounds and experience of their teachers and textbook writers.

5 Summary

We want our C++ programs to be easy to write, correct, maintainable, and acceptably efficient. To do that, we must design and program at a higher level of abstraction than has typically been done with C and early C++. Through the use of libraries, this ideal is achievable without loss of efficiency compared to lower-level styles. Thus, work on more libraries, on more consistent implementation of widely-used libraries (such as the standard library), and on making libraries more widely available can yield great benefits to the C++ community.

Education must play a major role in this move to cleaner and higher-level programming styles. The C++ community doesn't need another generation of programmers who by default use the lowest level of language and library facilities available out of misplaced fear of inefficiencies. Experienced C++ programmers as well as C++ novices must learn to use Standard C++ as a new and higher-level language as a matter of course and descend to lower levels of abstraction only where absolutely necessary. Using Standard C++ as a glorified C or glorified C with Classes only would be to waste the opportunities offered by Standard C++.

6 Acknowledgements

Thanks to Chuck Allison for suggesting that I wrote a paper on learning Standard C++. Thanks to Andrew Koenig and Mike Yang for constructive comments on earlier drafts. My examples were compiled using Cygnus' EGCS1.1 and run on a Sun Ultrasparc 10. The programs I used can be found on my homepages: http://www.research.att.com/~bs.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

7 References

[C++,1998] X3 Secretariat: *Standard – The C++ Language*. ISO/IEC 14882:1998(E). Information Technology Council (NCITS). Washington, DC, USA. (See http://www.ncits.org/cplusplus.htm).

[Horwitz,1999] Susan Horwitz:*Addison-Wesley's Review for the Computer Science AP Exam in C++*. Addison-Wesley. 1999. ISBN 0-201-35755-0.

[Koenig,1998] Andrew Koenig and Barbara Moo: *Teaching Standard C++*. (part 1, 2, 3, and 4) Journal of Object-Oriented Programming, Vol 11 (8,9) 1998 and Vol 12 (1,2) 1999.

[Stroustrup,1997] Bjarne Stroustrup: *The C++ Programming language (Third Edition)*. Addison-Wesley. 1997. ISBN 0-201-88954-4.

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

Science of Computer Programming 75 (2010) 638–667

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Design and evaluation of C++ open multi-methods☆

Peter Pirkelbauer *, Yuriy Solodkyy *, Bjarne Stroustrup

Texas A&M University, College Station, TAMU 3112, TX 77843, USA

ARTICLE INFO

Article history:
Received 2 February 2008
Received in revised form 13 June 2009
Accepted 25 June 2009
Available online 7 July 2009

Keywords:
Multi-methods
Open-methods
Multiple dispatch
Binary method problem
Augmenting method problem
Object-oriented programming
Generic programming
C++

ABSTRACT

Multiple dispatch – the selection of a function to be invoked based on the dynamic type of two or more arguments – is a solution to several classical problems in object-oriented programming. Open multi-methods generalize multiple dispatch towards open-class extensions, which improve separation of concerns and provisions for retroactive design. We present the rationale, design, implementation, performance, programming guidelines, and experiences of working with a language feature, called open multi-methods, for C++. Our open multi-methods support both repeated and virtual inheritance. Our call resolution rules generalize both virtual function dispatch and overload resolution semantics. After using all information from argument types, these rules can resolve further ambiguities by using covariant return types. Care was taken to integrate open multi-methods with existing C++ language features and rules. We describe a model implementation and compare its performance and space requirements to existing open multi-method extensions and work-around techniques for C++. Compared to these techniques, our approach is simpler to use, catches more user mistakes, and resolves more ambiguities through link-time analysis, is comparable in memory usage, and runs significantly faster. In particular, the runtime cost of calling an open multi-method is constant and less than the cost of a double dispatch (two virtual function calls). Finally, we provide a sketch of a design for open multi-methods in the presence of dynamic loading and linking of libraries.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Runtime polymorphism is a fundamental concept of object-oriented programming (OOP), typically achieved by late binding of method invocations. "Method" is a common term for a function chosen through runtime polymorphic dispatch. Most OOP languages (e.g. C++ [52], Eiffel [38], Java [6], Simula [11], and Smalltalk [30]) use only a single parameter at runtime to determine the method to be invoked ("single dispatch"). This is a well-known problem for operations where the choice of a method depends on the types of two or more arguments ("multiple dispatch"), such as the binary method problem [15]. A separate problem is that dynamically dispatched functions have to be declared within class definitions. This is intrusive and often requires more foresight than class designers possess, complicating maintenance and limiting the extensibility of libraries. Open-methods provide an abstraction mechanism that solves these two problems by separating operations from classes and enabling the choice of dynamic vs. static dispatch on a per-parameter basis.

Work-arounds for both of these problems exist for single-dispatch languages. In particular, the visitor pattern (double dispatch) [28] circumvents these problems without compromising type safety. Using the visitor pattern, the class designer

☆ This paper is an expanded version of the paper that was presented at GPCE 2007.

* Corresponding author. Tel.: +1 979 845 2938.
E-mail addresses: peter.pirkelbauer@tamu.edu (P. Pirkelbauer), yuriys@cs.tamu.edu (Y. Solodkyy), bs@cs.tamu.edu (B. Stroustrup).
URLs: http://www.parasol.tamu.edu/~peterp/ (P. Pirkelbauer), http://www.parasol.tamu.edu/~yuriys/ (Y. Solodkyy), http://www.parasol.tamu.edu/~bs/ (B. Stroustrup).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.06.002

provides an accept method in each class and defines the interface of the visitor. This interface definition, however, limits the ability to introduce new subclasses and hence curtails program extensibility [20]. In [55], Visser presents a possible solution to the extensibility problem in the context of visitor combinators, which make use of runtime type information (RTTI).

Providing dynamic dispatch for multiple arguments avoids the restrictions of double dispatch. When declared within classes, such functions are often referred to as "*multi-methods*". When declared independently of the type on which they dispatch, such functions are often referred to as *open-class extensions*, *accessory functions* [57], *arbitrary multi-methods* [41], or "*open-methods*". Languages supporting multiple dispatch include CLOS [50], MultiJava [20,40], Dylan [47], and Cecil [17]). We implemented and measured both multi-methods and open-methods. Since open-methods address a larger class of design problems than multi-methods and are not significantly more expensive in time or space, our discussion concentrates on open-methods.

Generalizing from single dispatch to open-methods raises the question how to resolve function invocations when no overrider provides an exact type match for the runtime-types of the arguments. Symmetric dispatch treats each argument alike but is subject to ambiguity conflicts. Asymmetric dispatch resolves conflicts by ordering the arguments based on some criteria (e.g. an argument list is considered left-to-right). Asymmetric dispatch semantics is simple and ambiguity free (if not necessarily unsurprising to the programmer), but it is not without criticism [16]. It differs radically from C++'s symmetric function overload resolution rules and does not catch ambiguities.

We derive our design goals for the open-method extension from the C++ design principles outlined in [51, Section 4]. For open-methods, this means the following: open-methods should address several specific problems, be more convenient to use than all work-arounds (e.g. the visitor pattern), and outperform them in both time and space. They should neither prevent separate compilation of translation units nor increase the cost of ordinary virtual function calls. Open-methods should be orthogonal to exception handling in order to be considered suitable for hard real-time systems (e.g. [37]), and parallel to the virtual and overload resolution semantics.

The contributions of this paper include:

- A design of open-methods that is consistent with C++ call-resolution semantics.
- An efficient implementation and performance data to support its practicality.
- A first known consideration of repeated and virtual inheritance for multi-methods.
- A novel idea of harnessing covariance of the return type for ambiguity resolution.
- A discussion of handling open-methods in the presence of dynamic linking.

Section 2 presents application domains for both open-methods and multi-methods as well as discuss the style of programming and the new techniques that open-methods enable. Section 3 describes our function call and ambiguity resolution mechanisms. Section 4 explains our design decisions, and compares their trade-offs in the context of (dynamically linked) libraries to trade-offs made by other researchers. Section 5 discusses the relationship of open-methods to other language features. Section 6 shows the necessary modifications to the C++ compiler and linker model as well as extensions of the IA-64 object model [22] based on our implementation. Section 7 gives an overview of research in the area of multi-methods for C++. Section 8 compares the performance of our approach to other methods that add support for multi-methods to C++. Section 9 compares open-methods to the visitor pattern in two real-world applications. Section 10 summarizes our contributions and sketches remaining open problems.

2. Application domains

Whether open-methods address a sufficient range of problems to be a worthwhile language extension is a popular question. We think they do, but like all style questions it is not a question that can in general be settled without examples and data. This is why in the context of this paper we start with presenting examples that we consider characteristic for larger classes of problems and that would benefit significantly. We then explain fundamentals that drive the design, provide the details of our implementation and compare its performance to alternative solutions. In what follows, we mark examples with 1 when they primarily demonstrate multiple dispatch and with 2 when they demonstrate open-class extensions.

2.1. Shape intersection[1]

An intersect operation is a classical example of multi-method usage [51, Section 13.8]. For a hierarchy of shapes, intersect() decides if two shapes intersect. Handling all different combinations of shapes (including those added later by library users) can be quite a challenge. Worse, a programmer needs specific knowledge of a pair of shapes to use the most specific and efficient algorithm.

Using the multi-method syntax from [51, Section 13.8], with **virtual** indicating runtime dispatch, we can write:

```
bool intersect (virtual const Shape&,     virtual const Shape& ); // open—method
bool intersect (virtual const Rectangle&, virtual const Circle&  ); // overrider
```

We note that for some shapes, such as rectangles and lines, the cost of double dispatch can exceed the cost of the intersect algorithm itself.

2.2. Data format conversion[1 2]

Consider an image format library, written for domains such as image processing or web browsing. Conversion between different representations is not among the core concerns of an image class and a designer of a format typically cannot know all other formats. Designing a class hierarchy that takes aspects like this into account is hard, particularly when these aspects depend on polymorphic behavior. In this case, generic handling of formats by converting them to and from a common representation in general gives unacceptable performance, degradation in image quality, loss of information, etc. An optimal conversion between different formats requires knowledge of exact source and destination types; therefore it is desirable to have open-class extensions in the language, like open-methods. Here is the top of a realistic image format hierarchy:

A host of concrete image formats such as RGB24, JPEG, and planar YUY2 will be represented by further derivations. The optimal conversion algorithm must be chosen based on a source–target pair of formats [33,60]. In Section 9, we present an implementation of this example; here we simply demonstrate with a call:

```
bool convert(virtual const image& src, virtual image& dst);

RGB24 bmp("image.bmp");
JPEG jpeg;
convert(bmp,jpeg);
```

2.3. Type conversions in scripting languages[1 2]

Similar to Section 2.2, this example demonstrates the benefits of open-methods in the context of type conversions. Languages used for scripting are often dynamically typed and a value may often be converted to other types depending on use. For example, variable x initialized as string can be used in contexts where integers or even dates are expected, while variable y initialized as integer can perfectly be used inside the catenation of strings. In such cases, an interpreter will try to convert actual values to the type required in the context according to some conversion rules. A typical implementation of such conversion will use either nested switch statements or a table of pointers to appropriate conversion routines. None of these approaches is extensible or easy to maintain. However, multi-methods provide a natural mechanism for such implementations:

```
class ScriptableObject              { ~virtual ScriptableObject(); };
class Number  : ScriptableObject  {};
class Integer   :  Number           {};
class String    :  ScriptableObject {};

void convert (virtual const ScriptableObject& src,  virtual  ScriptableObject& tgt);
void convert (virtual const Number&           src,  virtual  String&           tgt);
void convert (virtual const String&           src,  virtual  Number&           tgt);
//  ...   etc.
```

2.4. Compiler pass over an AST[2]

High-level source-to-source transformation infrastructures [46,53] typically use abstract syntax trees (ASTs) to represent programs. Using OOP, the commonalities of the AST classes can be factored in an OO-hierarchy. Then, programmers can write runtime polymorphic code for a family of classes by using pointers/references to a common base class.

```
struct Expr                          { virtual  ~Expr(); };
struct UnaryExpr          :  Expr        { Expr& operand; };
struct NotExpr            :  UnaryExpr  {};
struct ComplementExpr : UnaryExpr  {};

const Expr& propagate_constants(virtual const Expr& e);
```

For example, an expression class (i.e. Expr) would be the common base for unary (e.g. not or complement) and binary expressions (e.g. add or multiplication). Analysis or transformation passes that take the semantics of the expression into account (e.g. for propagating constants) need to uncover the real type. Typical implementations rely on the visitor pattern or type-tags to uncover the concrete type. Open-methods are a non-intrusive alternative for writing these compiler passes. In Section 9, we discuss our experience in implementing such a pass with open-methods and visitors.

2.5. Binary method problem[1]

Often times we have a two-argument method, whose meaning is trivial to define when both arguments are of the same type, but not so obvious in cases when arguments are of different types, though related through inheritance. Such methods are characteristic to many logical and arithmetic operations, and have been vigorously studied by Bruce et al. [15]. They define a *binary method* of some object of type τ as a method that has an argument of the same type τ. Binary methods pose a typing problem, and among different solutions to the problem, the authors propose to use multi-methods.

In the multi-methods setting, binary methods simply become multi-methods with two arguments of the same type. Consider for example an equality comparison of two objects:

```
class Point { double x, y; };

bool equal(const Point& a, const Point& b) {
  return a.x == b.x && a.y == b.y;
}

class ColorPoint : Point { Color c; };
```

When a class ColorPoint derives from Point and adds a color property, the question arises on how equal should be defined: should it just compare the coordinates or should it also compare the color properties of both arguments? The second option is only viable if both arguments are of type ColorPoint. If the argument types differ, we can choose either to return false or to compare the coordinates only. Depending on the problem domain both choices can be acceptable. Here we simply note that multi-methods are an ideal solution for the implementation of the latter policy where the comparison of Point with ColorPoint only compares coordinates:

```
bool equal(virtual const Point&       a, virtual const Point&       b);
bool equal(virtual const ColorPoint& a, virtual const ColorPoint&  b);
```

2.6. Selection of optimal algorithm based on dynamic properties of objects[1]

Often, we can use dynamic types to choose a better algorithm for an operation than would be possible using only static information. Using open-methods we can use the dynamic type information to select more efficient algorithms at runtime without added complexity or particular foresight. Consider a matrix library providing algorithms optimized for matrices with specific dynamic properties. Storing these dynamic properties as object attributes is not easily extensible and is error prone in practice. Letting the compiler track them using open-methods for dispatch (runtime algorithm selection) is simpler. For instance, the result of $A * A^T$ is a symmetric matrix—if such a matrix appears somewhere in computations, we may consider a broader set of algorithms when the result is used in other computations.

```
class Matrix                    { virtual ~Matrix(); };
class SymMatrix  : Matrix       {};  // symmetric matrix
class DiagMatrix  : SymMatrix   {};  // diagonal matrix

Matrix&      operator+(virtual const Matrix&      a, virtual const Matrix&      b);
SymMatrix&  operator+(virtual const SymMatrix&  a, virtual const SymMatrix&  b);
DiagMatrix&  operator+(virtual const DiagMatrix&  a, virtual const DiagMatrix&  b);
```

Depending on the runtime type of the arguments, the most specific addition algorithm is selected at runtime and the most specific result type returned. The static result type would still be Matrix& when the static type of an argument is a Matrix& since we cannot draw a more precise conclusion about the dynamic type of the result (see Section 3.4 and Section 4.2 for details). However, since the operator is selected according to the dynamic type, the optimal algorithm will be used for the result when it is part of a larger expression.

Other interesting properties to exploit include whether the matrix is upper/lower triangular, diagonal, unitary, non-singular, or symmetric/Hermitian positive definite. Physical representations of those matrices may also take advantage of the knowledge about the structure of a particular matrix and use less space for storing the matrix.

The polymorphic nature of the multiple dispatch requires the result to be returned by either a reference or a pointer to avoid slicing. Since the reference must refer to a dynamically allocated object, this creates a lifetime problem for that object. Common approaches to such problems include relying on a garbage collector and using a proxy to manage the lifetime. An efficient proxy is easy to write:

```
// A memory—managing proxy class (note lowercase name).
class matrix
{
    std:: unique_ptr<Matrix> the_matrix; // pointer to the actual polymorphic Matrix

    matrix(Matrix& actual) : the_matrix(&actual) {}
};

matrix operator+(const matrix& a, const matrix& b)
{
    // Forward operator+ to the actual open—method and
    // attach the result to the proxy.
    return matrix(∗a.the_matrix + ∗b.the_matrix);
}
```

The unique_ptr is a simple and efficient (not-reference counting) "smart" pointer that is part of the C++0x standard [9] and has been widely available and used for years [8].

2.7. Action systems[1]

Dynamically typed languages, such as Smalltalk [30], Ruby [54] or Python [45], can dispatch polymorphic calls on classes not bound by inheritance. As long as a method with a given name exists in the class, it will be called; otherwise an exceptional action is taken. Often, similar behavior is desirable in statically typed languages with possible restriction to objects derived from a certain base class. To achieve this, we may represent methods (here called actions) as objects and then apply a given action to a given set of parameters.

The above figure shows a class hierarchy with objects and actions. Note that the same action applied to a different object may have a completely different meaning.

```
Object& execute(virtual const Action&      act, virtual Object&   obj);
String& execute(virtual const ToString&    act, virtual Number& obj);
  File & execute(virtual const SaveToFile& act, virtual Blob&     obj);
// ... etc.
```

Action objects resemble function objects in C++ in a way. The main difference between actions and C++ function objects is that actions participate in call dispatching on equal bases with other arguments, while function objects invariably define the scope for call dispatching. Simply put, this means that in case of action objects, other arguments of a call can affect the choice of a call's target at runtime (symmetric behavior), while with function objects they cannot (asymmetric behavior).

2.8. Extending classes with operations[2]

Once defined, the object-oriented way to extend a class's functionality is to derive a new class and introduce the new behavior there. However, this technique only succeeds if the programmer has control over the source code that instantiates objects (for example, if the code has been designed to use a factory). Consider a system framework that responds to various events. The events may require logging in different logs and formats. While it is feasible to provide a common interface for different kinds of logs, it is rather difficult to foresee all possible formats in which logging can be done. Open-methods eliminate the need to modify class declarations directly, and improve the support for separation of concerns.

```
struct Log                  {};                      // Interface to different logs
struct FileLog      : Log   {};                      // Logs to File
struct EventLog     : Log   {};                      // Logs to OS event log
struct DebugLog     : Log   {};                      // Logs to debug output

struct Event                { virtual ~Event(); };   // Interface for various types of events
struct Access       : Event {};
struct FileAccess   : Access {};
struct DirectoryAccess : Access {};
struct DatabaseAccess : Access {};
```

```
// Specializations of how to log various types of events in text format
void log_as_text   (virtual Access&          evt, Log& log, int priority );
void log_as_text   (virtual DirectoryAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in XML format
void log_as_xml    (virtual Access&          evt, Log& log, int priority );
void log_as_xml    (virtual DirectoryAccess& evt, Log& log, int priority );

// Specializations of how to log various types of events in binary format
void log_as_binary (virtual Access&          evt, Log& log, int priority );
void log_as_binary (virtual DirectoryAccess& evt, Log& log, int priority );

// etc. for any other formats that may be required in the future.
```

2.9. Open-methods programming

The benefits noted in the examples stem from fundamental advantages offered by open-methods. They allow us to approximate widely accepted programming principles better than more conventional language constructs and allow us to see conventional solutions, such as visitors, as work-around techniques. For examples like the ones presented above, open-methods simply express the design more directly. From a programmer's point of view, open-methods

- *are non-intrusive*: We can provide runtime dispatch (a virtual function) for objects of a class without modifying the definition of that class or its derived classes. Using open-methods implies less nonessential coupling than conventional alternatives.
- *provide order independence for arguments*: The rules for an argument are independent of whether it is the first, second, third, or whatever argument. The first argument of a member function (the this pointer) is special only in that it has notational support. The dynamic/static resolution choice has become independent of the choice of argument order.
- *improve ambiguity detection*: Ambiguous calls are detected in a way that cannot be done with only a per-argument check (as for conventional multiple dynamic dispatch) or only a per-translation-unit check (as for conventional static checking). Using open-methods there is simply more information for the compiler and linker to use.
- *provide multiple dynamic dispatch*: We can directly select a function based on multiple dynamic types; no work-arounds are required.
- *improve performance*: faster than work-arounds when more than one dynamic type is involved with no memory overhead compared to popular work-around techniques (see Section 8).

From a language design point of view, open-methods make the rules for overriding and overloading more orthogonal. This simplifies language learning, programming, reasoning about programs, and maintenance.
Potential objections to the use of open-methods include that:

- the set of operations on objects of a class are not defined within the class. However, that is true as soon as you allow any free-standing function and is essential for conventional mathematical notation and programming styles based on that. Information hiding is not affected.
- the first argument is not fundamentally different so open-methods do not obey the "send a message to an object" ("object-oriented") model of programming. We consider that model unrealistically restrictive for many application domains, such as classical math [51].
- the set of overriders for a virtual function is not found within a specific set of classes (the set of classes derived from the class that introduced the virtual function). On the other hand, we never have to define a new derived class just to be able to override.
- open-methods are open; that is, they do not provide a closed set of overloading candidates for a given function name. However, we consider that a good feature in that it allows for non-intrusive extension. For C++, the decision not to syntactically distinguish overriders or overloaded functions was taken in 1983 and cannot be changed now [51, Section 11.2.4].

Obviously, we consider open-methods a significant net gain compared to alternatives, but the final proof (as far as proofs are possible when it comes to the value of programming language features) will have to wait for the application of open-methods in several large real-world programs.

3. Definition of open methods

Open-methods are dynamically dispatched functions, where the callee depends on the dynamic type of one or more arguments. ISO C++ supports compile-time (static) function overloading on an arbitrary number of arguments and runtime (dynamic) dispatch on a single argument. The two mechanisms are orthogonal and complementary. We define open-methods to generalize both, so our language extension must unify their semantics. Our dynamic call resolution mechanism

is modeled after the overload resolution rules of C++. The ideal is to give the same result as static resolution would have given had we known all types at compile time. To achieve this, we treat the set of overriders as a viable set of functions and choose the single most specific method for the actual combination of types.

We derive our terminology from virtual functions: a function declared virtual in a base class (super class) can be overridden in a derived class (sub class):

Definition 1. An *open-method* is a free-standing function with one or more parameters declared virtual.

Definition 2. An open-method f_2 *overrides* an open-method f_1 if it has the same name, the same number of parameters, covariant virtual parameter types, invariant non-virtual parameter types, and a possibly covariant return type. In such a case, we say that f_2 is an *overrider* of f_1.

Definition 3. An open-method that does not override another open-method is called a *base-method*.

Definition 4. A base-method together with all the open-methods that override it forms an *open-method family*.

While this is not strictly necessary, for practical reasons we require that a base-method should be declared before any of its overriders. This parallels other C++ rules and greatly simplifies compilation. This restriction does not prevent us from declaring different overriders in different translation units. For every overrider and base-method pair, the compiler checks if the exception specifications and covariant return type (if present) comply with the semantics defined for virtual functions.

Definition 5. A *Dispatch table* (DT) maps the type-tuple of the base-method's virtual parameters to actual overriders that will be called for that type-tuple.

The following example demonstrates a simple class hierarchy and an open-method defined on it:

```
struct A { virtual ~A(); } a;
struct B : A {} b;

void print(virtual A&, virtual A&); // (1) base—method
void print(virtual B&, virtual A&); // (2) overrider
void print(virtual B&, virtual B&); // (3) overrider
```

Here, both (2) and (3) are overriders of (1), allowing us to resolve calls involving every combination of A's and B's. For example, a call print(a,b) will involve a conversion of b to an A& and invoke (1). This is exactly what both static overload resolution and double dispatch would have done.

To introduce the role of multiple inheritance, we can add to that example:

```
struct X { virtual ~X(); };
struct Y : X, A {};

void print(virtual X&, virtual X&); // (4) base—method
void print(virtual Y&, virtual Y&); // (5) overrider
```

Here (4) defines a new open-method print on the class hierarchy rooted in X. Y inherits from both A and X, and according to our definition (5) overrides both (4) and (1).

We note that whether it would be better to require an overrider to be explicitly specified as such is an orthogonal decision beyond the scope of this paper. Here we simply follow the C++ tradition set up by virtual functions to do this implicitly.

3.1. Type checking and call resolution of open-methods

Type checking and resolving calls to open-methods involves three stages: compile time, link time, and runtime.

- *Overload resolution at compile time*: the goal of overload resolution is to find a unique open-method in the overload set visible at the call site, through which the call can be (but not necessarily will be) dispatched. The open-method determines the necessary casts of the arguments, and the return type expected at the call site.
- *Ambiguity resolution at link time*: the pre-linker aggregates all overriders of a given open-method family, checks them for return type consistency, performs ambiguity resolution, and builds the dispatch tables.
- *Dynamic dispatch at runtime*: the dispatch mechanism looks up the entry in the dispatch table that contains the most specific overrider for the dynamic types of the arguments and invokes that overrider.

This three-stage approach parallels the resolution to the equivalent modular-checking problem for template calls using concepts in C++0x [31]. Further, the use of open-methods (as opposed to ordinary virtual functions and multi-methods) can be seen as adding a runtime dimension to generic programming [7].

3.2. Overload resolution

The purpose of overload resolution in the context of open multi-methods is to identify an open-method that the compiler will use for type checking and inferring the result type expected from the call. In general, the C++ overload resolution rules [34] remain unchanged: the viable set includes both open-methods and regular functions and the compiler treats them equally. Once a unique best match is found, the call can be type checked against it. For the dispatch, any of its base-methods can be chosen. Which one is selected is irrelevant as any further overrider would likewise override all base-methods.

Consider the following example:

```
struct X { virtual ~X(); };
struct Y { virtual ~Y(); };
struct Z { virtual ~Z(); };

void foo(virtual X&,   virtual Y&); // (1) base—method
void foo(virtual Y&,   virtual Y&); // (2) base—method
void foo(virtual Y&,   virtual Z&); // (3) base—method

struct XY : X, Y {} xy;
struct YZ : Y, Z {} yz;

void foo(virtual XY&, virtual Y&);   // (4) overrides 1 and 2
void foo(virtual Y&,   virtual YZ&); // (5) overrides 2 and 3
```

A call foo(xy,yz) is ambiguous according to the standard overload resolution rules as overriders 4 and 5 are equally good matches. To resolve this ambiguity, a user may explicitly cast some or all of the arguments to make the call unambiguous accordingly to the overload resolution rules: e.g. calling foo(xy,**static_cast**<Y&>(yz)) will uniquely select 4 as a base-method for the call. Alternatively, a user may introduce a new overrider **void** foo(**virtual** XY&, **virtual** YZ&), which will become a unique best match for the call.

3.3. Ambiguity resolution

Once we are in the ambiguity resolution phase done by the pre-linker, we assume that the overload resolution phase has selected a unique best match for type checking of each open-method call site (otherwise it would have reported a compile-time error). At this phase we have information about all available overriders of a particular open-method family, and we only report ambiguities that prevent us from building a complete dispatch table.

C++ supports single, repeated, and virtual inheritance:

Note that to distinguish repeated and virtual inheritance, this diagram represents sub-object relationships, not just sub-class relationships. We must handle all ambiguities that can arise in all these cases. By "handle", we mean resolve or detect as errors.

Our ideal for resolving open-method calls combines the ideals for virtual functions and overloading:

- virtual functions: the same function is called regardless of the static types of the arguments at the call site.
- overloading: a call is considered unambiguous if (and only if) every parameter is at least as good a match for the actual argument as the equivalent parameter of every other candidate function and that it has at least one parameter that is a better match than the equivalent parameter of every other candidate function.

This implies that a call of a single-argument open-method is resolved equivalently to a virtual function call. The rules described in this paper closely approximate this ideal. As mentioned, the static resolution is done exactly according to the usual C++ rules. The dynamic resolution is presented as the algorithm for generating dispatch tables in Section 3.5. Before looking at that algorithm, we present some key motivating examples.

3.3.1. Single inheritance

In object models supporting single inheritance (Section 3.3), ambiguities can only occur with open-methods taking at least two virtual parameters. Such ambiguities can only be introduced by new overriders, not by extending the class hierarchy. They can be resolved by introducing a new overrider. Open-methods with one dynamic argument are identical to virtual functions and are always ambiguity free. Thus, open-methods provide an unsurprising mechanism for expressing non-intrusive ("external") polymorphism. This eliminates the need to complicate a class hierarchy just to support the later addition of additional "methods" in the form of visitors.

3.3.2. Repeated inheritance

Consider the repeated-inheritance case (Section 3.3) together with this set of open-methods visible at a call site to foo(d1,d2), where d1 and d2 are of type D&:

```
void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
```

Even though overriders for all possible combinations of B and C (the base classes of D) are declared, the call with two arguments of type D gets rejected at compile time. The problem in this case is that there are multiple sub-objects of type A inside D.

To resolve that conflict, a user can either add an overrider foo(D&,D&) visible at the call site or explicitly cast arguments to either the B or C sub-object. Making an overrider for foo(D&,D&) available at the call site eliminates the need to choose a sub-object. It would always be dispatched to the same overrider.

If the (B,C)-vs.-(C,B) conflict is resolved by casting, a question remains on how the linker should resolve a call with two arguments of type D. We know at runtime (by looking into the virtual function table's open-method table (see Section 6)) which "branch" of a D object (either B or C) is on. Thus, we can fill our dispatch table appropriately; that is, for each combination of types, there is a unique "best match" according to the usual C++ rules:

	A	**B**	**C**	**D/B**	**D/C**
A	AA	AA	AA	AA	AA
B	AA	BB	BC	BB	BC
C	AA	CB	CC	CB	CC
D/B	AA	BB	BC	BB	BC
D/C	AA	CB	CC	CB	CC

This depicts the dispatch table for the repeated-inheritance hierarchy in Section 3.3 and the set of overriders above. Since the base method is foo(A&,A&) and A occurs twice in D, each dimension has two entries for D: D/B means "D along the B branch". This resolution exactly matches our ideals.

Analogously to single inheritance, extending a class hierarchy using repeated inheritance cannot introduce ambiguities. Ambiguous sub-objects are determined at compile time and reported as errors.

3.3.3. Virtual inheritance

Consider the virtual-inheritance class hierarchy from Section 3.3 together with the set of open-methods from Section 3.3.2: In contrast to repeated inheritance, a D has only one A part, shared by B, C, and D. This causes a problem for calls requiring conversions, such as foo(b,d); is that D to be considered a B or a C? There is not enough information to resolve such a call. Note that the problem can arise in such a way that we cannot catch it at compile time, because D's definition could be in a different translation unit:

```
C& rc = d;
foo(b,rc);
B& rb = d;
foo(b,rb);
```

Using static type information to resolve either call would violate the fundamental rule for virtual function calls: use runtime type information to ensure that the same overrider is called from every point of a class hierarchy. At runtime, the dispatch mechanism will (only) know that we are calling foo with a B and a D. It is not known whether (or when) to consider that D a B or a C. Based on this reasoning (embodied in the algorithm in Section 3.5) we must generate this dispatch table:

	A	**B**	**C**	**D/A**
A	AA	AA	AA	AA
B	AA	BB	BC	??
C	AA	CB	CC	??
D/A	AA	??	??	??

We cannot detect the ambiguities marked with ?? at compile time, but we can catch them at link time when the entire set of classes and overriders is known.

3.4. Covariant return types

Covariant return types are a useful element of C++. If anything, they appear to be more useful for operations with multiple arguments than for single-argument functions. Covariant return types complicate the use of work-around techniques (Section 9.2).

As an example for using covariant return type, consider a class Symmetric derived from Matrix:

```
Matrix&    operator+(Matrix&,    Matrix&);
Symmetric& operator+(Symmetric&, Symmetric&);
```

It follows that we must generalize the covariant return rules for open-methods. Doing so turns out to be useful because covariant return types help resolve ambiguities.

In single dispatch, covariance of a return type implies covariance of the receiver object. Consequently, covariance of return types for open-methods implies an overrider–base-method relationship between two open-methods. Liskov's substitution principle [36] guarantees that any call type-checked based on a base-method can use overrider's covariant result without compromising type safety.

This can be used to eliminate what would otherwise have been ambiguities. Consider the class hierarchies A ← B ← C and R1 ← R2 ← R3 together with this set of open-methods:

```
R1∗ foo(virtual A&, virtual A&);
R2∗ foo(virtual A&, virtual B&);
R3∗ foo(virtual B&, virtual A&);
```

A call foo(b,b) appears to be ambiguous and the rules outlined so far would indeed make it an error. However, choosing R2∗ foo(A&,B&) would throw away information compared to using R3∗ foo(B&,A&): an R3 can be used wherever an R2 can, but R2 cannot be used wherever an R3 can. Therefore, we prefer a function with a more derived return type and for this example get the following dispatch table:

	A	**B**	**C**
A	AA	AB	AB
B	BA	BA	BA
C	BA	BA	BA

At first glance, this may look useful, but ad hoc. However, a closer look reveals that one of the choices is simply not type safe: a call to foo(b,b), type-checked against R3∗ foo(B&,A&) at compile time, would expect a pointer to an object of type R3 (or any of its sub-classes) returned, which R2 is not. This is why R2∗ foo(A&,B&) cannot be used for dispatching such a call. On the other hand, the same call type-checked against R2∗ foo(A&,B&) elsewhere is expecting a pointer to R2 (or any of its sub-classes) returned from the call, and hence would readily accept R3. This is why selecting R3∗ foo(B&,A&) is the only viable choice here, which consequently resolves the ambiguity.

From a pure implementational point of view, an open-method with a return type that differs from its base-method becomes a new base-method and requires its own dispatch table (or equivalent implementation technique). The fundamental reason is the need to adjust the return type in calls. Obviously, the resolutions for this new base-method must be consistent with the resolution for its base-method (or we violate the fundamental rule for virtual functions). However, since R2∗ foo(A&,B&) will not be part of R3∗ foo(B&,A&)'s dispatch table, the only consistent resolution is the one we chose.

If the return types of two overriders are siblings, then there is an ambiguity in the type-tuple that is a meet of the parameter type-tuples. Consider for example that R3 derives directly from R1 instead of R2, then none of the existing overriders can be used for ⟨B, B⟩ tuple as its return type on the one hand has to be a subtype of R2 and on the other a subtype of R3. To resolve this ambiguity, the user will have to provide explicitly an overrider for ⟨B, B⟩, which must have the return type derived from both R2 and R3.

Using the covariant return type for ambiguity resolution also allows the programmer to specify preference of one overrider over another when asymmetric dispatch semantics is desired.

To conclude: covariant return types not only improve static type information, but also enhance our ambiguity resolution mechanism. We are unaware of any other multi-method proposal using a similar technique.

3.5. Algorithm for dispatch table generation

Let us assume we have a multi-method $rf(h_1, h_2, \ldots, h_k)$ with k virtual arguments. Class h_i is a base of the hierarchy of the ith argument. $H_i = \{c : c <: h_i\}$ is a set of all classes from the hierarchy rooted at h_i. $X_f = H_1 \times H_2 \times \cdots \times H_k$ is the set of all possible argument type-tuples of f. Set $Y_f = \{\langle y_1, y_2, \ldots, y_k\rangle\} \subseteq X_f$ is the set of argument type-tuples, on which the user defined overriders f_j for f. The set $O_f = \{f_0, \ldots, f_{m-1}\}$ is the set of those overriders ($f_0 \equiv f$). $R = \{r_i | r_i f_i(y_1, y_2, \ldots, y_k)\}$ is the set of return types of all the overriders. A mapping $F_f : Y_f \leftrightarrow O_f$ is a bijection between type-tuples on which overriders are defined and the overriders themselves. A function $R_f : Y_f \leftrightarrow R$ maps an argument tuple of an overrider to the return type of that overrider.

Because different derivation paths may get different entries in the dispatch table, we assume that x_i in the type-tuple $x = \langle x_1, \ldots, x_k \rangle$ identifies not only the concrete type, but also a particular derivation path for it (see [56] for formal definitions). Under this assumption, we define $\beta(x_i)$ to be a direct ancestor (base-class) of x_i in the derivation path represented by x_i. For example, for the repeated-inheritance hierarchy from Section 3.3, $\beta(\mathsf{D/B}) = \mathsf{B}$, $\beta(\mathsf{D/C}) = \mathsf{C}$, $\beta(\mathsf{C}) = \mathsf{A}$, while for the virtual- inheritance hierarchy $\beta(\mathsf{D/A}) = \mathsf{A}$, $\beta(\mathsf{B}) = \mathsf{A}$, $\beta(\mathsf{C}) = \mathsf{A}$.

For the sake of convenience, we define:

$$\beta_i(x) \equiv \langle x_1, \ldots, \beta(x_i), \ldots, x_k \rangle, \quad \text{when } \beta(x_i) \text{ exists.}$$

With it, we extend the definition of β to type-tuples as follows:

$$\beta(x) \equiv \{\beta_i(x) \mid \beta_i(x) \text{ exists}, i = 1, k\}.$$

$P(X_f, <_P) : \langle x_1, \ldots, x_k \rangle <_P \langle y_1, \ldots, y_k \rangle \Leftrightarrow \forall i : x_i <: y_i \wedge \exists j : y_j \not<: x_j$ defines a partial ordering that models the ordering of viable functions for overload resolution as defined in [34]. $most_specific_arg(S) = \{s \in S \subseteq Y_f : \nexists t \in S : t <_P s\}$ is the set of the most specific (refined) argument tuples of S with respect to the partial ordering P. $most_specific_res(S) = \{s \in S \subseteq Y_f : \nexists t \in S : R_f(t) <: R_f(s)\}$ is the set of the most specific (refined) argument tuples of S with respect to sub-classing relation $<:$ on result types.

Dispatch table DT_f is a mapping $DT_f : X_f \rightarrow Y_f$ that maps all possible argument tuples to the argument tuples of overriders used for handling such a call.

For any combination of argument types $x \in X_f$, we recursively define entries of the dispatch table DT_f as follows:

$$DT_f[x] = \begin{cases} x, x \in Y_f \\ DT_f[s], s \in S = most_specific_res(most_specific_arg(\{DT_f(y) | y \in \beta(x)\})) \wedge |S| = 1 \\ \text{Ambiguity}, \quad \text{otherwise.} \end{cases}$$

The above recursion exhibits optimal substructure and has overlapping sub-problems, which lets us use dynamic programming [23] to create an efficient algorithm for generation of the dispatch table, as shown in Algorithm 1.

To demonstrate with an example, consider a simple class hierarchy with two classes A and B, where B derives from A, and two open-methods defined on the argument tuples $\langle \mathsf{A}, \mathsf{A} \rangle$ and $\langle \mathsf{A}, \mathsf{B} \rangle$. In this scenario $X_f = \{\langle \mathsf{A}, \mathsf{A} \rangle, \langle \mathsf{B}, \mathsf{A} \rangle, \langle \mathsf{A}, \mathsf{B} \rangle, \langle \mathsf{B}, \mathsf{B} \rangle\}$ with the following relations that hold on these argument tuples: $\langle \mathsf{A}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle$; $\langle \mathsf{B}, \mathsf{A} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle$; $\langle \mathsf{B}, \mathsf{B} \rangle <_P \langle \mathsf{B}, \mathsf{A} \rangle$; $\langle \mathsf{B}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{B} \rangle$; $\langle \mathsf{B}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle$;

The reverse topological order of elements in X_f would thus match the order in which we listed tuples in X_f. Sets of immediate ancestors with respect to $<_P$ would be:

$$\beta(x) = \begin{cases} \emptyset, x = \langle \mathsf{A}, \mathsf{A} \rangle \\ \{\langle \mathsf{A}, \mathsf{A} \rangle\}, x = \langle \mathsf{A}, \mathsf{B} \rangle \\ \{\langle \mathsf{A}, \mathsf{A} \rangle\}, x = \langle \mathsf{B}, \mathsf{A} \rangle \\ \{\langle \mathsf{A}, \mathsf{B} \rangle, \langle \mathsf{B}, \mathsf{A} \rangle\}, x = \langle \mathsf{B}, \mathsf{B} \rangle. \end{cases}$$

Note that the empty set of immediate ancestors is only possible on the tuple that starts the open-method hierarchy, where we by definition would always have an overrider—the base-method. A set of argument tuples of overriders $Y_f = \{\langle \mathsf{A}, \mathsf{A} \rangle, \langle \mathsf{A}, \mathsf{B} \rangle\}$ and thus we can directly set $DT_f[\langle \mathsf{A}, \mathsf{A} \rangle] = \langle \mathsf{A}, \mathsf{A} \rangle$ and $DT_f[\langle \mathsf{A}, \mathsf{B} \rangle] = \langle \mathsf{A}, \mathsf{B} \rangle$.

Now to fill in $DT_f[\langle \mathsf{B}, \mathsf{A} \rangle]$, where $\langle \mathsf{B}, \mathsf{A} \rangle$ is the first element in reverse topological order of X_f that is not in Y_f, we take the set of its immediate ancestors $\beta(\langle \mathsf{B}, \mathsf{A} \rangle) = \{\langle \mathsf{A}, \mathsf{A} \rangle\}$, and since there is only one, there cannot be a better match and thus $DT_f[\langle \mathsf{B}, \mathsf{A} \rangle] \leftarrow DT_f[\langle \mathsf{A}, \mathsf{A} \rangle] = \langle \mathsf{A}, \mathsf{A} \rangle$.

Similarly, to fill in the remaining $DT_f[\langle \mathsf{B}, \mathsf{B} \rangle]$ that comes last in the reverse topological order, we look at its immediate ancestors $\beta(\langle \mathsf{B}, \mathsf{B} \rangle) = \{\langle \mathsf{A}, \mathsf{B} \rangle, \langle \mathsf{B}, \mathsf{A} \rangle\}$ and compare the overriders used for them: $DT_f[\langle \mathsf{A}, \mathsf{B} \rangle] = \langle \mathsf{A}, \mathsf{B} \rangle <_P \langle \mathsf{A}, \mathsf{A} \rangle = DT_f[\langle \mathsf{B}, \mathsf{A} \rangle]$. Thus we propagate the most specific overrider: $DT_f[\langle \mathsf{B}, \mathsf{B} \rangle] \leftarrow DT_f[\langle \mathsf{A}, \mathsf{B} \rangle] = \langle \mathsf{A}, \mathsf{B} \rangle$.

To analyze its performance, we first note that comparison of two type-tuples from X_f can be done in time $O(k)$. If $n = max(|H_i|, i = 1, k)$ and $v = max(v_i, i = 1, k)$ (where v_i is a maximum number of times h_i is used as non-virtual base class in any class of hierarchy H_i) then $|X_f| <= (n * v)^k$ and the amount of edges for topological sort is less then $k * (n * v)^k$. Therefore the complexity of topologically sorting X_f is $O(k * n^k)$. The inner for-loop has complexity $O(k^2 * n^k)$ so the overall complexity is $O(n^k)$ since k is a constant defining the amount of virtual arguments. This means that the algorithm is linear in the size of the dispatch table.

3.6. Alternative dispatch semantics

While our goal is to unify virtual function dispatch and overload resolution into an open-methods semantics, this is not always possible. Consider for example the repeated-inheritance class hierarchy from Section 3.3 with a virtual function added:

Algorithm 1 Dispatch Table Generation

```
T ← topological_sort(X_f) // Topologically sort according to <_P
S ← reverse(T) // Reverse the order to have the least specific first
for all x ∈ S do
  if x ∈ Y_f then
    DT_f[x] ← x // Overriders themselves are the best matches for their arguments
  else
    ancestors = β(x) // Get type-tuples of immediate ancestors
    most_specific = {DT_f[extract_any(ancestors)]}
    while ¬ empty(ancestors) do
      a ← extract_any(ancestors)
      dominated ← false
      for all e ∈ most_specific do
        if DT_f[a] <_P DT_f[e] then
          // This ancestor's overrider is more specific
          most_specific ← most_specific − {e}
        else if DT_f[e] <_P DT_f[a] then
          // Overrider in the most_specific set is more specific
          dominated ← true
          break
        else if R_f(DT_f[a]) <: R_f(DT_f[e]) then
          // Incomparable by arguments, but more specific return type
          most_specific ← most_specific − {e}
        else if R_f(DT_f[e]) <: R_f(DT_f[a]) then
          // Incomparable by arguments, but ancestor's return type is less specific
          dominated ← true
          break
        end if
      end for
      if ¬ dominated then
        // When none of the overriders was more specific
        most_specific ← most_specific ∪ {DT_f[a]}
      end if
    end while
    if |most_specific| = 1 then
      // There was a unique most specific overrider, use it
      DT_f[x] ← y, where most_specific = {y}
    else
      Error: Unable to find unique best overrider among most_specific for handling x
    end if
  end if
end for
```

```
struct A        { virtual void foo(); }; // virtual function
struct B : A    { };
struct C : A    { virtual void foo(); }; // virtual function
struct D : B, C { };

void bar(A&);              // overloaded function
void bar(C&);              // overloaded function

void foobar(virtual A&);   // open−method
void foobar(virtual C&);   // open−method

D d;
B& db = d;                 // B part of D
C& dc = d;                 // C part of D

// (runtime) Virtual Member Function Semantics:
db.foo();                  // calls A::foo
dc.foo();                  // calls C::foo
d.foo();                   // error: ambiguous
```

```
// (compile time) Overload Resolution Semantics:
bar(db);          // calls bar(A&)
bar(dc);          // calls bar(C&)
bar(d);           // calls bar(C&) (why not ambiguous?)

// (runtime) open—method Semantics:
foobar(db);       // calls foobar(A&)
foobar(dc);       // calls foobar(C&)
foobar(d);        // error: ambiguous
```

Virtual dispatch semantics and overload resolution semantics go different ways in this case. Since the two language features are not entirely orthogonal, we had to decide which semantics to follow.

From a technical point of view, both semantics can be implemented for open multi-methods. The reason we decided not to model the semantics after overload resolution in this case is that the resulting cross-casting behavior could have been surprising to the user due to the implicit switching of different sub-objects. On the other hand, the difference between the ordinary virtual function (foo) call and the ordinary overloaded resolution for (bar) in this case is odd and depends on pretty obscure rules that may be more historical than fundamental. Calls to the open-method foobar follow the virtual function resolution. This is why our open-method semantics strictly corresponds to virtual member function semantics in ISO C++ but does not entirely reflect overload resolution semantics. The reason is that less information is available for compile-time resolution than for link-time or runtime resolution. For example, the resolution of **static_cast** and **dynamic_cast** can differ even given identical arguments: **dynamic_cast** can use more information than **static_cast**.

Due to our decision to model the semantics after virtual dispatch, we require covariance of the return type on overriders, while had we modeled after overload resolution, we could have only required convertibility of return types.

4. Discussion of design decisions

Type-safety and ambiguities have always been a major concern to systems with multiple open dispatch. One of the first widely known languages to support open-methods was CLOS [50]. CLOS linearizes the class hierarchy and uses asymmetric dispatch semantics to avoid ambiguity. Snyder [49] and Chambers [16,17] observe that silent ambiguity resolution makes errors in programs hard to spot. Therefore, Cecil uses symmetric dispatch semantics and dispenses with object hierarchy linearization in order to expose these errors at compile time. Recent studies [4,27,39,41] explore the trade-offs between multi-methods and modular type-checking in languages with neither a total order of classes nor asymmetric dispatch semantics. In particular, Millstein and Chambers discuss a number of models that embrace or restrict the expressive power of the language to different degrees. The described models range from globally type-checked programs to modularly type-checked units. We will briefly discuss our evaluation of these approaches in the context of C++ later in this section.

This work aims for maximal flexibility and relies on a global type-checking [1] approach for open-methods. We motivate this approach with the goal not only to support object-oriented programming but also to enhance the support for functional and generic programming styles in C++.

The cost of the global type-checking approach is that some ambiguities can be detected late—in particular at the load time of dynamically linked libraries (DLLs). DLLs are almost universally used with C++; thus a design for open-methods that does not allow for DLLs is largely theoretical. We do not currently have an implementation supporting dynamic linking, but we outline a design addressing the major issues in such a scenario.

Our guiding principle is to support the use cases described in Section 2 with language features that are guaranteed to be type-safe in every scenario. The idea is to report errors as long as we can assume that ambiguities can be resolved by programmers. Only when it is too late for that do we have to use type-safe resolution mechanisms. This section discusses the design decisions we have made based on three language aspects: ambiguity resolution, covariant return types, and pure (abstract) open-methods.

4.1. Late ambiguities

Late ambiguities are ambiguities that are detected at a stage in the build process when programmer intervention is no longer feasible. They can occur, for example, when classes use virtual inheritance while some definitions necessary to declare a resolving overrider cannot be accessed. Consider the example given in Section 3.3.3. Examples for late ambiguities include:

- the class D was defined as a local class, since the class name would be local to the function scope.
- the class D was defined in an implementation file of a library, but the class definition was not exported in a header file.
- a library defined classes A, B, and C as well as implemented, but did not export, an open-method foo. The definition of D results in a late ambiguity.

In all cases, a programmer could not declare a resolving overrider.

A second source of late ambiguities is when independently developed libraries define conflicting overriders, but the definition of one of the involved classes is not available. Consider the single-inheritance hierarchy of Section 3.3 with an open-method foo(A,A). A library defines, but does not export B and an overrider for foo(B, A), while another library defines C and an overrider for foo(A,C). A call foo(b,c) is ambiguous but cannot be resolved, because the definition of B is not available. Ambiguities that emerge from the use of dynamically linked libraries are always late.

Resolution mechanism for late ambiguities: If there is no unique best match for a possible type-tuple, we choose an overrider from all best matches. Interestingly, any overrider will result in a correct program provided the rest of the program is correct and in principle we could even pick a random overrider from the set of best matches. Nevertheless, the choice is deterministic, but remains unspecified.

Not to specify which overrider we choose among type-safe candidates keeps the resolution mechanism symmetric as no candidate is preferred. The use of a deterministic choice is not strictly necessary, but it allows for reproducibility—the same method will always be selected from a set of candidates.

Consider the following example of image format conversion. For a discussion of the problem and an implementation see Section 2.2 and Section 9.1, respectively. The following code shows a common header file and two independently developed libraries that support additional image formats.

```
// Common header: ImageLibrary.h
struct Image {
  virtual ~Image();
  // ...
};
struct TiffImage : Image { /* ... */ };

void convert(virtual const Image&     from, virtual Image&     to) {... }
void convert(virtual const TiffImage& from, virtual Image&     to) { ... }
void convert(virtual const Image&     from, virtual TiffImage& to) { ... }

// DLL—Jpeg supporting JPEG images
#include "ImageLibrary.h"
struct JpegImage : Image { /* ... */};
void convert(virtual const Image&     from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage& from, virtual Image&     to) { ... }
void convert(virtual const TiffImage& from, virtual JpegImage& to) { ... }
void convert(virtual const JpegImage& from, virtual TiffImage& to) { ... }

// DLL—Png supporting PNG images
#include "ImageLibrary.h"
struct PngImage : Image { /* ... */};
void convert(virtual const PngImage& from, virtual Image&     to) { ... }
void convert(virtual const Image&    from, virtual PngImage&  to) { ... }
void convert(virtual const TiffImage& from, virtual PngImage& to) { ... }
void convert(virtual const PngImage& from, virtual TiffImage& to) { ... }
```

The header file of an image library framework defines two classes (Image, TiffImage), and a base-method convert together with two overriders that implement conversions from TiffImage to a general Image and vice versa. A library (DLL-Jpeg) derives a new type JpegImage from Image and introduces new overriders for convert that handle all possible combinations of known image formats. Likewise, another library (DLL-Png) derives a new class PngImage from Image and introduces similar overriders. Now a call to convert a JpegImage into a PngImage is ambiguous. Libraries DLL-Jpeg and DLL-Png could stem from different vendors that do not know about each other. In systems that use dynamically linked libraries, such problems are hard to predict and design for.

Note that, since the class definitions of the respective other library were not available when DLL-Png and DLL-Jpeg were implemented, neither developer could possibly provide resolving overriders. The question thus arises to which convert should a call convert(JpegImage, PngImage) resolve.

Any overrider (including base-method) has to assume that a dynamic type resolving to Image is an unknown derived type. Consequently, each convert must be written so that it manipulates its arguments of types Image polymorphically (for example, by using virtual functions). This implies that as long as convert's code does not make more assumptions about its arguments than the interface defined in the base-class guarantees, any overrider can be chosen.

Alternative techniques to handle or prevent (late) ambiguities include asymmetric choice, preventive elimination of overriders that could be prone to symmetry, or exceptions that signal an error:

- *System specified choice:* Other systems with open-methods use a specified policy to resolve ambiguities. These involves preferred treatment of overriders that are more specialized on a specified argument (e.g. CLOS [50]) and class hierarchy linearization (CLOS, Dylan [47]). Making the resolution explicit breaks symmetric dispatch, as programmers can write code that exploits the specification.
- *Limit extensibility:* In [39,41], Millstein and Chambers discuss limitations to the type systems that prevent late ambiguities. Their system *M* disallows virtual inheritance across modules. Moreover, open-methods have a specified argument position. Adding overriders across module boundaries is permitted only when the type in that argument is covariant and the type is defined in the same module. MultiJava [20] is based on system *M*. In practice, these limitations have

been found to be overly restrictive (Relaxed MultiJava [21,40] and C++ concepts [35]). In addition, requiring C++ code to comply with the provided inheritance restrictions is not an option.

In [4], Allen et al. develop a different set of restrictions for modular type checking of multiple dispatch for Fortress [3]. Instead of restricting multiple inheritance across modules, the notion of a meet function resolves ambiguities that originate from virtual inheritance. Moreover, their set of restrictions is sensitive to whether a function is a multi-method (defined in class) or an open-method (freestanding function). Overriding open-methods across module boundaries is not possible. Like in System *M*, overriding multi-methods is tied to a single distinguished argument position (the self argument) and the module of the type definition.

Systems that require overriders defined in another module to override a specific argument position with a covariant type defined in that module are unable to handle bidirectional image conversion well. Assuming that the first argument is special, DLL-Jpeg could not provide overriders for conversions to JpegImage.

– *User specified choice:* Parasitic methods as implemented in Java [14] (an implementation for Smalltalk also exists [26]) add an object-oriented flavor to multi-methods and make them an integral part of classes. Multi-methods can be inherited from a base class and overridden (or shadowed) in the derived class. Parasitic methods give the receiver precedence over other arguments. The encapsulation guarantees that a compiler can check for multiple argument ambiguities. Virtual-inheritance ambiguities are implicitly resolved by users, as the resolution is sensitive to the order of multi-method declarations within the class definition.

  In [27], Frost and Millstein unify encapsulated multiple dispatch with predicate dispatch. They replace the dependence on textual order with first match semantics, where later predicates implicitly exclude earlier predicates.

  The global checking presented in this paper resolves fewer virtual-inheritance ambiguities silently than an encapsulated approach would. Moreover, the use of encapsulation requires control over the construction of the receiver object. Even if that can be handled by using a factory approach, this would be unable to solve and would only recast the ambiguity illustrated by the conversion example: Which converter class takes precedence, the one defined by DLL-Jpeg or the one defined by DLL-Png?
– *Glue-methods:* Relaxed Multi-Java [40] resolves ambiguity conflicts by introducing glue-methods (to glue DLL-Jpeg and DLL-Png) that the system-integrator provides. This is a viable solution for software developers integrating several libraries, but it is not a feasible scenario for end-user applications, as dynamically linked modules can be loaded into the process without the direct request of a developer. This is the case for various component object models where applications may request an object by name from the system. The operating system will locate and load the module in which the object resides.
– *Throw an exception:* Some implementations (e.g. Cmm [48]) throw an exception at dispatch time when an ambiguity is encountered. We disagree with this approach because each candidate alone is a type-safe choice and should be able to handle the requested operation. Moreover, this approach forces programmers to consider open-method calls as a potential source for exceptions, while their choice of how to handle this exception is limited and likely will result in program termination.
– *Program termination:* Instead of waiting until runtime, the application can terminate (or fail to link) when ambiguous overriders are detected. We argue analogously to the exception case that termination is an inadequate response for a choice among type-safe operations.

4.2. Consistency of covariant return types

Before we go into a detailed discussion, we would like to point out that the main focus of this section is on the consistency of covariant return types among overriders available at runtime. The use of covariant return type for ambiguity resolution is orthogonal to the problems discussed here and is discussed in detail in Section 3.4.

Different DLLs can specify conflicting covariant return types. Consider a two-class hierarchy A ← B and another two-class hierarchy R1 ← R2. The base-method R1 foo(**virtual** A&, **virtual** A&) is defined in a header visible by two dynamically linked modules D_1 and D_2 that do not know anything about each other. Module D_1 introduces overrider R2 foo(A&, B&) and module D_2 introduces overrider R1 foo(B&, B&). Each of the dynamically linked modules perfectly type-checks and links with foo() resolved through the dispatch table (a superscript in a cell denotes the type that is returned by an overrider; e.g. AB^2 denotes R2 foo(A&, B&)):

$AA^1 in D_1$	**A**	**B**	$AA^1 in D_2$	**A**	**B**
A	AA^1	AB^2	**A**	AA^1	AA^1
B	AA^1	AB^2	**B**	AA^1	BB^1

When both libraries are linked together, we get the dilemma of how to resolve a call with both arguments of type B. On the one side foo(B&,B&) from D_2 is more specialized, but on the other side foo(A&,B&) from D_1 imposes the additional requirement that the return type of whatever is called for ⟨B, B⟩ should be a subtype of R2, which R1 is not. Such a scenario would be rejected at compile/link time; however at load time we do not have this option anymore.

Keeping all dispatch tables of a particular open-method consistent on the overrider that will be called for a particular combination of types will force us to choose between suboptimal and type-unsafe alternatives. What is worse is that there may not be a unique type-safe alternative.

Imagine for example that a module D_3 introduces an overrider R3 foo(B&, A&), where R1 ← R3, so R2 and R3 are siblings. When D_1 and D_3 are loaded together, neither R2 foo(A&, B&) nor R3 foo(B&, A&) can be used to resolve a call with both arguments of type B—both alternatives are type-unsafe for the other overrider.

To deal with this subtlety, we propose for the DLL case to weaken the requirement that the same overrider should be called for the same tuple of dynamic types regardless of the static types used at the call site. We require that the same overrider be used only if it is type-safe for the caller. Strictly speaking, R1 foo(B&,B&) is not an overrider of R2 foo(A&, B&) as defined in Section 3, because its return type is not changing covariantly in respect to the types of arguments. Therefore, it cannot be considered for the dynamic resolution of calls made statically through the base-method R2 foo(A&, B&).

Taking the above into account, we propose that the dynamic linker fills in the dispatch table of every base-method independently. This results in:

AA^1	**A**	**B**	AB^2	**B**	BA^3	**A**	**B**
A	AA^1	AB^2	**A**	AB^2			
B	BA^3	BB^1	**B**	AB^2	**B**	BA^3	BA^3

It looks as if the dispatch table for the base-method R1 foo(A&,A&) now violates covariant consistency, but in reality it does not because all the return types in it are cast back through thunks to R1, which is the type statically expected at the call site.

As can be seen, this logic may result in different functions being called for the same type-tuple depending on the base-methods seen at the call site. We note, however, that *the call is always made to the most specialized overrider that is type-safe for the caller*.

4.3. Pure open-methods

There are no abstract (pure virtual) open-methods; that is, every open-method must be defined. Consider a (dynamic) library D1 that introduces a new class and a second (dynamic) library D2 that defines a new abstract open-method. When both libraries are (dynamically) linked together the presence of an overrider for the class in D1 cannot be guaranteed. The alternative would be runtime "method not defined" errors (reported as exceptions), but that solution would be inconsistent with the rest of C++ and would limit the use of open-methods in embedded systems.

5. Relation to orthogonal features

In this section, we discuss the relationship of open-methods to other language features.

5.1. Namespace

Virtual functions have a class scope and can only be overridden in the derived classes. Open-methods do not have such a scope by default, so the question arises: when should an open-method be considered an overrider and when just a different open-method? Let us look at the following example:

```
namespace X
{
  class A {};
  void bar(virtual A&); // base method

  class B : A {};
  void bar(virtual B&); // (1)
}

namespace Z
{
  void bar(virtual B&); // (2)
}

namespace Y
{
  class D : X::A {};
  void bar(virtual D&); // (3)
}

class C : X::A {};
void bar(virtual C&); // (4)
```

In the presented implementation, an overrider has to be declared in the same namespace as its base-method (1). Open-methods with the same name and compatible parameter types, defined in different namespaces, would not be considered

overriders. The major benefit of this approach is that it is easy to understand and implement. Unfortunately such semantics are not unifiable with overrider declarations of virtual function calls, where derived classes can be declared in a different namespace. **using** declarations present a potential work around to these limitations.

An alternative would be to let overriders be declared in any namespace (1, 2, 3, 4). It is easy to understand, but defeats the purpose of namespaces that were introduced to better structure the code and avoid name-clashes among independently developed modules.

Another alternative may consider an open-method to be an overrider, if its base-method is defined in the same scope or in the scope of their argument types and their base classes. In this scenario (1, 3, 4) would override; (2) would not. Among its advantages is that it closely resembles argument dependent lookup. It would also work for virtual functions. Its downside, however, is that it is harder to comprehend.

5.2. Access privileges

Open-methods are generic freestanding functions, which do not have the access privileges of member functions. If an open-method needs access to non-public members of a class, that class must declare it a particular open-method as a friend.

5.3. Smart pointers

In C++, programmers use smart pointers, such as auto_ptr (in current C++) as well as shared_ptr and weak_ptr (in Boost [59] and C++0x [8]) for resource management. The use of smart pointers together with open-methods is no different from their use with (virtual) member functions. For example:

```
struct A { virtual ~A(); };
struct B : A {};
void foo(virtual A&);

void bar(shared_ptr<A> ptr)
{
  foo(*ptr);
}
```

Defining open-methods directly on smart pointers is not possible. In the following example, (1) yields an error, as ptr1 is neither a reference nor a pointer type. The declaration of (2) is an error, because shared_ptr is not a polymorphic object (it does not define any virtual function). Even when shared_ptr were polymorphic, the open-method declaration would be meaningless. A shared_ptr<B> would not be in an inheritance relationship to shared_ptr<A>, thus the compiler would not recognize foo(**virtual** shared_ptr<B>&) as an overrider.

```
void foo(virtual shared_ptr<A>  ptr1); // (1) error
void foo(virtual shared_ptr<A>& ptr2); // (2) error
```

6. Implementation

We have implemented open-methods as described in Section 3 by modifying the EDG compiler front-end [24]. This includes dispatch table generation and thunk generation for multiple inheritance and covariant return. To reduce the dispatch table size, we have also implemented the dispatch table compression techniques presented in [5]. Our current implementation does not support dynamically linked libraries and detection of late ambiguities.

6.1. Changes to the compiler and linker

Our mechanism extends ideas presented in [25,57] as to the compiler and linker model. We adopted the multi-method syntax proposed in [51], which in turn was inspired by an earlier idea by Doug Lea (see [51, Section 13.8]). One or more parameters of a non-static freestanding function can be specified to be **virtual**. Overloading functions based only on the virtual specifier is not allowed.

A virtual argument must be a reference or pointer to a polymorphic class (that is, a class containing at least one virtual function). For example:

```
struct A { virtual ~A(); };

void print(virtual A&);              // ok
void print(int, virtual A*);         // ok
void print(int, virtual const A&); // ok

void dump(virtual A);                // compiler error
void dump(virtual int);              // compiler error
```

Fig. 1. Object model for repeated inheritance.

For each translation unit, the EDG compiler lowers the high level abstractions in C++ to equivalent code in C. We added an implementation that lowers open-method calls to C according to the object-model presented in Section 6.2. In addition, the compiler puts out an *open-method description* (OMD) file that stores the data needed to generate the runtime data structure discussed in Section 6.2. This includes the names of all classes, their inheritance relationships, and the kind of inheritance. Open-methods are represented by name, return type, and their parameter list. Finally, the OMD-file also contains definitions of all user-defined types that appear in signatures of open-methods (both as virtual and regular parameters). These definitions are necessary to generate class definitions for arguments to open-methods that are passed by value.

The pre-linker uses Coco/R [58] to parse the OMD-files. Then, the pre-linker synthesizes the OMD-data, associates all overriders with their base-methods, generates dispatch tables, issues link-errors for ambiguities, determines the indices necessary to access the open-method, and initializes the data structures described in Section 6.2.

When the call of an overrider requires adjustments of the this-pointers (as is sometimes needed in multiple-inheritance hierarchies), the pre-linker creates thunks and makes the dispatch table entries refer to them instead. During dispatch table synthesis, the linker will report errors for all argument combinations that do not have a unique best overrider. The output of the pre-linking stage is a C-source file containing the missing definitions. If the linker generates a library, the pre-linker also puts out a merged OMD-file.

6.2. Changes to the object model

We augment the IA-64 C++ object model [22] by four elements to support constant time dispatching of open-methods. First, for each base-method there will be a dispatch table containing the function addresses. Second, the v-table of each sub-object contains an additional pointer to the *open-method table (om-table)*. Finally, the indices used for the om-table offsets are stored as global variables.

The Figs. 1 and 2 show the layout of objects, v-tables, om-tables and dispatch-tables for repeated and virtual inheritance. Our extensions to the object-model are shown with gray background. From left to right the elements in each diagram represent the object, v-table, om-table, and dispatch table(s) for the class hierarchy in Section 3.3. From top to the bottom, the objects are of type A, B, C, and D, respectively.

An open-method can be declared after the declarations of the classes used in its virtual parameters. Therefore, the compiler cannot reserve v-table entries to store the data related to open-method dispatch immediately in a class's virtual function table. Hence, we always extend every v-table by one pointer referencing the om-table, which can be laid down later by the pre-linker.

The om-table reserves one position for each virtual parameter of each base-method, where objects of this type can be passed as arguments. This position stores an index into the corresponding dimension of the dispatch table. Since the size of

Fig. 2. Object model for virtual inheritance.

the om-tables is not known at compile time, our technique relies on a literal for each open-method and virtual parameter position (called foo_1st, foo_2nd in Figs. 1 and 2) that determines the offset within the om-tables.

Note that these figures depict our actual implementation, where entries for first argument positions already resolve one dimension of the table lookup. Entries for all other argument positions store the byte offset within the table.

In the presence of multiple-inheritance, a this-pointer shift might be required to pass the object correctly. In this case, we replace the address of the overrider by an address of a thunk that takes care of correctly adjusting the this-pointer. As described in Section 3.3.2 in the case of repeated inheritance, different bases can show different dispatch behavior depending on the sub-object to which the this-pointer refers. As a result, different bases may point to different om-tables. In the case of virtual inheritance, the open-method dispatch entries are only stored through the types mentioned in the base-method. Hence, in the virtual-inheritance case, all open-method calls are dispatched through the virtual base type.

6.3. Alternative approaches

We considered a few other design alternatives and explored their trade-offs in extensibility and performance.

6.3.1. Multi-methods

Unlike open-methods, multi-methods require the base-method to be declared in the class definition of its virtual parameters. This allows the offset within the v-table be known at compile time, which saves two indirections per argument of a function call (one for the om-table, and one to read the index within the om-table). For a call with k virtual arguments, open-methods need $4k + 1$, while multi-methods need only $2k + 1$ memory references to dispatch a call. The downside of multi-methods is that existing classes cannot easily be extended with dynamically dispatched functions.

With the restriction of in-class declarations imposed by multi-methods it seems logical to declare a multi-method either as a member function or as a friend non-member function. Consider:

```
class Matrix
{
  // multi—method declaration as a non—member function
  friend Matrix& operator+(virtual const Matrix& lhs, virtual const Matrix& rhs);

  // equivalent declaration as a member function
  virtual Matrix& operator*(virtual const Matrix&);
};
```

We implemented only the non-member version of multi-methods. The member version can be implemented with exactly the same techniques. However, in many cases it is harder to write code that uses the member version because an overrider must be a member of (only) one class—and the main rationale for multi-methods is to elegantly deal with combinations of classes. Even the non-member (friend) version is hard to use.

By requiring a declaration to be present in a class, we limit the polymorphic operations to those that the class designer thought of. That requires too much foresight of the class designer or leads to unstable classes (classes that keep having multi-methods added). Such problems are well known in languages relying on member functions. Open-methods provide an abstraction mechanism that solves such problems by separating operations from classes.

6.3.2. Chinese remainders

As we saw in Section 6.2, support of open-methods required an extra indirection via an om-table to get the index of the class in the appropriate argument position. This extra indirection was needed because open-methods are not bound to the class, and as a result, we do not know how many of them a class may have; therefore we cannot reserve entries in the v-table for them. In this section, we present an "ideal" scheme for implementing open-methods, inspired by ideas presented in [29]. The proposed scheme circumvents the necessity for om-tables by moving all the necessary information from the class to the dispatch table.

Suppose that for every multi-method f there is a function $I_f : T \times N \rightarrow N$ such that for any type $t \in T$ (where T is a domain of all types) and argument position $n \in N$ it returns an index of type t in the nth dimension of the f's dispatch table. If such a function is reasonably fast (preferably constant time) and its range is small (preferably from zero to the maximum number of types that can be used in any argument position) then we can efficiently implement multiple dispatch by properly arranging the rows and columns according to the indices returned by I_f. As in [29], we use the Chinese Remainder theorem [23] to generate the function I_f.

Chinese Remainder Theorem

Let $m_1, \ldots, m_k$ be integers with $gcd(m_i, m_j) = 1$ whenever $i \neq j$. Let m be the product $m = m_1 m_2 \cdots m_k$. Let $a_1, \ldots, a_k$ be integers. Consider the system of congruences:

$$\begin{cases} x \equiv a_1 (\text{mod } m_1) \\ x \equiv a_2 (\text{mod } m_2) \\ \ldots, \\ x \equiv a_k (\text{mod } m_k). \end{cases}$$

Then there exists exactly one $x \in Z_m$ satisfying this system.

Since we may have different class hierarchies in different argument positions, we have to consider each argument position separately. Assuming that there can be q different types $t_{i1}, t_{i2}, \ldots, t_{iq}$ in an argument position i, we may assign a different prime number $m_{ij} : j = 1, q$ to each of them and then according to the Chinese Remainder Theorem find a number x_i that satisfies the above equation. Storing x_i for each dimension (argument position) of the dispatch table, we will come to the dispatching algorithm shown in the listing 2. Since k is known at compile time, no actual iteration is required and the algorithm takes constant time.

Algorithm 2 Dispatching with Chinese Remainders

for all argument positions i of a multi-method f **do**
  $n_i = x_i \mod m_i$
end for
call $D[n_1, \ldots, n_k]$ with arguments provided

In this scenario, every class (or more specifically every argument position i where this class may appear as virtual argument) will have a prime number m_i assigned to it, while the dispatch table will have a number x_i computed through Chinese Remainders, associated with each of its dimensions. The result of $x_i \mod m_i$ gives us the column within the appropriate dimension of the dispatch table.

This dispatching technique has the nice property that it does not need any modifications of the v-table in order to introduce a new open-method on the class. Once allocated, prime numbers can be reused for any number of open-methods defined on the class regardless of the argument position in which a type is used. After dispatch table allocation, we simply have to compute the number x_i for each of the argument positions. Extending such a table, which may be required after the introduction of a new class in the hierarchy, is also simple: allocate new rows and columns and recompute x_i taking the prime numbers of newly added classes into account.

We demonstrate the approach with an example. Consider the following class hierarchy and an open-method foo defined on it:

```
class A            {};          // Assigned prime 2
class B : public A {};          // Assigned prime 5
class C : public A {};          // Assigned prime 3
```

```
class D : public B, public C {};  // Assigned prime 13 for D/B and 7  for D/C sub−object
class E : public D {};            // Assigned prime 17 for E/B and 19 for E/C sub−object

void foo(virtual A&, virtual A&);
void foo(virtual B&, virtual B&);
void foo(virtual B&, virtual C&);
void foo(virtual C&, virtual B&);
void foo(virtual C&, virtual C&);
void foo(virtual E&, virtual E&);
```

The following dispatch table is built:

	A^{2}	**B**^{5}	**C**^{3}	**D/B**^{13}	**D/C**^{7}	**E/B**^{11}	**E/C**^{17}	
A^{2}	AA	AA	AA	AA	AA	AA	AA	$x \equiv 0 (\bmod 2)$
B^{5}	AA	BB	BC	BB	BC	BB	BC	$x \equiv 1 (\bmod 5)$
C^{3}	AA	CB	CC	CB	CC	CB	CC	$x \equiv 2 (\bmod 3)$
D/B^{13}	AA	BB	BC	BB	BC	BB	BC	$x \equiv 3 (\bmod 13)$
D/C^{7}	AA	CB	CC	CB	CC	CB	CC	$x \equiv 4 (\bmod 7)$
E/B^{11}	AA	BB	BC	BB	BC	EE	EE	$x \equiv 5 (\bmod 11)$
E/C^{17}	AA	CB	CC	CB	CC	EE	EE	$x \equiv 6 (\bmod 17)$
	$x \equiv 0 (\bmod 2)$	$x \equiv 1 (\bmod 5)$	$x \equiv 2 (\bmod 3)$	$x \equiv 3 (\bmod 13)$	$x \equiv 4 (\bmod 7)$	$x \equiv 5 (\bmod 11)$	$x \equiv 6 (\bmod 17)$	$x = 1062506$

Dispatching a call will then look like $DT_{foo}[X_{DT_{foo}} \bmod P(a_1), X_{DT_{foo}} \bmod P(a_2)](a_1, a_2)$. Having pointers to the actual arguments of a call, we can look up the v-tables of those arguments and the prime numbers associated with their types. Suppose that the prime number associated with the first argument is 11, while the prime number associated with the second argument is 3. To get the row number inside the dispatch table associated with the first argument, we compute the remainder of dividing 10 62 506 by 11, which is 5. Row number 5 corresponds to the B sub-object of an object with dynamic type E. Similarly, we get the column associated with the type of the second argument through finding the remainder of dividing 10 62 506 by 3, which is 2. Column number 2 corresponds to type C, which means that the dynamic type of the second argument is C. The number 10 62 506 is associated with the dispatch table, through which the call is being dispatched. To find the overrider that will be handling the call, we simply look up an address of the function that is stored at the intersection of the fifth row and the second column, which is foo(B&,C&).

Despite its elegance, this approach is rather theoretical because it is hard to use for large class hierarchies. The reason is that we need to assign different prime numbers to each class and perform computations on numbers that are bound by the product of these primes. The product of only the first nine primes fits into a 32-bit integer and the first 15 primes into a 64-bit integer. Table compression techniques [5], or the use of minimal perfect hash functions [23] instead, can help overcome the problem.

We would like to mention that in response to an earlier version of this paper Gabor Greif sent us his unpublished notes on a similar use of Chinese Remainders for implementing multiple dispatch [32] in Dylan.

7. Related work

Programming languages can support multi-methods either through built-in facilities, preprocessor, or library extensions. Naturally, tighter language integration enjoys a much broader design space for type checking, ambiguity handling, and optimizations compared to libraries. In this section, we will first review both library and non-library approaches for C++ and then give a brief overview of multi-methods in other languages.

7.1. Cmm

Cmm [48] is a preprocessor-based implementation for an open-method C++ extension. It takes a translation unit and generates C++ dispatch code from it. Cmm is available in two versions. One uses RTTI to recover the dynamic type of objects to identify the best overrider. The other achieves constant time dispatch by relying on a virtual function overridden in each class. Dispatch ambiguities are signaled by throwing runtime exceptions. Cmm allows dynamically linked libraries to register and unregister their open-methods at load and unload time. In addition to open-method dispatch, Cmm also provides call-site virtual dispatch. Call-site virtual dispatch delays the binding to regular overloaded functions, if one of their actual arguments is preceded by the **virtual** keyword.

```
void foo(A&);
void foo(B&); // B derives from A

// call site virtual dispatch
foo(virtual x); // which foo gets invoked depends on the dynamic type of x
```

Cmm does not provide special support for repeated inheritance, and therefore its dispatch technique does not entirely conform to virtual function semantics.

7.2. DoubleCpp

DoubleCpp [10] is another preprocessor-based approach for multi-methods dispatching on two virtual parameters. It essentially translates these multi-methods into the visitor pattern. For doing so, DoubleCpp requires access to the files containing the class definitions in order to add the appropriate accept and visit methods. DoubleCpp, unlike other visitor-based approaches, reports potential ambiguities.

7.3. Accessory function

The accessory functions papers [25,57] allow open-method dispatch based on a single virtual argument and discuss ideas to extend the mechanism for multiple dispatch. The compilation model they describe uses, like our approach, a compiler and linker cooperation to perform ambiguity resolution and dispatch-table generation. However, the accessory functions are integrated into the regular v-tables of their receiver types, which requires the linker to not only generate the dispatch table but also to recompute and resolve the v-table index of any other virtual member function. Neither paper provides a detailed discussion of the intricacies when multiple inheritance is involved. The authors do not refer to a model implementation to which we could compare our approach.

7.4. Loki

Loki [2], based on Alexandrescu's template programming library with the same name, provides several different dispatchers that balance between speed, flexibility, and code verbosity. Currently, it supports multi-methods with two arguments only, except for the constant-time dispatcher that allows more arguments. The static dispatcher provides call resolution based on overload resolution rules, but requires manual linearization of the class hierarchy in order to uncover the most derived type of an object first. All other dispatchers do not consider hierarchical relations and effectively require explicit resolution of all possible cases.

7.5. OOLANG

In [43], Panizzi and Pastorelli describe their open-method implementation for OOLANG, a language developed for the Apemille SPMD supercomputer that has a C++ like object model. The paper gives special attention to the handling of covariant return types. OOLANG's system differs from our implementation in the handling of repeated inheritance. Classes that repeatedly inherit from a base-class must define an overrider for each open-method that uses the base-class as type for a virtual parameter. Furthermore, OOLANG does not use covariant return type information for ambiguity resolution.

7.6. Other approaches

Besides the approaches mentioned in Section 4, languages can provide multi-method abstractions through a library (e.g. Python [45]). Chambers and Chen [19] present an alternative implementation technique based on a lookup DAG. Their work generalizes multiple dispatch to be a subset of predicate-based dispatch.

7.7. Multiple dispatch in practice

In order to estimate how often multiple dispatch is used in practice, Muschevici et al. [42] studied programs that utilize dynamic dispatch. The article introduces a language independent model for describing multiple dispatch, and defines six metrics on generic functions (i.e. in C++ an open-method family or a virtual function and its overriders) that measure aspects such as the number of arguments used for dynamic dispatch, the number of overriders in a multi-method family, etc. Using these metrics, the article analyzes nine applications – mostly compilers – written in six different languages: CLOS, Dylan, Cecil, MultiJava, Diesel [18], and Nice [12]. Their results show that 13%–32% of generic functions utilize the dynamic type of a single argument, while 2.7%–6.5% of them utilize the dynamic type of multiple arguments. The remaining 65%–93% of generic functions have a single concrete method, and therefore are not considered to use the dynamic types of their arguments. In addition, the study reports that 2%–20% of generic functions had two and 3%–6% had three concrete function implementations. The numbers decrease rapidly for functions with more concrete overriders. Since the multiple dispatch semantics presented in this paper is in line with the model defined by Muschevici et al., we expect similar results for C++ with open-methods.

8. Results

In order to discuss time and space performance, we compare code generated by our C++ Open Method Compiler, described in Section 6, to a number of prototype implementations, the visitor pattern, Cmm, DoubleCpp, and the Loki library. The prototypes of our design alternatives were implemented in C to approximate the lowering of C++ code to C. They were initially developed to assess performance trade-offs of different approaches and work-around techniques and include open-methods (can be declared freely), multi-methods (have to be declared in class, thus the om-tables can be embedded into the v-table, saving two indirections per argument Section 6.3.1.), and a Chinese Remainder (Section 6.3) based implementation. These implementations lay out the dispatch table for the concrete example described below as C data structures, and then dispatch calls through it.

We wrote 20 classes (representing shapes, etc.) that can intersect each other. Overall, this results in 400 combinations for binary dispatch functions. We implemented 40 specific intersect functions to which all of the 400 combinations are dispatched. In order to get a reliable timing of the function invocation, these 40 intersect functions only increment a counter. Since not all techniques we use support multiple inheritance, these 20 classes only use single inheritance. The actual test consists of a loop that randomly chooses 2 out of 32 objects and invokes the intersect method. We implemented a table-based random number generator that is simple and does not contain any floating-point calculations or integer-divisions. We ran the loop twice with the same random numbers. The first run allows implementations that build the dispatch data structure on the fly to warm up and load data/code into the cache. The second loop was timed. The clock-cycle based timer takes the time before and after the loop and we calculate the average number of clock-cycles per loop to compare the results.

8.1. Implementations

We tested the approaches on a Pentium D, 2.8 GHz running CentOS Linux and a Core2Duo running Mac OSX. The code for the performance tests was compiled with g++ 4.1 (Linux) and gcc 4.0.1 (OSX) with optimization level set to -O3. The C++ Open Method Compiler generates source code lowered to C, which was compiled with the corresponding gcc versions and linked to the pre-linker generated dispatch tables.

Using the Chinese Remainder approach, the number associated with the dispatch table grows exponentially with the number of types. Therefore the test is limited to 8 types instead of 20 and the size of the executable is omitted.

For Loki, we only tested the static dispatcher because the others require manual handling of all possible cases. Using other dispatchers would have been closer to a scenario of a manually allocated array of functions through which calls are made. However, as we indicated before, the dual nature of multi-methods require them to provide both dynamic dispatch and automatic resolution mechanism.

8.2. Results & interpretation

Our experimental results can be summarized in terms of execution time and program size:

Approach	**Size (bytes)** Linux	**Cycles/Loop** Pentium-D	**Cycles/Loop** Core2Duo
Virtual function	n/a	75	55
Multi-methods prototype	42 972	78	60
Open-methods prototype	40 636	82	63
C++ Open-method Compiler	42 504	82	64
Double Cpp	34 812	120	82
C++ Visitor	38 236	132	82
Chinese Remainders prototype	n/a	175	103
Cmm (constant time)	155 344	415	239
Cmm	155 056	1 320	772
Loki Library	75 520	3 670	2 238

Executable size: To obtain a comparable size of the executable, we used the regular EDG frontend to generate C code for the alternative approaches. Then we compiled all intermediate C files with gcc, where optimizations were set to minimize the code size. Moreover, we stripped off the symbols from the executables. The size of the dispatch tables is mentioned as one of the major drawbacks of providing multi-methods as a programming language feature [57]. However, our results reveal that the best achievable code size is roughly the same for visitors, prototyped multi-/open-method, and C++ Open-method Compiler implementations. With the visitor, each shape class has intersect methods for all 20 shapes of the hierarchy. A somewhat smarter approach would be to remove redundant intersect overriders. However, removing specific overriders is tedious and difficult to maintain, since the dispatch would be based on the static type information of the base class. Even an optimized approach would require as many v-table entries as there are in a dispatch table, simply because each type contains 20 intersect entries in the v-table. Multiplying this with the number of shapes, 20, results in 400, exactly the number of entries found in the dispatch table. We do not discuss the program size of the two Cmms and Loki, since they use additional header files such as <typeinfo> and <stdexcept> that distort a direct comparison.

Execution time: The results for prototyped multi-methods, prototyped open-methods, and C++ Open-method Compiler are (as expected) roughly comparable to a single virtual function dispatch, which needs 75 (55 on the Core2Duo) cycles per loop. Hence, the better performance compared to the visitors is not surprising. However, the fact that multi-methods reduce the runtime to 62% (73%) of the reference implementation using the visitor is noteworthy. We conjecture this is an effect of the size of the class hierarchy and that the time to double dispatch depends on the number of overriders. On the Pentium D, two observations support our conjecture. Firstly, the DoubleCpp-based visitor has no redundant overriders and runs slightly faster. Secondly, we simulated an analysis pass dispatching over AST-objects of 20 different types and counting the category to which they belong (type, declaration, expression, statement, other). In this case, the double dispatch has only 20 leaf-functions instead of 400 and our dispatch test runs 78 cycles instead of 132. The open-method approach requiring only five overriders, is still faster and needs 68 cycles.

The difference between the prototyped multi-methods and open-methods (the comparison with the C++ Open-method Compiler is stated in parentheses) is within the expected range. Four more indirections require 4 (4) more clock cycles on the Pentium and 3 (4) more on the Core2Duo. Although significantly slower, Cmm (constant time) performs better than expected, since its author estimates the dispatch cost as 10 times a regular virtual function call. As expected, the two non-constant time approaches perform worst.

Significance of performance: The performance numbers come from experiments designed to highlight the cost of multiple dispatch: the functions invoked hardly do anything. Depending on the application the improved performance may or may not be significant. For the image conversion example, gains in execution speed are negligible compared to time spent in the actual conversion algorithm. In other cases, such as the evaluation of expressions using user-defined arithmetic types, traversal of abstract syntax trees, and some of the most frequent shape intersect examples, the speed differences among the double dispatch approaches appear to be notable.

Contrary to much "popular wisdom", our experiments revealed that for many applications the use of dispatch tables for open-methods and multi-methods actually reduces the program size compared to brute-force and work-around techniques. Under the assumption that the use of open-methods in C++ would be similar to Muschevici et al.'s results (Section 7.7), we conclude that the size of the dispatch table will remain small for most practical cases.

9. Experiences

In order to compare open-methods with double dispatch and the visitor pattern, we have implemented some of the examples from Section 2.

9.1. Image format conversion

The first example is image conversion. To meet the performance requirements typical for image processing applications, information about the exact source and destination formats is indispensable for an efficient conversion. With this information, we can call a routine geared for that specific pair of formats. Any attempt to work through a common base interface will significantly hinder performance, and should be avoided. This is why we use a fairly shallow class hierarchy to represent different image formats. Another interesting aspect of this example is that when the pair of formats is known statically, it is feasible to write a generic conversion algorithm that relies on some format traits. This is an approach taken by Adobe's GIL library [13]. Therefore the main goal in this example is to uncover the dynamic types of both arguments and pass on these uncovered arguments together with their static types to a set of overloaded template functions.

```
template <class SrcImage, class DstImage>
bool generic_convert(const SrcImage& src, DstImage& dst);

typedef unsigned char color_component;

struct image
{
  // member—functions to access row buffer, width, height etc.
};

struct RGB : image // abstract base of all RGB images
{
  struct color { color_component R, G, B, A; };
  virtual color get_color(int i, int j) const          = 0;
  virtual void set_color(int i, int j, const color& c) = 0;
};

struct RGB32 : RGB { /*implements get_color, set_color*/};
// ... Similar definitions for RGB24, RGB16, RGB15, RGB08

struct YUV : image // abstract base of all YUV images
{
```

```
  struct color { color_component Y, U, V, A; };
  virtual color get_color(int i, int j) const          = 0;
  virtual void set_color(int i, int j, const color& c) = 0;
};

struct UYVY : YUV { /*implements get_color, set_color*/};
// ... Similar definitions for YUY2,Y41P,CLJR,YVU9,YV12,IYUV,I420,Y800 etc.

struct CMYK : image
{
  struct color { color_component C, M, Y, K; };
  virtual color get_color(int i, int j) const          = 0;
  virtual void set_color(int i, int j, const color& c) = 0;
};
```

Open multi-methods to handle the cases can be listed separately from class definitions:

```
// Base open—method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }

// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);

// Fast generic conversions, generated for each combination of types
bool convert(virtual const RGB32& src, virtual RGB32& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual RGB24& dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual YUY2&  dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual YVU9&  dst) { return generic_convert(src, dst); }
bool convert(virtual const RGB32& src, virtual I420&  dst) { return generic_convert(src, dst); }
```

In case of double dispatch, the code becomes cluttered with definitions to support the mechanism:

```
// Forward declare all classes that would participate in double dispatch
struct RGB32;
struct RGB24;
// ... others

struct image
{
  // member—functions to access row buffer, width, height etc.

  // Double dispatch support code
  virtual bool convert_to(image& dst) const = 0;
  virtual bool convert_from(const RGB32& src) { return false; }
  virtual bool convert_from(const RGB24& src) { return false; }
  virtual bool convert_from(const RGB16& src) { return false; }
  // ... etc. for all other leaf image classes
};

struct RGB32 : RGB
{
  virtual bool convert_to(image& dst) const { return dst.convert_from(*this); }
  virtual bool convert_from(const RGB32& src) { return generic_convert(src, *this); }
  virtual bool convert_from(const RGB24& src) { return generic_convert(src, *this); }
  virtual bool convert_from(const RGB16& src) { return generic_convert(src, *this); }
  // ... etc. for all other leaf image classes
};
```

The major disadvantage of the double dispatch approach is that we have to foresee the whole hierarchy at the moment we are defining its root. This is necessary for declaring the interface for uncovering types. Once it is defined, we cannot extend it for newly created classes—they will all be treated as their closest ancestor in the hierarchy. Another problem with double dispatch is that its supportive structures clutter the code. This may be acceptable when double dispatch is needed for only one algorithm, but when several algorithms require it (e.g. we would also like to have a polymorphic **bool** compare(**virtual const** image& a, **virtual const** image& b)) then the code may quickly get out of hand. While this aspect

of the double dispatch can be solved with the visitor pattern at the cost of two extra virtual calls, the open-method solution will remain cleaner as open-methods do not even need to be defined together with the class. We discuss the visitor pattern in greater detail in our second example.

The number of lines in the implementation with open-methods was smaller, but all in all, the number of lines in both implementations is growing as the square of the number of classes in the hierarchy. We note that for open multi-method implementations this is a rather exceptional case, because the class hierarchy was shallow, while we were interested in uncovering all possible type combinations. For the double dispatch, this is rather typical case because the supportive definitions will have to be there anyway.

In the image conversion example the main purpose of the open-methods is to discover the dynamic types of both arguments and then forward the call to an overloaded function. This raises a question of whether the introduction of parameterized overriders would not make such definitions easier. In such a case, users can introduce only a base open method and a parameterized version of all the overriders. The compiler will then use the parameterized version to generate all the entries in the dispatch table:

```
// Base open—method. Fails as we do not know anything about the formats
bool convert(virtual const image& src, virtual image& dst) { return false; }

// Slow polymorphic conversions
bool convert(virtual const RGB& src, virtual RGB& dst);
bool convert(virtual const RGB& src, virtual YUV& dst);
bool convert(virtual const YUV& src, virtual RGB& dst);
bool convert(virtual const YUV& src, virtual YUV& dst);

// Fast generic conversions, generated for each combination of types
template <class Source, class Destination>
bool convert(virtual const Source& src,virtual Destination& dst)
{
  return generic_convert(src, dst);
}
```

This feature, however, can be a subject of a separate work, so we do not investigate it here.

9.2. AST traversal

The second example discusses the use of open-methods to traverse ASTs. The key focus thereby is on extending classes with open dispatch rather than multiple dispatch. Open-methods essentially become virtual functions that can be added to a class after it has been defined. The examples in this section reflect our experience of writing an analysis pass for the Pivot source-to-source transformation infrastructure [53]. The Pivot uses the visitor pattern to type-safely uncover the dynamic type of AST nodes. The Pivot consists of approximately 150 classes, but in the ensuing discussion, we limit the AST hierarchy to only two of them, where one, Expr, is a base class for all kinds of expressions, and the other, Unary, is an implementation of unary expressions.

```
struct Expr
{
  . . .
  virtual accept(Visitor& v) const { v.visit(*this); }
};

struct Unary : Expr
{
  . . .
  accept(Visitor& v) const { v.visit(*this); }
};

struct Visitor
{
  void visit(const Expr&   ) = 0;
  void visit(const Unary&) = 0;
};
```

Forwarding calls to base implementations: Currently, the Pivot has some 150 node types. The Pivot provides a number of intermediate abstract base classes that factor commonalities (e.g. Expr, Type, Declaration, etc.) of the 150 node types. If the logic of the visitor can be implemented in terms of a single base class, the bodies of the more specific types will need to explicitly invoke the base implementation (compare to the implementation for Unary). Open-methods have this forwarding behavior by default.

```
struct SimpleVisitor : Visitor
{
  virtual void visit (const Expr&   e) { /* . . . */ };
  virtual void visit (const Unary& u) { visit(static_cast<Expr&>(u)); } // calls visit (Expr&)
};

// All objects derived from Expr will be handled by simpleOpenMethod
void simpleOpenMethod(virtual const Expr&) {/* . . . */}

void foo(Expr& e, Unary& u)
{
  SimpleVisitor vis;

  e.accept(vis); // invokes SimpleVisitor :: visit (const Expr&)
  u.accept(vis); // invokes SimpleVisitor :: visit (const Unary&) first

  simpleOpenMethod(e); // invokes simpleOpenMethod(const Expr&)
  simpleOpenMethod(u); // invokes simpleOpenMethod(const Expr&)
}
```

Passing of arguments and results: The signatures of the visit and accept functions are determined when the visitor and the AST node classes are defined. Passing additional arguments to (or returning a value from) the visit functions requires intermediate storage as part of the visitor class. In the following example, inh and syn correspond to the input and result values of a function.

```
struct AnalysisPassVisitor : Visitor
{
  const InheritedAttr& inh; // data member for input parameter
  SynthesizedAttr*     syn; // data member for return value

  Visitor(const InheritedAttr & inherited) : inh(inherited), syn() {}
  . . .
};
```

To avoid code duplication, it is useful to factor constructing the visitor and reading out the result into separate functions.

```
SynthesizedAttr* visit_foo (const Expr& e, const InheritedAttr & inh)
{
  AnalysisVisitor v(inh); // construct the visitor and pass the context
  e.accept(v);
  return v.syn;           // read and return the result
}
```

With open-methods, additional arguments can easily be specified as part of their signatures.

```
SynthesizedAttr* analysisPass(virtual const Expr& e, const InheritedAttr& inh);
```

Covariant return type: Since the result requires intermediate storage, covariant return types cannot easily be implemented with the visitor pattern. Consider the following implementation of a visitor that creates and returns a copy of an AST node.

```
struct CloneExpr : Visitor
{
  Expr* result; // data member for return value

  // make a copy of an Expr object
  virtual void visit (const Expr&   e) { result = new Expr(e); }
  // make a copy of an Expr object
  virtual void visit (const Unary& u) { result = new Unary(u); }
};

Expr* clone(const Expr& e) // analog of a base—method
{
  CloneExpr v;
  e.accept(v);
  return v.result;
}
```

Cloning a unary expression loses some type information, because the cloned objects would get returned as Expr. An implementation that is able to return covariant types requires a different visitor implementation (or instantiation) with

similar boilerplate code for each covariant return type. These repetitive definitions can be eliminated by using templates. The following example shows a cloning visitor that returns a Unary object.

```
struct CloneUnary : Visitor
{
  Unary* result; // data member for covariant return value
  virtual void visit (const Expr&  e) { assert(false); } // Can never be called!
  virtual void visit (const Unary& u) { result = new Unary(u); }
};

Unary* clone(const Unary& u) // analog of an overrider with covariant return type
{
  CloneUnary v;
  u.accept(v);
  return v.result;
}
```

The definition of open-methods with covariant return types is straightforward:

```
Expr& clone(virtual const Expr&    a); // base—method
Unary& clone(virtual const Unary& u); // overrider with covariant return type
```

Similar to open-methods, a hand-crafted technique for modeling covariant return type with visitors also creates additional "dispatch tables" for each overrider with covariant return type. Those are created in a form of v-tables for additional visitors. Interestingly enough, the overall size of such dispatch tables would be larger than those generated for open-methods, because visitors require v-table entries for visit methods that can never occur at runtime (see the assert in CloneUnary::visit(Expr&) above). This is not the case with open-methods as overriders with covariant return type will operate on smaller class hierarchies for their arguments.

Passing open-methods as callbacks: Open-methods nevertheless sometimes have disadvantages in comparison with the visitor pattern. Consider a traversal mechanism that traverses an AST in certain order. The mechanism can accept either a visitor or an open-method for node visitation:

```
void evaluation_order_traversal  (const Expr& e, Visitor& v);
void evaluation_order_traversal  (const Expr& e, void(*fn)(const Expr&));
```

In the case of a visitor, we can pass or accumulate some data during visitation. However, in the case of an open-method, we would need to accumulate data elsewhere.

```
struct CodeGenerationVisitor : Visitor
{
  std::vector<Instruction> instructions; // instruction stream inside visitor
  void visit(const Expr& e) { /* generate code for e*/ }
  // ...
};

CodeGenerationVisitor v;
evaluation_order_traversal(root_node(),v);
```

Although our current implementation does not support taking the address of an open-method, we can simulate that behavior by wrapping the open-method call inside a thunk.

```
std::vector<Instruction> instructions; // global instruction stream

void generate_code(virtual const Expr&   e) { instructions.push_back(...); }
void generate_code(virtual const Unary& e) { instructions.push_back(...); }
//...
void thunk_generate_code(const Expr& e) { generate_code(expr); }

evaluation_order_traversal(root(), &thunk_generate_code);
```

10. Conclusions and future work

We have presented a novel approach to dispatching open multi-methods that is in line with the multiple-inheritance semantics of the current C++ object model and the C++ overload resolution rules. This implies compile-time or link-time detection of ambiguities. By considering covariant return types in the ambiguity resolution, we reduce the number of potential conflicts. We have discussed an implementation based on modifications to the EDG compiler front-end and have

described a mechanism that supports the integration of several translation units. Our evaluation of different approaches to implementing open-methods in C++ shows that our approach is significantly better (in time and space) than current work-arounds. Indeed, it is only 16% slower than single dispatch. Since the dispatch is constant time and does not rely on exceptions to signal ambiguities, it is applicable in embedded and hard real-time systems.

Future plans to extend our work include:

10.1. Virtual function templates

Virtual function templates are a powerful abstraction mechanism not part of C++ (see [51, §15.9.3]). Generating v-tables for virtual function templates requires a whole-program view and C++ traditionally relies almost exclusively on separate compilation of translation units. In [44], we demonstrate that the pre-linker described in this paper is able to synthesize dispatch tables for an approximation of templated open-methods. The concrete semantics of templated virtual functions (and open-methods) remains an open topic.

10.2. Function pointers to open-methods

Pointers to member functions in C++ preserve polymorphic behavior when they point to a virtual member function. To be in line with this semantics, pointers to open-methods should preserve dynamic dispatch too. This could be implemented by generating a thunk every time an address of an open-method is taken, and using the address of this thunk instead. Inside the function, the compiler simply generates a call to the appropriate open-method. Note that similar to single dispatch in C++, it will not be possible to take the address of a particular open-method overrider—the returned function will always dispatch dynamically.

10.3. Calling a base implementation

C++ provides a syntax to call a particular base implementation of a virtual member function directly, avoiding dynamic dispatch. This is often used to call the function in the base class. To do this, C++ requires the user to use a fully qualified name of virtual member function: e.g. `p->MyClass::foo();`. It is likely that similar functionality will be required for open-methods.

We propose to be able to fix the dynamic type of an argument at the point of the open-method call to a particular unambiguous base class. This can either be done via `fix_type<Base>(arg)` or by introducing a special syntax, such as `arg as Base`. The concrete form is still under discussion. Under this approach users will be able to say `foo(d1 as B, d2)`, which means that the runtime type of `d1` is considered to be `B` rather than its actual runtime type. This effectively fixes the corresponding row or column in the dispatch table during the call. We note that the static type of `d1` has to be unambiguously derived from `B` in order for such type fix to be applicable.

This approach differs from the one used in MultiJava [21], where users have a choice between `resend` and `super` calls. `resend` invokes a less specific implementation that the current overrider refines, providing it can be uniquely determined. In cases when the overrider that invokes `resend` refines multiple overriders, a compile-time error is reported. A call to `super` dispatches to an implementation for a strict superclass of the receiver object.

Acknowledgements

We would like to thank to Nan Zhang for contributions to this research in its early stages and to Quadrox NV (Belgium) for providing the source code for experimenting with image conversion ideas. We are also grateful to Luke Wagner, Damian Dechev, Jaakko Järvi, Gabriel Dos Reis, and numerous anonymous referees for their helpful suggestions for improvement.

References

[1] R. Agrawal, L.G. Demichiel, B.G. Lindsay, Static type checking of multi-methods, in: OOPSLA'91: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA, 1991.
[2] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele Jr., S. Tobin-Hochstadt, The Fortress Language Specification, Tech. rep., version 1.0 (March 2008).
[4] E. Allen, J. Hallett, V. Luchangco, S. Ryu, J. Guy, L. Steele, Modular multiple dispatch with multiple inheritance, in: SAC'07: Proceedings of the 2007 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2007.
[5] E. Amiel, O. Gruber, E. Simon, Optimizing multi-method dispatch using compressed dispatch tables, in: OOPSLA'94: Proceedings of the Ninth Annual Conf. on Object-oriented Programming Systems, Language, and Applications, ACM Press, New York, NY, USA, 1994.
[6] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, 4th ed., Prentice Hall, PTR, 2005.
[7] M.H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
[8] P. Becker, The C++ Standard Library Extensions: A Tutorial and Reference, 1st ed., Addison-Wesley Professional, Boston, MA, USA, 2006.
[9] P. Becker, Working Draft, Standard for Programming Language C++, Tech. Rep. N2857, JTC1/SC22/WG21 C++ Standards Committee (March 2009).
[10] L. Bettini, S. Capecchi, B. Venneri, Double Dispatch in C++, Software - Practice and Experience 36 (6) (2006) 581–613.
[11] G.M. Birtwistle, O. Dahl, B. Myhrhaug, K. Nygaard, Simula BEGIN, Auerbach Press, Philadelphia, 1973.

[12] D. Bonniot, B. Keller, F. Barber, The Nice user's manual (2008). http://nice.sourceforge.net/manual.html.
[13] L. Bourdev, J. Järvi, Efficient run-time dispatching in generic programming with minimal code bloat, in: Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon, 2006.
[14] J. Boyland, G. Castagna, Parasitic methods: An implementation of multi-methods for Java, in: OOPSLA'97: Proceedings of the 12th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 1997.
[15] K. Bruce, L. Cardelli, G. Castagna, G.T. Leavens, B. Pierce, On binary methods, Theor. Pract. Object Syst. 1 (3) (1995) 221–242.
[16] C. Chambers, Object-oriented multi-methods in Cecil, in: ECOOP'92: Proceedings of the European Conf. on Object-oriented Programming, Springer-Verlag, London, UK, 1992.
[17] C. Chambers, The Cecil language: Specification and rationale. 3.2, Tech. rep. (2004).
[18] C. Chambers, The Diesel Language, specification and rationale (2006). http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-diesel-lang/diesel-spec.pdf.
[19] C. Chambers, W. Chen, Efficient multiple and predicated dispatching, in: OOPSLA'99: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 1999.
[20] C. Clifton, G.T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: OOPSLA'00: Proceedings of the 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2000.
[21] C. Clifton, T. Millstein, G.T. Leavens, C. Chambers, MultiJava: Design rationale, compiler implementation, and applications, ACM Trans. Program. Lang. Syst. 28 (3) (2006) 517–575.
[22] codesourcery.com, The Itanium C++ ABI, Tech. rep. (2001).
[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 2001.
[24] Edison Design Group, C++ Front End, http://www.edg.com/ (July 2008).
[25] C.B. Flynn, D. Wonnacott, Reconciling encapsulation and dynamic dispatch via accessory functions, Tech. Rep. 387 (1999).
[26] B. Foote, R. E. Johnson, J. Noble, Efficient multimethods in a single dispatch language, in: Proceedings of the European Conference on Object-Oriented Programming, Glasgow, Scotland, July.
[27] C. Frost, T. Millstein, Modularly typesafe interface dispatch in JPred, in: 2006 International Workshop on Foundations and Development of Object-oriented Languages, FOOL/WOOD'07, Charleston, SC, USA, 2006.
[28] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[29] M. Gibbs, B. Stroustrup, Fast dynamic casting, Softw. Pract. Exper. 36 (2) (2006) 139–156.
[30] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
[31] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: linguistic support for generic programming in C++, in: OOPSLA'06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006.
[32] G. Greif, Chinese Dispatch – Unpublished notes for a talk, Dylan Hackers Conference, Berlin (July 2002). http://www.opendylan.org/cgi-bin/viewcvs.cgi/trunk/www/papers/ChineseDispatch.lout?rev=8014&view=markup.
[33] International Standardization Organization, ISO/IEC 10918-1:1994: Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines, pub-ISO, pub-ISO:adr, 1994.
[34] ISO/IEC 14882 International Standard, Programming languages: C++, American National Standards Institute, 1998.
[35] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: Challenges of constrained generics in C++, in: PLDI'06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 2006.
[36] B. Liskov, Keynote address - data abstraction and hierarchy, in: OOPSLA'87: Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum), ACM Press, New York, NY, USA, 1987.
[37] L. Martin, Joint Strike Fighter, Air Vehicle, C++ Coding Standard, Lockheed Martin, 2005.
[38] B. Meyer, Eiffel: The Language, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
[39] T. Millstein, C. Chambers, Modular statically typed multimethods, Information and Computation 175 (1) (2002) 76–118.
[40] T. Millstein, M. Reay, C. Chambers, Relaxed MultiJava: Balancing extensibility and modular typechecking, in: OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conf. on Object-oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2003.
[41] T.D. Millstein, C. Chambers, Modular Statically Typed Multimethods, in: ECOOP'99: Proceedings of the 13th European Conf. on Object-Oriented Programming, in: LNCS, vol. 1628, Springer-Verlag, London, UK, 1999.
[42] R. Muschevici, A. Potanin, E. Tempero, J. Noble, Multiple dispatch in practice, in: OOPSLA'08: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 2008.
[43] E. Panizzi, B. Pastorelli, Multimethods and separate static typechecking in a language with C++-like object model, The Computing Research Repository (CoRR) cs.PL/0005033.
[44] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Dynamic algorithm selection for runtime concepts, Science in Computer Programming, in press, (http://dx.doi.org/10.1016/j.scico.2009.04.002).
[45] G.v. Rossum, The Python Language Reference Manual, Network Theory Ltd., 2003, Paperback.
[46] M. Schordan, D. Quinlan, A source-to-source architecture for user-defined optimizations, in: JMLC'03: Joint Modular Languages Conference, in: LNCS, vol. 2789, Springer-Verlag, 2003.
[47] A. Shalit, The Dylan Reference Manual, 2nd edition, Apple Press, 1996.
[48] J. Smith, Draft proposal for adding Multimethods to C++., Tech. Rep. N1463 (2003).
[49] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: OOPSLA'86: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, ACM, New York, NY, USA, 1986.
[50] G.L. Steele Jr, Common LISP: the Language, 2nd ed., Digital Press, Newton, MA, USA, 1990.
[51] B. Stroustrup, The Design and Evolution of C++, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.
[52] B. Stroustrup, The C++ Programming Language, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[53] B. Stroustrup, G. Dos Reis, Supporting SELL for high-performance computing, in: 18th International Workshop on Languages and Compilers for Parallel Computing, in: LNCS, vol. 4339, Springer-Verlag, 2005.
[54] D. Thomas, A. Hunt, Programming Ruby: The Pragmatic Programmer's Guide, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[55] J. Visser, Visitor combination and traversal control, in: OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2001.
[56] D. Wasserrab, T. Nipkow, G. Snelting, F. Tip, An operational semantics and type safety proof for multiple inheritance in C++, in: OOPSLA'06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006.
[57] D. Wonnacott, Using accessory functions to generalize dynamic dispatch in single-dispatch object-oriented languages, in: COOTS'01: 6th USENIX Conference on Object-Oriented Technologies and Systems, San Antonio, TX, USENIX, 2001.
[58] A. Wöß, M. Löberbauer, H. Mössenböck, LL(1) conflict resolution in a recursive descent compiler generator, in: JMLC'03: Joint Modular Languages Conference, in: LNCS, vol. 2789, Springer-Verlag, 2003.
[59] www.boost.org, The Boost C++ Libraries, retrieved on July 4th, 2008.
[60] www.fourcc.org, Video codec and pixel format definitions, retrieved on February 20th, 2007.

Runtime Concepts for the C++ Standard Template Library

Peter Pirkelbauer
Texas A&M University
College Station, TX, U.S.A.
peter.pirkelbauer@tamu.edu

Sean Parent
Adobe Systems, Inc.
San Jose, CA, U.S.A.
sparent@adobe.com

Mat Marcus
Adobe Systems, Inc.
Seattle, WA, U.S.A.
mmarcus@adobe.com

Bjarne Stroustrup
Texas A&M University
College Station, TX, U.S.A.
bs@cs.tamu.edu

ABSTRACT

A key benefit of generic programming is its support for producing modules with clean separation. In particular, generic algorithms are written to work with a wide variety of unmodified types. The *Runtime concept* idiom extends this support by allowing unmodified concrete types to behave in a runtime polymorphic manner. In this paper, we describe one implementation of the runtime concept idiom, in the domain of the C++ standard template library (STL). We describe and measure the performance of runtime-polymorphic analogs of several STL algorithms. We augment the runtime concept idiom by employing a dispatch mechanism that considers both type and concept information to maximize performance when selecting algorithm implementations. We use our implementation to demonstrate the effects of different compile-time vs. run-time algorithm selection choices, and we indicate where improved language and compiler support would be useful.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General

General Terms

Design, Languages

Keywords

Generic Programming, Runtime Polymorphism, C++, Standard Template Library

1. INTRODUCTION

ISO C++ [12] supports various programming paradigms, notably object-oriented programming and generic programming. Object-oriented techniques are used when runtime polymorphic behavior is desired. When runtime polymorphism is not required, generic programming is used, as it offers non-intrusive, high performance compile-time polymorphism; examples include the C++ Standard Template Library (STL) [5], the Boost Libraries [1], Blitz++ [17], STAPL [4].

Recent research has explored the possibility of a programming model that retains the advantages of generic programming, while borrowing elements from object-oriented programming, in order to support types to be used in a runtime-polymorphic manner. In [15], Parent introduces the notion of non-intrusive value-based runtime-polymorphism, which we will refer to as the *runtime concept* idiom. Marcus et al. [14], [3], and Parent [16] extend this idea, presenting a library that encapsulates the common tasks involved in the creation of efficient runtime concepts. Järvi et al. discuss generic polymorphism in the context of library adaptation [13].

A key idea in generic programming is the notion of a *concept*. A concept [10] is a set of semantic and syntactic requirements on types. Syntactic requirements stipulate the presence of operations and associated types. In the runtime concept idiom, a class R is used to model these syntactic requirements as operations. The binding from R to a particular concrete type T is delayed until runtime. Any type T that syntactically satisfies a concept's requirements can be used with code that is written in terms of the runtime concept.

In this paper, we apply these principles to develop a runtime-polymorphic version of some STL sequence containers and their associated iterators. Runtime concepts allow the definition of functions that operate on a variety of container types.

Consider a traditional generic function expressed using C++ templates:

```
// conventional template code
template <class Iterator>
Iterator
random_elem(Iterator first, Iterator last)
{
  typename Iterator::difference_type dist = distance(first, last);
  return advance(first, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

Objects of any type that meet the iterator requirement

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 23rd ACM symposium on applied computing (SAC), 2008. http://doi.acm.org/10.1145/1363686.1363734
SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

can be used as arguments to random_elem. However, those requirements cannot be naturally expressed in C++98, (though they can in C++0x), and the complete function definition is needed for type checking and code generation. The resulting code is very efficient, but this style of generic programming does not lend itself to certain styles of software development (e.g. those relying on dynamic libraries).

We can write essentially the same code using the runtime concept idiom:

```
// with runtime concept idiom
wrapper_forward<int>
random_elem(wrapper_forward<int> f, wrapper_forward<int> l)
{
  wrapper_forward<int>::difference_type dist = distance(f, l);
  return advance(f, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

However, here the binding between the iterator type and the function is handled at runtime and we can compile a use of random_elem with only the declarations of random_elem available:

```
// with runtime concept idiom:
wrapper_forward<int>
random_elem(wrapper_forward<int> f, wrapper_forward<int> l);
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

By using runtime concepts, function implementations (e.g.: random_elem) are isolated from client code. The parameter type wrapper_forward subsumes all types that model the concept forward-iterator. The implementation can be explicitly instantiated elsewhere for known element types, and need not be available to callers.

This reduced code exposure in header files makes runtime concepts suitable for (dynamically linked) libraries and when source code cannot be shared. However, the use of runtime concepts comes at a cost. The function random_elem is written in terms of the concept forward-iterator. The runtime complexity of distance and advance is $O(n)$ for forward-iterators, while it is constant time for randomaccess-iterators. Passing iterators of vector<int> as arguments would incur unnecessary runtime overhead.

This paper makes the following contributions:

- We apply the runtime concept idiom to part of a core C++ library (STL) and analyze the runtime overhead.
- We enhances the runtime concept idiom with a prototype of an open, extensible, and loosely coupled algorithm library– a runtime counterpart of the STL algorithms. Its dispatch mechanism selects the best matching algorithm instance according to runtime concept and type information of the actual atguments. This eliminates the need for dynamic dispatch when a matching algorithm instance is present. Runtime analogs of four STL algorithms are presented and their performance is analyzed.
- We explore the problem of writing single algorithms that can simultaneously accommodate runtime-polymorphic variations in both container and element type. Performance measurements indicate that current language level support is not sufficient to support this idiom, but we point out where language enhancements might make it viable in the future.

The structure of this paper is: sections 2 and 3 revisit fundamental ideas of generic programming and illustrate the runtime concept idiom in the STL domain. Section 4 discusses our concrete application to the STL's sequences and algorithms. Section 5 evaluates our prototype implementation and its runtime performance; section 6 compares our model to alternatives; section 7 points to possible extensions and summarizes our contribution.

2. GENERIC PROGRAMMING

The ideal for generic programming is to represent code at the highest level of abstraction without loss of efficiency in both actual execution speed and resource usage compared to the best code written through any other means. The general process to achieve this is known as *lifting*, a process of abstraction where the types used within a concrete algorithm are replaced by the semantic requirements of those types necessary for the algorithm to perform.

Semantic Requirement: Types must satisfy these requirements in order to work properly with a generic algorithm. Semantic requirements are stated in tables, in documentation, and may at times be asserted within the code. Checking types against arbitrary semantic requirements is in general undecidable. Instead, compilers for current C++ check for the presence of syntactic constructs, which are assumed to meet the semantic requirements.

Concept: Dealing with individual semantic requirements would be unmanageable for real code. However, sets of requirements can often be clustered into natural groups, known as *concepts*. Although any collection of requirements may define a concept, only concepts which enable new classes of algorithms are interesting.

Model: Any type that satisfies all specified requirements of a concept is said to be a model of that concept.

Generic Algorithm: A generic algorithm is a derivative of an efficient algorithm, whose implementation is independent from concrete underlying data structures. The requirements that an algorithm imposes on a data structure can be grouped into concepts. An example that is part of the STL is find and the requirement on the template argument is to model forward-iterator.

Concept Refinement: A concept C_r that adds requirements to another concept C_0 is a concept refinement. When compared to C_0, the number of types that satisfy the requirements of C_r decreases, while the number of algorithms that can be directly expressed increases. For example, constant time random access, a requirement added by the concept randomaccess-iterator, enables the algorithm sort.

Algorithm Refinement: Parallel to concept refinements, an algorithm can be refined to exploit the stronger concept requirements and achieve better space- and/or runtime-efficiency. For example, the complexity of reverse for bidirectional-iterator is $O(n)$, while it is $O(n\ lg\ n)$ for forward-iterator (assuming less than $O(n)$ memory usage).

Regularity: Dehnert and Stepanov [9] define regularity based on the semantics of built-in types, their operators, the complexity requirements on the operators, and consistency conditions that a sequence of operations has to meet. Regularity is based on value-semantics and requires operations to construct, destruct, assign, swap, and equality-compare two instances of the same type. This is sufficient for a number of STL data structures and algorithms includ-

ing vector, queue, reverse, find. A stronger definition adds operations to determine a total order, which enables the use of STL's map, set, sort. Code written with built-in types in mind will work equally well for regular user defined types. Programmers' likely familiarity with built-in types makes the notion of regularity important.

3. RUNTIME CONCEPTS

In order to make our exposition as self-contained as possible, and to allow us to experiment with potential improvements, we have implemented and will illustrate the runtime concept idiom with hand constructed classes. For a deeper treatment of the runtime concept idiom, along with its library support and optimizations see Marcus et al. [14] and [3].

The runtime concept idiom employs a three-layer architecture, the concept layer, the model layer, and the wrapper layer. The following example explains the interaction of these three layers based on a *runtime concept* Copyable, that supports copy construction. The operational requirements of runtime concepts are expressed with an abstract base class. Here, the runtime concept for Copyable requires the single operation clone.

```
struct concept_copyable {
  virtual concept_copyable& clone() const = 0;
};
```

Concrete types T and runtime concepts are loosely coupled by means of the runtime model layer. By a *runtime model*, we mean a class template M parametrized on T and inheriting from the runtime concept. A model holds the data and implements the pure virtual functions declared in the runtime concept by forwarding the calls to T:

```
template <class T>
struct model_copyable : concept_copyable {
  model_copyable(const T& val) : t(val) {}
  model_copyable& clone() { return *new model_copyable(t); }

  T t;
};
```

The *wrapper* layer wraps concept_copyable objects and manages their lifetime. It contains operations that guarantee regular semantics (constructor, destructor, etc.) and makes other operations accessible through the dot operator. Here, the implementations of the copy constructor and the assignment operator makes use of the function clone.

```
struct wrapper_copyable {
  template<typename T>
  wrapper_copyable(const T& t) : c(new model_copyable<T>(t)) {}

  // exemplary regular operations
  ~wrapper_copyable() { delete c; }
  wrapper_copyable& operator=(const wrapper_copyable&);

  // use of the concept interface
  wrapper_copyable(const wrapper_copyable& rhs)
  : c(&rhs.c->clone()) {}

  concept_copyable* c;
};
```

That is, wrapper_copyable is what a user can use to create objects that can be copied. for example:

```
wrapper_copyable val(1); // stores 1 in val
wrapper_copyable copy(val); // creates a copy of val
```

The wrapper_copyable constructor deduces the type of the model_copyable instance needed to access the data through concept_copyable. It also creates the model_copyable object needed to hold the value.

4. RUNTIME POLYMORPHIC STL

This section presents our implementation of the runtime concept idiom for iterators and several loosely coupled algorithms. The same techniques can be applied to provide runtime polymorphic STL containers.

4.1 Runtime Concepts of Iterators

To allow for the modeling of iterator concepts, we must extend the idiom from §3 to support concept-layer and model-layer refinements. Following Marcus et al. [14], we employ inheritance in order to support concept and model refinements while minimizing source code duplication.

We illustrate the implementation of runtime concepts and models using the concept forward-iterator and its refinement bidirectional-iterator. For the wrapper class layer implementation we also refer the reader to [14].

4.1.1 Concept Interface Refinements

Runtime concept interfaces are essentially abstract base classes that define a set of function signatures but do not have data members. The code that follows omits the template parameters that corresponds to iterator::reference.

```
template <class ValueType>
struct concept_forward {
  virtual void operator++() = 0;
  virtual concept_forward& clone() const = 0;
  // ...
};

template <class ValueType>
struct concept_bidirectional : concept_forward<ValueType> {
  virtual void operator--() = 0;
  virtual concept_bidirectional& clone() const = 0;
  // ...
};
```

Refinement of runtime concepts are accomplished via inheritance from a base concept class and add new signatures or refine inherited signatures with a covariant return type. Prefix and postfix operators share the same member function declarations and return **void**. The semantically correct implementation of the return value is left to the wrapper classes. Types related to the elements stored inside the container are passed as template arguments (e.g.: value_type).

4.1.2 Model Refinements

Models implement the abstract operations of the runtime concept for concrete types. At the root of the model-hierarchy is the model_base that stores a copy of the concrete iterator.

```
template <class Iterator, class IterConcept>
struct model_base : IterConcept {
  Iterator it;
};
```

The first template argument determines the concrete iterator type. The second template argument corresponds to the concept interface that this model will implement. For an iterator of list<**int**>, these would be list<**int**>::iterator and concept_bidirectional<**int**>, respectively.

```
template <class Iterator, class IterConcept>
struct model_forward : model_base<Iterator,IterConcept> {
  IterConcept& clone();

  bool operator==(const concept_forward& rhs) const
  {
    assert(typeid(*this) == typeid(rhs));
    // ...
  }
  // ...
};

template <class Iterator, class IterConcept>
struct model_bidirectional : model_forward<Iterator,IterConcept> {
  void operator--();
  // ...
};
```

Each model refinement implements the operations defined in the corresponding concept interface. The meaning of the two template arguments Iterator and IterConcept is the same as for model_base. Binary operations (e.g.: **operator**==) require the second argument to have the same dynamic type as the receiver. Conversions could be encoded with double dispatch [7] but are currently not supported.

4.1.3 Model Selection

The function select_model selects the model refinement based on the iterator type tag of T.

```
template<class Iterator>
typename map_iteratortag_to_concept_interface<Iterator>::type*
select_model(const Iterator& it);
```

The template meta function map_iteratortag_to_concept_interface maps the iterator tag to a runtime model. For example, bidirectional_iterator_tag is mapped on model_bidirectional. select_model instantiates this model with the iterator type and the correct concept interface. For list<**int**> this would be model_bidirectional<list<**int**>::iterator, concept_bidirectional<**int**>>. Finally, select_model constructs the model on the heap, and returns a pointer to it.

4.2 The Algorithms Library

The algorithms library prototypes a runtime counterpart of several STL algorithms. Each function in the library originates from an algorithm instantiation with one of our iterator wrappers or a concrete iterator. By default, the library contains an instance for the weakest concept an algorithm supports. For example, the default entry for lower_bound would be instantiated with wrapper_forward. These minimal instantiations are meant to serve as fallback-implementations. To improve performance, the system integrator or even a (dynamically loaded) library can add more specialized functions.

The algorithms are defined in terms of existing STL algorithms and iterator-value_types (e.g.: algolib::advance<**int**>). Consider a library defined on advance and **int** that by default contains an instance for forward_iterator. The sample code adds one generic implementation for wrapper_randomaccess and a specific for list<**int**>::iterator.

```
// add generic implementation suitable for all randomaccess iterators.
algolib::add_generic<
    algolib::advance<int>, // library name
    wrapper_randomaccess<int> // iterator-type
  >();
// add specific implementation for std::list<int>.
algolib::add_specific<
    algolib::advance<int>, // library name
    std::list<int>::iterator // iterator-type
  >();
```

In addition, we provide library access functions with names that match their STL counterparts. The access functions are defined in the same namespace as the wrapper classes. Together with argument dependent look-up (ADL), this enables seamless integration of runtime concepts into user code. The code snippet shows a function that takes two iterator wrapers as arguments and calls the library access functions (i.e.: distance, advance).

```
wrapper_forward<int>
random_elem(wrapper_forward<int> f, wrapper_forward<int> l)
{
  wrapper_forward<int>::difference_type dist = distance(f, l);
  return advance(f, rand() % dist);
}
```

At runtime, a library call selects the best applicable function present based on the dynamic type of the model. Starting with the **typeid** of the actual iterator model, it walks the **typeid**s of the inheritance chain until it finds a suitable algorithm instance or the fallback implementation.

If in our example first and last wrap the concrete type std::list<**int**>::iterator, the dispatch mechanism will peel off all runtime concept layers and call std::advance with a std::list<**int**> iterator. In case the iterators in first and last belong to a std::vector, the runtime model is re-wrapped by a wrapper_randomaccess iterator and std::advance<wrapper_randomaccess> is invoked.

Although the dispatch mechanism is semantically equal to virtual function calls, we rejected alternative library designs, which would model algorithms as pure virtual functions that are declared in concept interfaces. This would break the separation between concept requirements and algorithms. Providing a new algorithm would require adding a new function signature to the concept interface, thereby breaking binary compatibility with existing applications. In addition, such a design would create a number of unused instantiated functions. For example, the class concept_forward would need virtual function declarations for all STL algorithms that are defined for forward iterators (e.g.: adjecent_find, destroy, equal_range, etc.). Consequently, the model classes would need to implement those functions regardless whether a specific program uses them or not.

4.3 Retroactive Runtime Concepts

The goal of retroactive concept modeling is to place requirements on the element type of containers after the container has been declared (and elements have been inserted). This would allow us to write code that operates on a variety of containers and element types. Instead of the introductory example (§1) we would like random_elem to simultaneously handle list<**int**>, vector<**double**>, etc.:

```
retrowrapper_forward<>
random_elem(retrowrapper_forward<> f, retrowrapper_forward<> l)
{
  retrowrapper_forward<>::difference_type dist = distance(f, l);
  return advance(f, rand() % dist);
}
```

For reads and writes through such iterators the well studied variance problems [11] apply. However, sequence-modifying operations (e.g.: sort, rotate, etc.) which only permute the

elements and element modifications through a runtime concept interface are type safe.

Since the concrete element type of the containers are not known, functions that access elements cannot be implemented in terms of any concrete type. Instead, we manipulate the elements through proxy_reference<C> objects. A proxy_reference<C> object maps the element-access operations of the runtime concept C onto the elements. While this is sufficient for manipulating the values through interfaces, this technique fails when argument type deduction is involved.

The original design of the STL tried to allow for arbitrary proxy types by including reference as one of the traits for an iterator but this was found to be insufficient in general. To see this problem, we look at the implementation of swap:

```
template <typename T>
void swap(T& x, T& y)
{
  T tmp(x);
  x = y;
  y = tmp;
}
```

If T is a proxy_reference, then this code will swap the two proxy_references, not the underlying values. What we would like is that the syntax T& would match *any type which is a reference to T* not generate a reference to the proxy type.

The best that we are able to easily achieve is a proxy_reference which behaves as a reference when the referenced value is not mutable. To test the ideas presented in this paper with the standard algorithms we used the following single shared reference scheme:

- proxies maintain a count of the number of proxies referring to a single value.
- when assigning through a proxy if the reference count is greater than one, then a copy of the value is made and all *other* proxies referring to the value are set to refer to the copy.

This relies on the fact that it would be inefficient to make a copy of a value and then assign over it. This is a very fragile and costly solution but it was sufficient to test the ideas in this paper. Solving the proxy dilemma properly in C++ is an open problem.

5. TESTS

To assess the performance cost of runtime concepts we tested the approaches described in sections §3 and §4. The numbers presented in this section were obtained on an Intel Pentium-D (2.8GHz clock speed; 512MB of main memory at 533 MHz) running CentOS Linux 2.6.9-42. We compiled with gcc 4.1.1 using −O3 and −march=prescott. Initially, the vector contained 8 million numbers in ascending order starting from zero. Then we invoke four algorithms: reverse, find of zero, sort, and lower_bound of zero.

vector<T>: As reference point for our performance tests we use the vector instantiated with a concrete type. The table shows the number of cycles each operation needs to complete for a container of **int** and **double** respectively. The column to the right of the number of cycles shows the slowdown factor compared to vector<**int**>. The algorithms with O(n) runtime complexity (i.e.: reverse and find) run approximately twice as long when used with type **double**. This discrepancy can be explained by the size of the stored datatype; **double** is twice as big as **int**.

	int		**double**	
reverse	50135428	1	101270708	2.0
find	24970288	1	52763884	2.1
sort	569752400	1	1170042300	2.1
lower_bound	5628	1	14784	2.6

vector<Scalar>: For this scenario, we defined a concept Scalar that extends the concept Copyable §3 with a function that allows conversions to type **double**. This conversion function is used to implement the concept LessThanComparable required by sort and lower_bound. The test uses an overloaded swap operation to efficiently swap the pointers (same size as **int**) of the polymorphic object-parts. Thus, the runtime of reverse is comparable to the optimal case; and beats the runtime for **double** by a factor of 2. The slowdown of find can be traced back to the equals-operator, which uses one virtual function call and one type-test per iteration. The sort algorithm requires two virtual function calls for each less-than comparison. Compared to the overhead of virtual function calls and type tests, the size of the element type is insignificant.

	int		**double**	
reverse	50260070	1.0	50112846	1.0
find	243300778	9.74	279807066	11.21
sort	10719950962	18.82	11098532844	19.48
lower_bound	20482	3.64	17626	3.13

Sequence<T>: The following table shows the results, when the algorithm library contains instantiations for concrete iterators. The time needed to select the best match is the only overhead that occurs.

	int		**double**	
reverse	50017254	1.0	101235652	2.0
find	24093454	1.0	51413502	2.1
sort	543942098	1.0	1163111236	2.0
lower_bound	17206	3.1	20286	3.6

Only when instances for the concrete iterators are missing, our system resorts to fallback implementations.

	int		**double**	
reverse	4736673970	94.5	4763215828	95.0
find	467545414	18.7	525890540	21.1
sort	16684763676	29.3	16980691560	29.8
lower_bound	19040	3.38	20552	3.7

The 94x slower performance for reverse is unacceptable, even for a fallback implementation. A closer analysis of the fallback test on reverse reveals that three factors contribute to its slowness: the fallback algorithm is bidirectional iterator based, virtual iterator functions, and model allocation on the heap.

To pinpoint and quantify the contribution of each of these factors we performed additional experiments: we started by adding a reverse function operating on randomaccess−iterator concepts, which improved the performance marginally to the factors 91.5x and 94.8x for int and doubles respectively. Each iteration of reverse has one call to std::iter_swap. The

gcc implementation of std::iter_swap calls another function that swaps the two elements to which the iterators point. Each function invocation creates copies of the iterators, which results in 16 million *unnecessary* heap allocations (and deallocations). By providing our own reverse implementation, we eliminated those copies. Then, reverse is only 7.4x slower for **int** (7.6x for **double**). The measured slowdown is less on other architectures (3x on a Pentium-M). Instead of rewriting the STL algorithm, we could adopt Adobe's small object optimization [14] where the wrapper classes reserve a buffer to embed small objects (Adobe's open source library [3]).

vector<RetroactiveScalar>: The following table shows the performance results for retroactively imposing runtime concepts on container elements (§4.3).

	int		**double**	
reverse	14580637680	290	14702289350	293
find	5525953258	221	5691982996	228
sort	145095555976	254	147783100836	259
lower_bound	30100	5.3	31010	5.5

Since the overhead of operating on Sequences has been determined by the previous test, we tested the performance of our proxyref-implementation by dispensing with runtime concepts for iterators. Instead, we use an iterator-class that has non-virtual access functions and wraps a vector::iterator. As described in §4.3 the iterator abstracts the concrete element type and operates on proxy_references. We used boost::shared_ptr to implement the single shared reference semantics. In the case of reverse, the poor performance is caused by the fact that each iteration requires the construction and destruction of five proxy_reference objects: two when the iterators are dereferenced, two when swap is invoked, and one when a temporary element inside swap is constructed (in total 20 million). Each of these operations performs an update of the shared_ptr. In addition, each temporary element is allocated on the heap (4 million).

6. RELATED WORK

The ASL [3] introduced the runtime concept idiom, employing type erasure [2] to provide the *any regular* library (similar to the boost any library), and its generalization, the *poly* library. The *poly* library generalizes the idiom to support refinement and polymorphic downcasting, encapsulates the common tasks required to create non-intrusive runtime-polymorphic value-based wrappers. The *poly* library design goals and implementation are elaborated in [14].

ASL also provides the *any iterator* library offering runtime-polymorphic iterators for specific types as a proof of concept. Becker [6] presents a similar library. In this paper we focus more on performance.

Bourdev and Järvi [8] discuss a mechanism for falling back to static-dispatch when type erasure is present.

Our work extends the above results to a library of algorithms operating on runtime-polymorphic containers of runtime-polymorphic elements, achieving realistic performance levels by using static dispatch where possible.

7. FUTURE WORK AND CONCLUSION

In the described system, an algorithm library handles the dispatch to the most appropriate algorithm present in the system. Comparable to function overload resolution, the template instantiation mechanism in C++0x considers the concept of all arguments to determine a unique best match. The current implementation of our algorithms, however, finds the best match based on the type of only one argument. Other arguments (i.e.: the second iterator in calls to reverse, sort, etc.) are obliged to conform to a type that the first argument determines. The presented algorithms are extensible with new iterators and sequences, unknown at the compile time, as long as they conform to the STL-defined concepts. Subsequent work is expected to support multiple dispatch and provide for modular runtime concept refinements. Iterator types for which repeated algorithm invocations frequently resolve to the fallback implementations would benefit from a runtime system that is capable of dynamic algorithm instantiation.

In this paper, we have discussed a runtime polymorphic version of several STL algorithms. Our implementation enables us to use STL's sequences where the binding to a concrete data structure is deferred until runtime. To improve runtime performance, if the data structures in use are known to the system integrator, our algorithms can leverage static dispatch. As a result, the runtime overhead becomes negligible for large data sets. We discussed retroactive runtime concept imposition on container elements and pointed out where better language support would be needed to further the runtime generic programming model.

8. ACKNOWLEDGMENTS

We thank Jaakko Järvi, Damian Dechev, Yuriy Solodkyy, Luke Wagner and the anonymous referees for their helpful suggestions.

9. REFERENCES

[1] The Boost C++ libraries, 2002. http://www.boost.org/.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[3] Adobe System Inc. Adobe Source Library. http://opensource.adobe.com, 2005.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *LCPC '01*, pages 193–208, Cumberland Falls, Kentucky, Aug 2001.

[5] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[6] T. Becker. Type erasure in C++: The glue between object oriented and generic programming. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07*, July 2007.

[7] L. Bettini, S. Capecchi, and B. Venneri. Double dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.

[8] L. Bourdev and J. Järvi. Efficient run-time dispatching in generic programming with minimal

code bloat. In *Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon*, Oct. 2006.

[9] J. C. Dehnert and A. A. Stepanov. Fundamentals of Generic Programming. In *Selected Papers from the International Seminar on Generic Programming*, pages 1–11, London, UK, 2000. Springer-Verlag.

[10] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM Press.

[11] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *ECOOP '02: Proceedings of the 16th European Conf. on Object-Oriented Programming*, pages 441–469, London, UK, 2002. Springer-Verlag.

[12] ISO/IEC 14882 International Standard. *Programming languages: C++*. American National Standards Institute, September 1998.

[13] J. Järvi, M. A. Marcus, and J. N. Smith. Library composition and adaptation using C++ concepts. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA, 2007. ACM Press.

[14] M. Marcus, J. Järvi, and S. Parent. Runtime polymorphic generic programming—mixing objects and concepts in ConceptC++. In K. Davis and J. Striegnitz, editors, *Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07*, July 2007.

[15] S. Parent. Beyond objects: Understanding the software we write. Presentation at C++ connections, November 2005.

[16] S. Parent. Concept-Based Runtime Polymorphism. Presentation at BoostCon, May 2007.

[17] T. L. Veldhuizen. Arrays in Blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag.

Why C++ is not just an Object-Oriented Programming Language

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

C++ directly supports a variety of programming styles. In this, C++ deliberately differs from languages designed to support a single way of writing programs. This paper briefly presents key programming styles directly supported by C++ and argues that the support for multiple styles is one of its major strengths. The styles presented include: traditional C-style, concrete classes, abstract classes, traditional class hierarchies, abstract classes and class hierarchies, and generic programming. To provide a context for this overview, I discuss criteria for a reasonable and useful definition of ‘‘object-oriented programming.’’

1 Introduction

There are many tools and techniques that can help in our effort to build useful, economical, and maintainable systems. To complete ambitious and complex projects, we rely on a wide variety of techniques and tools that must work together.

The title of this paper singles out a programming language†. However, the real topic is programming, or if you prefer a longer formulation, the design and implementation of systems. A programming language is just one of the means by which we try to achieve our goals.

The definition of ‘‘object-oriented programming’’ is no longer a popular topic of discussion at major conferences. A practical definition of ‘‘object-oriented programming,’’ ‘‘object-oriented analysis,’’ ‘‘object-oriented design,’’ ‘‘object-oriented technology,’’ etc., is, however, a burning issue for people who want to turn the oft-repeated promises made for techniques and languages called ‘‘object-oriented’’ into reality in everyday projects. It has become a practical rather than academic topic of discussion. What is ‘‘object-oriented technology?,’’ what benefits can be expected from it? at what risks?, how do those techniques, benefits, and risks compare with those associated with alternatives?

A systems builder trying to explain to an accountant why money should be spent for tools supporting object-oriented techniques needs more than a statement to the effect that ‘‘object-oriented is great’’ or that ‘‘really great techniques are really object-oriented.’’ You simply cannot ask someone to bet their company’s future on vague promises phased in ill-defined terms. Nor is a well-polished and logically coherent semi-mathematical treatment of the

† This paper is primarily based on an invited talk with the same title given at OOPSLA’95 in Austin Texas. The style of this paper is clearly affected by its origins as a relatively short talk. I would have preferred this paper to be either much longer or much shorter, but I did not have the time to do either.

subject of direct practical use.

We need to define ‘‘object-oriented’’ to be something specific so that we can point out specific benefits and risks associated with its use. We must also be specific about what is *not* object-oriented, and what benefits and lack of benefits we can expect from various non-object-oriented techniques.

Consequently, this paper starts out discussing what makes a good definition of ‘‘object-oriented.’’ Next, I present a range of useful techniques which may or may not be object oriented and discuss their advantages and disadvantages.

2 Defining ‘‘Object-oriented’’

To be useful and intellectually honest, a definition of ‘‘object-oriented’’ must

[1] not be a mere synonym for ‘‘good,’’
[2] not exclude most accepted meanings,
[3] have a firm historical basis,
[4] exclude something.

Not everything good is object- oriented, and not everything object-oriented is good. I think I can support both claims from experience. I have seen examples of the latter often enough: it is not uncommon to find programs that apply techniques usually deemed object-oriented extensively or even exclusively, yet are hard to comprehend, hard to maintain, and perform abysmally. Such examples occur in every programming language. But then, of course, some people respond ‘‘that just proves that the program wasn’t *truly* object-oriented.’’ To which the answer must be that either the term has become meaningless or there must be something good beyond what is called ‘‘object-oriented.’’

On the other hand, when we define ‘‘object-oriented,’’ we must not be too exclusive. Object-oriented programming is a broad intellectual discipline, not the mere use of specific language features. Attempts to define ‘‘object-oriented’’ to mean ‘‘what I’m selling’’ are not uncommon, but are fundamentally sleazy.

Any definition of ‘‘object-oriented’’ should be historically reasonable. Words are only useful for communication, really only mean something, if we agree on a meaning for them. There are several plausible, logically coherent, and mutually contradictory definitions of ‘‘object oriented’’ in use. However, the mainstream usage stems directly from the ideas pioneered by programming language Simula and the design techniques it was developed to support. The communities of programmers and designers centered around languages such as C++, CLOS, Eiffel, Object Pascal, and Smalltalk have contributed much to this tradition.

A meaningful definition of any concept must exclude something.

3 A Broad Definition of ‘‘Object-oriented’’

Given these general criteria for a definition of ’’object-oriented’’ you can find several plausible candidates, and several communities have their own definitions. However, I suggest we stick to the traditional definition of object-oriented used within broad communities of programmers. A language or technique is object-oriented if and only if it directly supports:

[1] Abstraction − providing some form of classes and objects.
[2] Inheritance − providing the ability to build new abstractions out of existing ones.
[3] Run-time polymorphism − providing some form of run-time binding.

This definition includes all major languages commonly referred to as object-oriented: Ada95, Beta, C++, CLOS, Eiffel, Simula, Smalltalk, and many other languages fit this definition. Classical programming languages without classes, such as C, Fortran4, and Pascal, are excluded. Languages that lack direct support for inheritance or run-time binding, such as Ada88 and ML are also excluded.

ML is a good example of something that is good but not object-oriented. I like ML; it is an interesting, innovative, and powerful language, but it is functional rather than object-oriented and its polymorphism is resolved at compile time rather than at run-time. Thus, saying that ML isn’t object-oriented is not a criticism, it’s an observation about definitions and the nature of ML.

Techniques and tools are object-oriented if and only if they support the use of object-oriented programming. For example, a design method is object-oriented if its regular and proper use leads to

programs that exploit abstraction, inheritance, and polymorphism where appropriate. I strongly prefer design methods that directly and naturally support the use of at least one of the major object-oriented languages supporting in ways that exploit its features in an idiomatic way.

For example, it is often possible to simplify application code by hiding objects with different representations and different implementation details behind a common ‘‘abstract’’ interface (see §6.4 and §6.6). Conversely, the implementation of related concepts can often be greatly simplified by exploiting commonality through inheritance (see §6.5 and §6.6). A major purpose of design methods and the CASE tools that commonly support them is to make design simpler, more regular, and more predictable. Thus, to earn the label ‘‘object-oriented,’’ a design method must regularly and predictably help the discovery of commonality that can be exploited in these ways. Ideally, an object-oriented design method must strongly encourage the expression of this commonality using the most appropriate facilities in one or more of the languages supporting object-oriented programming. Minimally, the method and its supporting tools must not be a hindrance to the use of object-oriented facilities in the programming language used to implement the design. Much confusion arise because not every design method that claims to be object-oriented does that.

Please remember that I’m looking for a practical understanding of the notion of ‘‘object-oriented’’ rather than a formal definition. A formal definition is useful, indeed it may be essential. However, to be relevant, a formal definition must match a coherent view of what the formal definition is meant to specify precisely.

4 Purity

There has been much debate about ‘‘purity,’’ in the context of languages supporting object-oriented programming. In my opinion, much of that discussion is confused by the − often unstated − assumption that not only does ‘‘object-oriented’’ imply ’’good,’’ but also by the further assumption that only ‘‘object-oriented’’ features are good. Consequently, it is − wrongly − assumed that a language that provides features deemed non-object-oriented must be worse than a language that does not. People who like a language to support classes as part of a hierarchy only and functions/methods attached to one specific class only often call such a language ‘‘a pure object-oriented language.’’ If we don’t like the idea of restricting the definition of classes and functions that way we can call such a language ‘‘just an object-oriented programming language.’’

I prefer to have more facilities available than can be provided by methods defined on classes within a single hierarchy. A lot of good design goes beyond that relatively narrow domain. Incidentally, I have come to dislike the adjective ‘‘hybrid’’ as used to distinguish ‘‘pure’’ object-oriented systems from others. Too often, ‘‘hybrid’’ is used in a prejudicial manner. If I must apply a descriptive label, I use the phrase ‘‘multi-paradigm language’’ to describe C++.

4.1 Use of Language Features

Even when all the features required to support object-oriented programming are available, you don’t need to use them all the time. Some classes just don’t belong in a hierarchy and some functions don’t belong to any particular object.

The key to maintainable, efficient, and evolvable programs isn’t particular language features. It is the ability to develop concepts needed for a solution and to express them clearly in a program. Language features exist to make such expression simple and direct.

Object-oriented programming can be done in a language lacking one or more of the features required to directly support object-oriented programming. However, doing so is unnecessarily difficult, very difficult to support with tools, and often prohibitively expensive.

Furthermore, there are things that can’t be expressed directly using only the ‘‘pure’’ object-oriented constructs mentioned above. For example, some entities belong together, but their relationships are not hierarchal. Some entities simply do not obey the rules of a particular object-oriented language. Some things that you build in an object-oriented world are manipulated from the outside so that it is difficult to make guarantees about the way they are used.

5 C++ Design Ideals

I felt the need for facilities outside what is conventionally called ‘‘object-oriented,’’ so I supplied some in C++. However, C++ isn’t meant to be everything to everybody. No one programming language and no one view of how to write programs is sufficient for everything. Constraints-based programming, logic programming, functional programming, and various forms of concurrent programming are examples of good and useful styles of programming not supported by C++.

No single language can support every style. However, a variety of styles can be supported within the framework of a single language. Where this can be done, signifcant benefits arise from sharing a common type system, a common toolset, etc. These technical advantages translate into important practical benefits such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages.

C++ was designed to support a range of styles that I considered fundamentally good and useful. Whether they were object-oriented, and in which sense of the word, was either irrelevant or a minor concern:

[1] Abstraction − the ability to represent concepts directly in a program and hide incidental details behind well-defined interfaces − is the key to every flexible and comprehensible system of any significant size.

[2] Encapsulation − the ability to provide guarantees that an abstraction is used only according to its specification − is crucial to defend abstractions against corruption.

[3] Polymorphism − the ability to provide the same interface to objects with differing implementations − is crucial to simplify code using abstractions.

[4] Inheritance − the ability to compose new abstractions from existing one − is one of the most powerful ways of constructing useful abstractions.

[5] Genericity − the ability to parameterize types and functions by types and values − is essential for expressing type-safe containers and a powerful tool for expressing general algorithms.

[6] Coexistence with other languages and systems − essential for functioning in real-world execution environments.

[7] Run-time compactness and speed − essential for classical systems programming.

[8] Static type safety − an integral property of languages of the family to which C++ belongs and valuable both for guaranteeing properties of a design and for providing run-time and space efficiency.

These facilities and general properties can be supported in several alternative ways. For example, one programming language may support a facility in its core language where another supports it in a library. Similarly, a facility provided by a run-time mechanism in one language may be provided by a compile-time mechanism in another.

The requirement for coexistence is essential for any language claiming to be general-purpose. Looking at the world from the perspective of a given programming language, we find that almost every real-life system contain parts that are written in others languages and designed according to principles foreign to that language. To be general-purpose, a language must somehow take the unpredictable, ugly, and constantly changing demands of program fragments written in ‘‘other languages’’ into account.

To be genuinely general purpose, a language must possess facilities that allow it to share data with program fragments written in other languages, to invoke code fragments written in other languages, and have code invoked by code written in other languages. For example, systems relying on callbacks can be rather ugly to program, but not being able to use such systems in a direct and idiomatic way would be crippling for a language as a tool for real-world programming. In many languages, a common use of ‘‘foreign’’ code is exactly to violate the languages rules: to do things that can’t be done − or can’t be done efficiently − in the language itself.

Alternatively, access to facilities in ’’the outside world’’ could be carefully fitted into the framework of the object-oriented programming language through special facilities in the run-time environment or in libraries. However, accessing facilities in the manner they were meant to be used is often

easier and less awkward than to fit them into our language framework. A general mechanism for accessing ‘‘foreign’’ code also leads to more extensible systems than a requirement to fit each individual ‘‘foreign’’ facility into the language framework.

Over the years, we have seen spectacular improvements both in hardware performance and in compilation techniques. However, run-time efficiency and compact representation is still absolutely essential to many people.

Static type safety is an essential part in my view of both design and implementation (see, for example [Stroustrup,1991]). The guarantees provided and the discipline of design imposed have been found extremely valuable by many people working in a wide range of application areas. Static type checking is of course not a panacea, but it is something I would not attempt major projects without.

The fundamental ideal of C++ is actually the fundamental ideal for a lot of languages:

center box; c. Represent concepts and relationships between concepts directly and affordably.

Naturally, there are many ways of approaching this ideal. It is worth remembering that all of the languages usually mentioned in a discussion of practical use of object-oriented techniques are suitable vehicles for good design. A rational discussion of languages is one of relative merits, applicability to specific problem areas, and personal preferences, rather than one of absolutes.

Representing concepts directly is a restatement and possibly a generalization of ideas relating to data abstraction and information hiding. Representing hierachical relationships is the traditional key to object-oriented programming. There are, however, clean and useful relationships that are not hierarchical yet can still be represented directly in a program (for example, see §6.3 and §6.7).

Being more concerned with producing good software than with finding the most elegant expression of ideas in the abstract, I insist on affordability. Affordability is a multi-facetted issue that involves not only run-time efficiency, but also availability of suitable hardware, availability of designers and programmers comfortable with new techniques, etc.

6 Programming Style and Language Features

I will now give examples of programming styles and language features supporting them. Some are commonly referred to as ‘‘object-oriented,’’ some are not, but that doesn’t prevent me from recommending them in some contexts.

6.1 Conventional Notation

There are aspects of conventional code and conventional notation that I would like to see maintained even in a strictly object-oriented overall design. Being able to say plain square root of two, `sqrt(2)` is nice, and so is the ability to write `x+y*z` and know that it means add `x` to the product of `y` and `z`. We have about 400 years of experience with such notation and it is deeply ingrained in our technical culture.

6.2 Concrete Types

Very simple concepts, such as integers, floating point numbers, complex numbers, points, lines, pairs, dates, disk locations, bcd characters, error messages, currency, are usually not considered suitable topics for discussion in academic articles or at conferences. These are usually considered too simple to merit discussion. However, the mundane is often statistically more significant than the sophisticated.

Provided they can be implemented in a way that is simple, elegant, efficient and flexible enough, I consider such simple concepts excellent candidates for independent proper types − as opposed to presenting them to users as plain data structures or as parts of a larger class hierarchy. Consequently, part of a design effort should focus on these little abstractions. These very concrete types should be designed carefully and supported well.

To illustrate why, I’ll contrast this approach to the use of a plain data structure and to the use of class hierarchies. To make this discussion concrete, I’ll use the example of a date.

6.2.1 Structures and Functions

The simplest way of presenting a date in a program is simply to specify its data layout. For example:

```
struct date {
    // representation
};
```

Given that, programmers can do anything at all with `date`s. That ''anything at all'' is the strength and weakness of this idea. Naturally, a ''standard'' set of functions is usually provided to manipulate a structure such as `date`. However, such a set of functions is rarely complete, and even when it is, programmers find reasons to manipulate `date`s directly. Consequently, it is usually not possible to change the definition of `date` after the initial release of the software; it is simply too difficult to track down every use of a `date` and modify it use to the new definition.

The reason that the set of functions is rarely complete is that there is no incentive to make it so. A programmer can always write new functions accessing the `date` structure, and the dominant culture encourages the programmer to do so. Writing a new function that is ''just right'' for the job, carries no overhead, and relies on no potentially untrustworthy code is often considered better than improving a standard and general-purpose set of access functions and using it. Often, it is also far easier. This trend is typically reinforced by poor documentation.

6.2.2 A Concrete Class

A simple `Date` type can remedy most of the problems related to using a data structure directly. Consider:

```
class Date {
public:
    // public interface, consisting
    // of non-virtual functions
private:
    // representation and other
    // implementation details
};
```

Such a `Date` type will provide

[1] constructors specifying how objects of the type are to be initialized;

[2] functions for examining a `Date`; these functions will be explicitly declared not to modify the value of the object;

[3] functions for manipulating `Date`s without actually having to know the details of the representation or fiddle with the intricacies of the implementation.

In addition, `Date`s can be freely copied.

This set of member functions supplied as members of `Date` should be those that provide a basic semantics for a `Date` and also requires direct access to the representation of `Date` to be implemented.

The set of member functions should be almost minimal; many operations that users would find convenient can be supplied separately (see §6.3). I dislike classes with dozens or even hundreds of member functions. Such a class does not represent a well-thought-out concept; it's a glorified data structure produced by somebody who couldn't decide on what was really wanted.

The member functions are declared non-virtual to ensure that there is no time or space overheads involved in using this `Date`, and to ensure that the semantics of `Date` cannot be modified later. Similarly, the representation of `Date` is declared `private` to prevent access by any function not explicitly mentioned in the class itself.

The representation of a concrete type should be compact. Sometimes millions of objects of such classes exist, and even with modern memory sizes space overheads can be a burden. If nothing else, reading and writing objects with bloated representations can be a nuisance.

The use of concrete types must be fast. In my world at least, programmers are prone to represent something as plain data structure out of fear of overheads supposedly associated with abstractions.

There are no time or space overheads associated with the `Date` class as defined above. The size is identical to that of the plain `date` structure, and inlining is done for simple member functions to make these as fast as the code a programmer would write accessing a plain structure directly.

Often, it is important that simple concrete types, such as `Date`, be layout-compatible with simple data structures, such as `date`, as used in traditional languages. This allows simple exchange or sharing of information with code written in traditional

languages. This can be a major convenience if your operating system, your database, or your high-performance numeric library is written in a traditional language and requires manipulation of data of a specific layout.

The `Date` class is very simple and very basic. It requires no elaborate framework, no class hierarchies, no clever dispatcher to mediate access, etc. It doesn't affect the overall structure of a program much; it just provides a lot of help at the detailed programming level − below the level of detail of interest to most managers and to many designers.

If such types are that simple, why bother spending time on them?

[1] The concepts best represented by such simple types are common; most applications can use a few dozen or a few hundred such types. Thus, any benefits we get from a single concrete type, we get many times over.

[2] The problems relating to lack of encapsulation of plain structures (§6.2.1) are eliminated.

[3] The replication of effort writing simple access functions is eliminated.

[4] Making a concrete type the subject of a conscious design effort typically results in a better thought out, more comprehensive, and better documented concept. In principle, this could equally well be done for the plain structure approach, but in practice that typically doesn't happen.

[5] Writing the basic functions of a concrete type is not difficult, but it is not trivial either. For example, adding a year to a date requires us to handle leap years. By relying on common access functions, we eventually achieve an implementation that has been better thought out and has fewer errors.

[6] Since more of the implementation is documented and shared, user code becomes more uniform. Thus, code written by others become easier to comprehend.

These are classical reuse benefits and I don't think we should decline them just because they are easy to obtain.

6.2.3 Re-using Concrete Types

For many concrete types, derivation doesn't make sense. Consider, deriving a new class from `Date`:

```
class MyDate : public Date {
    // …
};
```

Is it ever valid for `MyDate` to be used as a plain `Date`? Well, that depends on what `MyDate` is, but in my experience it is rare to find a concrete type that makes a good base class without modification.

Derivation from a concrete type is almost always a mistake. A concrete type is a self-contained entity that can't easily added to in a way that makes sense. A concrete type is ''re-used'' unmodified in the same way as built-in types such as `int` are. For example:

```
class Date_and_time {
public:
    // …
private:
    Date d;
    Time t;
};
```

This form of use (reuse?) is usually simple, effective, and efficient.

Maybe it was a mistake not to design `Date` to be easy to modify through derivation? It is sometimes asserted that *every* class should be open to modification by overriding and by access from derived class member functions. This view leads to a variant of `Date` along these lines:

```
class Date2 {
public:
    // public interface, consisting
    // primarily of virtual functions
protected:
    // representation and other
    // implementation details
};
```

Here, the functions are declared `virtual`, meaning that a class derived from `Date2` (in the style of `MyDate` above) can provide its own versions. To make it possible to write such overriding functions easily and efficiently, the representation is declared `protected`. A `protected` member of a class is accessible not just to the classes' own members, but

also to the member functions of derived classes.

This achieves the objective of making `Date2` arbitrarily mallable by derivation, yet keeping its user interface unchanged. However, there are costs:

[1] Efficiency of basic operations − a C++ virtual function call is a fraction slower than an ordinary function call, virtual functions cannot be inlined as often as non-virtual functions, and a class with virtual functions typically incurs a one word space overhead.

[2] Need to use free store − the aim of `Date2` is to allow objects of different classes derived from `Date2` to be used interchangeably. Because the sizes of these derived classes differ, the obvious thing to do is to allocate them on the free store and access them through pointers or references. Thus, the use of genuine local variables dramatically decreases.

[3] Inconvenience to users − to benefit from the polymorphism provided by the virtual functions, accesses to `Date2`s must be through pointers or references.

Naturally, these costs are not always significant, and as we will see in §6.4 the behavior of a class defined in this way is often exactly what we want. However, for a simple concrete type, such as `Date2`, the costs are unnecessary and can be significant.

Please note that the costs are fundamental; different languages present the facilities differently, but every language that provides run-time polymorphism incur these costs in some way or other.

Finally, a well-designed concrete type is often the ideal representation for a more mallable type. For example:

```
class Date3 {
public:
    // public interface, consisting
    // primarily of virtual functions
protected:
    Date d;
};
```

6.3 Namespaces

Class hierarchies express (hierarchical) relationships, but not every relationship in a program can or should be expressed as a hierarchical relationship between classes. For example, if a class is intented for use only in the context of another class it can be declared a member of that class exactly the way a function can be:

```
class Date {
public:

    enum Month {
        jan, feb, mar,
        apr, may, jun,
        jul, aug, sep,
        oct, nov, dec
    };

    // …
};

Date::Month m = Date::nov;
```

More generally, C++ provides namespaces for grouping declarations [Stroustrup,1994]. For example, many operations on `Date`s shouldn't be members of class `Date` because they don't need direct access to the representation of a `Date`. Providing such functions as non-member functions leads to a cleaner `Date` class, but we would still like to make the association between the functions and the class explicit:

```
namespace Chrono {

    // facilities for dealing with time:

    enum Month {
        // …
    };

    class Date {
        // …
    };

    int diff(Date a, Date b);
    bool leapyear(int y);
    Date next_weekday(Date d);
    Date next_saturday(Date d);

    // …
}
```

A namespace is not a module; it is not an object. A namespace is a general scope mechanism to support a variety of techniques related to modularity. Not incidentally, namespaces provides a way of avoiding

name clashes in software composed out of libraries from different suppliers. For example:

```
namespace LibA {
    class String {
        // A-style string
    };
    // ...
}

namespace LibB {
    class String {
        // B-style string
    };
    // ...
}

LibA::String s1 = "Nicholas";
LibB::String s2 = "Annemarie";
```

6.4 Abstract Classes

It is possible to completely disassociate implementation and interface. For example, we might implement a set using either an array or a list in such a way that the two kinds of sets can be used interchangeably:

```
class set {
    // ...
};

class v_set : public set,
    private vector {
    // ...
};

class l_set : public set,
    private list {
    // ...
};
```

or graphically:

I use the dotted lines to show that `private` inheritance is an implementation issue that does not affect the interface of the derived class.

Importantly, a common interface (here, `set`) can be provided long after the design and implementation of implementation classes (here, `vector` and `list`). I find that when people design things, they typically first invent something fairly concrete. They design an array, they invent a list, and only later do they discover an abstraction that covers both in a given context. Using abstract classes as shown above, we use (re-use?) `vector` and `list` without the foresight (and cost) necessary to design them as part of a common hierarchy.

As a matter of fact, you can do this ‘‘late abstraction’’ several times. Say, I want to represent the notion of ‘‘something you could read from.’’ This is a very different abstraction from `set`, yet I can provide such an interface to arbitrary sets as well as for lists, vectors, files, and input streams much in the way I provided `set` as an interface to `vector` and `list`.

Late abstraction using abstract classes allows us to provide different implementations of a concept even when there is no significant similarity between the implementations.

6.5 Classical Hierarchies

Sometimes we do have sufficient foresight to design a classical hierarchy. More importantly, sometimes the various implementations of a concept have a high degree of commonality so that there is significant benefit in organizing these implementations into a hierarchy. For example, consider a class hierarchy that one might find in an application relying on a windows system:

Presumably, the implementations of the application classes `ival_dial` and `ival_slider` are greatly facilitated by code and data provided by the ‘‘system classes’’ `window`, `dial`, and `slider`. That is, you build your code incremently and your interfaces incrementally.

6.6 Hierarchies and Abstract Classes

A classical hierarchy is a nice way of providing a variety of related concepts and a nice way of minimizing the effort of building their implementations. However, you do get the classes in the hierarchy tightly coupled. If anything significant in a base class changes, all of the derived classes must change (or at least be recompiled) to match. In particular, any significant change to ''system classes'' at the base of a hierarchy, such as `window`, will affect application classes, such as `ival_dial`. Worse, the choice of a foundation library representing system resources determines the structure of the application class hierarchy and permeates the application code.

There are quite a few ways of dealing with this. For example, some languages have a solution (at its associated costs) mandated, and some implementations of C++ allow major changes to base classes without requiring re-compilation of derived classes. However, here I'll show a solution using abstract classes to make dependencies explicit. Logically, it closely parallels the way the abstract class `set` was used to insulate users from the details of `vector`s and `list`s.

Here, an application hierarchy

is written independently of implementation details and then later tied into an implementation hierarchy without affecting the users of the application hierarchy:

This expresses the design in such a way that the application code becomes independent of any change in the implementation hierarchy.

I have used the `BB` prefix for realism; suppliers of major libraries invariably prepend some identifying initials. In the future, I expect namespaces (§6.3) to be used instead.

The declaration of a class that ties an application class to the implementation hierarchy will look something like this:

```
class BB_ival_slider
    : public ival_slider,
    protected BB_slider {
public:
    // functions overriding ival_slider
    // functions as needed to implement
    // the application concepts
protected:
    // functions overriding BB_slider
    // and BB_window functions as
    // required to conform to user
    // interface standards
private:
    // representation and other
    // implementation details
};
```

This structure assumes that details of what is to be displayed by a windows system is expressed by overriding virtual functions in the `BB_window`s hierarchy. This may not be the ideal organization of a user interface system, but it is not uncommon.

I use protected members and protected inheritance to allow classes derived from `BB_ival_slider` to use information about its implementation.

6.7 Generic Programming

A major theme in the C++ community over the last few years has been the development of techniques exploiting the template mechanism.

6.7.1 Parameterization

Independent concepts should be independently represented and should be combined only when needed.

Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies in the implementation of classes and functions. In particular, fitting weakly related class into a single hierarchy can be a source of unnecessary and problem-causing dependencies.

Consider the concepts of sorting, character, string, and collating sequence. A sort algorithm is independent from the concept of a character. The

concept of a string is independent of any particular kind of a character. Finally, the collating sequence which you use when you sort strings of characters is independent of these other three concepts.

This independence can be expressed directly. Here is a string parameterized by the kind of characters contained so that we can make strings of both built-in and user-defined character types:

```
template<class C>
class string {
    // …
};

class Jchar {
    // Japanese characters
};

string<char> s1, s2;

string<unsigned char> us1, us2;

string<Jchar> js1, js2;
```

Independently, we can define the notion of a collating sequence and a string comparison function:

```
template<class C>
class std_coll {
public:
    bool eq(C a, C b)
        { return a==b; }
    bool lt(C a, C b)
        { return a<b; }
};

template<
    class C,
    class Coll = std_coll<C>
        >
int cmp(
    string<C>& s1,
    string<C>& s2
       )
{
    // compare s1 with s2
    // using Coll::eq and Coll::lt
    // to do character comparisons
}
```

The `cmp` template function takes two template arguments:

- the type of characters in the strings, and
- the collator supplying the character comparison operations.

The ability to pass operations as template parameters is a very powerful expressive mechanism. It is also important for efficiency. For example, it is trivial for a compiler to inline all uses of `eq()` and `lt()`. This can be a significant advantage compared to C where operations can only be passed as pointers to functions so that function call overheads are incurred.

The second template paramenter has a default so we need only specify it if we want a non-standard collating sequence. The first template parameter can usually be deduced from the arguments to `cmp()`. For example:

```
cmp(s1,s2);
cmp(js1,js2);
cmp(us1,us2)
cmp<char,no_case>(s1,s2);
```

Here, `no_case` is a collator defining `eq()` and `lt()` not to be case sensitive.

Typically, the `string` class, the `cmp()` function, the collator classes, and the character classes will be written by different people. Only the final user puts all of the independently developed pieces together.

This style of design relying on templates and additional template arguments (in this example, `Coll`) to express policies, is the basis of much of the C++ standard library. The result is exceptional flexibility and unsurpassed run-time efficiency.

6.7.2 Containers and Algorithms

I want to have algorithms written once and used for objects of many different types. I want these algorithms to run as fast as functions written for a single argument type. I want this to be compile-time checked so that I can be more confident of my code. I don't want to be forced to fit my types into a hierarchy simply to be able to use them for the generic algorithms.

The containers and algorithms in the C++ standard library use a variant of the philosophy of keeping independent concepts separate. Much of the library is based on the notion of a of sequence. Examples of sequences are arrays, sets, lists, maps,

files.

A sequence has a beginning and an end. The end is one beyond the last element of the sequence. Positions in a sequence are represented by iterators.

```
begin                  end
  |                     |
  v                     v
 XXX -> .... -> XXX -> 0
```

Given an iterator for an element of a sequence, we can get to the next element using the `++` (increment) operator. Given an iterator for an element, we can access the element itself using the `*` (dereference) operator.

Given this simple notion, a surprising number of useful algorithms can be expressed. For example, this template function writes all elements of a container to output:

```
template<class C>
void print(C& s)
{
    C::iterator p=s.begin();

    while ( p!=s.end() ) {
        cout << *p;  // output
        p++;         // next
    }
}
```

The C++ standard library containers and algorithms are primarily the work of Alex Stepanov. A surprising number of containers and algorithms can be expressed using just a few kinds of iterators. Importantly, the resulting generic algorithms are efficient even compared to hand-crafted assembly code. For example, the C++ standard library algorithm, `sort()` is for many simple and realistic examples several times faster than the C standard library `qsort()` function. For more information about the C++ standard library and the principles underlying its design see [Koenig,1995] [Stepanov,1994].

7 Closing Remarks

Are the various facilities presented above object-oriented or not? Which ones? Using what definition of object-oriented?

In most contexts, I think these are the wrong questions. What matters is what ideas you can express clearly, how easily you can combine software from different sources, and how efficient and maintainable the resulting programs are. In other words, how you support good programming techniques and good design techniques matters more than labels and buzz words.

The fundamental idea is simply to improve design and programming through abstraction. You want to hide details, you want to exploit any commonality in a system, and you want to make this affordable.

I would like to encourage you not to make object-oriented a meaningless term. The notion of ‘‘object-oriented’’ is too frequently debased

- by equating it with good,
- by equating it with a single language, or
- by accepting everything as object-oriented.

I have argued that there are − and must be − useful techniques beyond object-oriented programming and design. However, to avoid being totally misunderstood, I would like to emphasize that I wouldn’t attempt a serious project using a programming language that didn’t at least support the classical notion of object-oriented programming. In addition to facilities that support object-oriented programming, I want − and C++ provides − features that go beyond those in their support for direct expression of concepts and relationships.

Several of the themes related to C++ programming style in this paper have been developed further in [Koenig,1995b]. The design and evolution of C++, including its most recent features, is discussed in [Stroustrup,1994].

8 Acknowledgements

Thanks to the OOPSLA’95 program committee for inviting me to give the talk upon which this paper is based, and especially to May Loomis for encouraging me to get this paper written. Carolyn Heaps transcribed the audio tape of my talk to produce the first draft of this paper. Tim Griffin and Christos Polyzois made many constructive comments.

9 References

[Koenig,1995] Andrew Koenig (editor): *Draft Proposed International Standard for Information Systems − Programming Language C++*. ANSI Standards Secretariat. CBEMA, 1250 Eye Street NW, Suite 200, Washington DC20005, USA. 1995.

[Koenig,1995b] Andrew Koenig and Bjarne Stroustrup: *A Foundation for Native C++ Styles*. Software − Practice & Experience. To appear 1995.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. ISO Programming language C++ project. Doc No: X3J16/94-0095, WG21/N0482. May 1994.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1991.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.

Supporting SELL for High-Performance Computing

Bjarne Stroustrup and Gabriel Dos Reis

Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Abstract. We briefly introduce the notion of *Semantically Enhanced Library Languages*, SELL, as a practical and economical alternative to special-purpose programming languages for high-performance computing. Then we describe the *Pivot* infrastructure for program analysis and transformation that is our main tool for supporting SELL. Finally, we outline how the IPR (The Pivot's *Internal Program Representation*) can be used to represent central notions of high-performance computing, such as parallelizable array operations. Please note that many of our considerations are practical in nature (relating to software engineering and economics) rather than fundamental from a scientific point of view. Our focus is on a broad exposition of ideas, rather than technical details [1].

1 Languages and libraries

For ease of programming, portability, and acceptable performance, we design and implement special-purpose programming languages for high-performance computing [20]. These languages usually die young. Here, we briefly present a rationale for an alternative approach: *Semantically Enhanced Library Languages*, SELLs. A SELL is a language created by extending a programming language (usually a popular general-purpose programming language) with a library providing the desired added functionality and then using a tool to provide the desired semantic guarantees needed to reach a goal (often a higher level semantics, absence of certain kinds of errors, or library-specific optimizations). This paper focuses on a tool, *The Pivot*, being developed to support SELLs in ISO C++ [15, 7] and its application to High-Performance Computing.

1.1 Reasons for infant mortality

It is fun to design a new programming language. Doing the initial implementation and trying the new language with clever examples can be most exhilarating. However, bringing the implementation up to the level where users who care nothing about language design subtleties is plain hard work. Building supporting tools, such as debuggers and profilers, is hard work and not intellectually stimulating for most people who design programming languages.

[1] This is full version of a short paper presented to LCPC05

Real users also need basic numeric libraries, basic graphical facilities, libraries for interfacing with code written in other languages, "hand holding" tutorials, detailed manuals, etc. Doing each of those things once can be interesting and most educational, doing them all is tedious and often expensive. Porting the implementation, tool base, and key applications to new machines and compilers every half year or so is not only tedious, but also career death for many people. Basically, designing, implementing, maintaining and supporting a language is tremendously expensive. Only a large user community can shoulder the long-term parts of that.

The net effect is that on the order of 200 new languages are developed each year and that about 200 old languages become unsupported each year. "Language death" doesn't just happen to bad languages. For example, you can find a collection of 16 languages for high-performance computing in Wilson and Lu [20]: *Parallel Programming using C++*. Most have very appealing aspects, many are based on brilliant insights, all were supported by an enthusiastic research group, and all had years of stable funding. None are in major use today. None are supported by an organization outside the one that developed them. All but one are dead[2]. Interestingly, the live one (Charm++ [20]) is more of a library than a language.

In addition to the really ambitious language design projects, thousands of researchers work on research dialects and associated tools for their research. Such dialects are not built from scratch; instead, a compiler and key support tools are modified to serve the new dialect. Essentially all become unsupported upon graduation, funding expiration, tenure, promotion, transfer of maintenance responsibilities, change of fashion, change of any part of the tool chain, change of management, consolidation of IT operations, etc.

Some of these languages are designed for research only, but many are aimed at non-research use and most language designers harbor dreams of wide use for their languages. However, most of these new languages and dialects never see non-research use. The ones that do see non-research use are generally unloved by maintenance organizations. That is not just prejudice and unwillingness to learn or to change. There are perfectly good reasons for the lack of enthusiasm in maintenance organizations. For example, the supply of reasonably priced support personnel tends to be severely limited; good designers and good researchers (typically with PhDs) rarely want to become maintainers with an typical maintainer's salary, work conditions, and career prospects. Also, each new language and dialect has its own tool chain that needs to be kept current and in sync with other tools. The cost of doing so for a minor dialect is typically higher than for a major language — because the cost of the latter is amortized over millions of users. These reasons are often solid in economic and management terms, even though they can be heartbreaking for the proponents of a new language or dialect. For example the largest application using ML within AT&T was rewritten in a non-research language and so were the few uses of a very

[2] We'd love to be proven wrong on this; if you have a counter example, please tell us and we'll celebrate this exceptional success together.

interesting rule-based language R++ that can be seen as an early precursor of aspect-oriented programming [9].

1.2 Dialects

A popular “low-cost” approach to providing special-purpose facilities is to add pragmas, annotations, compiler options, “just a couple of extensions”, etc. to a popular language. This creates a dialect of a host language that can serve a specific user-group quite well. However, each such dialect needs its own variant of the compiler and similar tools and tend to be second priority (or worse) to the maintenance organization of the host language. Consequently, such a simple dialect is rarely supported for more than a single compiler or for a single platform. Rarely are such dialects as widely usable as the host language. This isolates the users of the dialect and limits the usefulness of their work.

Worse, two such dialects are rarely composable. A user must choose between “dialect A” and “dialect B”. For example, you might be able to get direct support for parallel programming or for data-base integration, but not both. The fundamental problem is that when two groups each create a dialect, they quickly end up on separate branches of a source code tree or in separate compilers.

Clearly, the ideal for support of special-purpose facilities includes composability with other special-purpose facilities and availability on every platform where the host language is used. It follows that support for special-purpose languages ideally fit within a framework that can accommodate a very general notion of "feature" rather than having the designer of each special-purpose feature provide separate facilities for accommodating syntactic and semantic extensions.

1.3 Language ideals

The prevalence of new languages and dialects in research together with their failure in production use lead to a most unfortunate chasm between researchers and developers. It is not uncommon that neither understand the other’s concerns, ideals, tools, and vocabulary. This can lead to misunderstanding, conflict, and mutual distrust. The difference in the languages used in research and in development is a significant barrier to technology transfer. Thus, language design can do serious harm — even when a design is technically sound and provides a vastly superior solution to a particular problem.

However, let’s be clear: The ideal solution to every major application problem is a special-purpose language designed specifically to meet the ideals of that application. For a given problem, we can always provide a superior solution in the form of a special-purpose language. We can tailor the semantics to the specific needs, we can adjust the syntax to be minimal and/or specialized for the culture of the programmers who will use it. From a narrow point of view, a good general purpose-language is at most second best. However, once we take tool costs, training costs, previous experience, cost of interacting with

other applications and groups, time from start of project to application delivery, etc. into account the balance dramatically shifts in the direction of the general-purpose language and well-known techniques.

So, a special-purpose language is the ideal for the individual application, but a general-purpose language has great advantages for maintenance, training, collaboration between diverse groups, tool chain availability, etc. What are the alternatives to a special-purpose language? Which tools and techniques can provide many of the advantages without suffering the problems? Many programmers and researchers still dream of a general-purpose language that is so expressive, elegant, and efficient that it can cover all application areas for all kinds of users without significant inconveniences. Some vigorously insists that their favorite language actually is such a language. However, no current language in major use fits this ambitious profile, and we don't see a research language that could possibly grow into that position in the short-to-medium term (e.g., within the next 15 years). Consequently, we suggest an alternative approach: semantically enhanced libraries embedded in a widely-used general-purpose language.

The rest of this paper describes the tools built to support this approach for C++ libraries.

2 A brief overview of the Pivot

The Pivot is a general framework for the analysis and transformation of C++ programs. The Pivot is designed to handle the complete ISO C++, especially more advanced uses of templates and including some proposed C++0x features. It is compiler independent. The central part of the Pivot is a fully typed abstract syntax tree called IPR (*Internal Program Representation*).

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g. [11, 3, 12, 10]. However, few — if any — handle all of ISO Standard C++ [7, 15], most are specialized to particular forms of analysis or transformation, and few will work well in combination with other tools. We are particularly interested in advanced uses of templates as used in generic programming, template meta-programming, and experimental uses of libraries as the basis of language extension. For that, we need a representation that deals with types as first-class citizens and allows analysis and transformation based on their properties. In the C++ community, this is discussed under the heading of *concepts* and is likely to receive significant language support in the next ISO C++ standard (C++0x) [17, 18, 16, 13, 19]. You can think of a concept as the type of a type. From the point of view of support for HPC — and for the provision of special-purpose facilities in general — a concept can be seen as a way of specifying new types with associated semantics without the modification of compilers or new syntax. That done, the SELL approach then uses the concepts as a hook for semantic properties beyond what C++ offers.

This paper is an overview and will not go into details of concepts, the Pivot, the IPR, or our initial uses.

2.1 System organization

To get IPR from a program, we need a compiler — only a compiler "knows" enough about a C++ program to represent it completely with syntactic and type information in a useful form. In particular, a simple parser doesn't understand types well enough to do a credible general job. We interface to a compiler in some appropriate (to a specific compiler) and minimally invasive fashion. A compiler-specific IPR generator produces IPR on a per-translation-unit basis. Applications interface to "code" through the IPR interface. So as not to run the compiler all the time and to be able to store and merge translation units without compiler intervention, we can produce a persistent form of IPR called XPR (*eXternal Program Representation*).

Fig. 1. An overview of *The Pivot* infrastructure

Only a complete and correct C++ compiler can collect sufficient information for type-driven or concept-driven applications. From a compiler, we generate IPR containing fully typed abstract syntax trees. In particular, every use of a function name and operator is resolved to its proper declaration, all scope resolution is done, and all implicit calls of constructors and destructors are known.

We have IPR generators from GCC and EDG, so that the Pivot is not compiler specific. Early versions of this system (and its precursors) have been used to write pretty printers, generate XML for C++ source, CORBA IDL from C++ classes, and distributed programs using C++ source augmented with a library defining modularity. We do not assume the ability to produce information suitable for feeding IPR directly back into the internal formats of a compiler. Such

a facility may exist for a give compiler, but to preserve the Pivot's independence of individual compilers such interfaces are not a priority. The reason for preserving compiler independence is to maximize the portability of IPR-based tools. A tool that is built directly on a compiler's internal interface cannot easily be ported to another compiler. In fact, the interfaces to current C++ compilers' data structures for syntax and type information differ dramatically and most are de facto inaccessible for technical, commercial, or political reasons.

2.2 IPR principles

The IPR (*Internal Program Representation*) is a typed abstract syntax tree representation that retains all information from the C++ source (with exception of macro definitions). The IPR is compact, completely typed (every entity has a type, even types), representation with an interface consisting of abstract classes. The IPR has a unified representation so that its memory consumption is minimal. For example there will be only on node representing the type `int` and only one node representing the integer value `42` in a program that uses those two entities. This minimalism (in time and space) of the IPR is key to its use for large systems — million line programs are no longer rare.

The IPR does its own memory management so users do not have to keep track of created objects. The IPR is arguably optimal in the number of indirections needed to access a given piece of information. The IPR is minimal in that it holds only information directly present in the C++ source. IPR can be annotated by the user and flow graphs can be generated. However, that's considered jobs for IPR applications rather than something belonging to the core framework itself. In particular, traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code", visitors [5], iterators [14, 2, 15], or tools such as Rose [11]. The needs of the application — rather than the IPR — determines what traversal method is most suitable.

The IPR library has two parts:

1. an immutable interface suitable and simplified for the large number of application that do not need to modify input source
2. a slightly more complicated interface for users that need to mutate the representation of an IPR module.

The IPR can represent both correct and incorrect (incomplete) C++ code and both individual translation units and merged units (such as a complete program). It is therefore suitable for both analysis of individual separately-compiled units and whole-program analysis.

The IPR represents ISO C++ code. That implies that it can trivially be extended to represent C [8, 6] code and easily extended to common C++ dialects. However, since the initial aim of the Pivot is to look into high-level type-based and concept-based transformation, there is no immediate desire to extend it to cope with other languages with significantly different semantics, such as Fortran or Java.

Consider the declaration:

```
int foo(int a = 5, float b = 2.4)
```

The parameter `b` has type `float` and default argument `2.4` (a floating-point literal of type `double`). Its IPR representation is displayed in Fig. 2.

Fig. 2. IPR for the parameter declaration `float b = 2.4`

Here, the named "fields" of the representation indicate member functions that supply the information; however stored. The arrows represent the values returned by such function for this example.

In IPR, a declaration is represented by four essential and two optional components:

1. a name,
2. a scope (in which the name is declared)
3. a type, and
4. a set of specifiers.
5. an optional initializer
6. an optional set of annotations

The initializer for a parameter is its default argument value. The name of the declaration, represented as an `Identifier`, is an expression. That type is represented as `decltype(b)`, meaning *the declared type of* `b`. That instance of `Decltype` in turn has a type (the *concept* of that type), not shown on the picture. The type (`float`) of the parameter declaration is a built-in type. Its representation is the identifier `float` interpreted as a type; basically `As_type(float)`. The default argument value, `2.4`, is a literal of type `double`. To maintain independence of any particular implementation, a literal is represented as a (string,

type) pair. A declaration knows its enclosing declarative region, and the owner of that region (membership). An annotation is a (name,value) pair optionally attached to an IPR node by a Pivot application for its own uses. An annotation does not affect the way the IPR functions.

User programs can annotate IPR nodes. The IPR "remembers" the C++ source locations of its nodes, so that a tool can refer back to the original source code.

2.3 XPR principles

The XPR (*eXternal Program Representation*) is a compact (about the size of the original C++ source) and human readable ASCII representation of IPR. It can be thought of as a persistent form of IPR or even as an object database for IPR. XPR can be used as a transfer format between two different runs of the Pivot (saving us the bother of restarting a compiler) or two different implementations of the IPR. The XPR is designed to be parsed without a symbol table and with only a single token look ahead (in other words: fast). It is strictly prefix notation where types are concerned. Expressions and statements are rather C++ like.

Here is an XPR example:

```
D : <T : typename>
   class : (public B1, public B2) {
      int16 : private typedef short;
      count : private int16;
      size : private int;
      element : T;
      #ctor : public (i1 : int16, i2 : int)
              count(0), size(1) { };
      foo : public (a : int <-5, b : float <-2.4) int
      {
         a = b * a + element;
         return a;
      };
};
```

Obviously we use the *name : definition* notation. The details are "C++ like" while avoiding grammar ambiguities; for example, `< ... >` defines template parameters, `( ... )` function parameters, and `<-` means initialize. The corresponding C++ is:

```
template<class T>
class D : public B1, public B2 {
   private:
   typedef short int16;
   int16 count; int size;
   T element;
public:
   D(int16 c, int s) : count(c), size(s) { }
   int foo(int a = 5, float b = 2.4)
   {
```

```
        a = b * a + element;
        return a;
    }
};
```

Bindings to declarations and types are represented as annotations and kept near to, but separate from the human readable part.

2.4 Pivot Summary

The Pivot is a framework for analysis and transformation of ISO C++ programs with an emphasis on the higher levels of the C++ type system, such as templates, overloading, specialization, and concepts (proposed for C++0x). It is very general and can be used to interface arbitrary C++ analysis and transformation tools to a compiler (as long as the tool can manage without macro definitions). The Pivot is compiler neutral and minimally invasive into a tool chain.

The library implementing the IPR is elegant, compact, and efficient. It is just 2,500 lines of C++ to cope with all of C++, unify types (and literals and anything else we might want to unify), and manage memory.

3 High-level program representation for HPC

Type systems have been introduced in programming primarily for correctness and efficiency. For example, if we know at translation time that an operation involving read and write accesses is alias free, we can exploit that for generating efficient code. Some programming languages, notably FORTRAN, are designed to allow the compiler to assume the absence of aliases. Other general-purpose programming languages, such as C C++, allow only a restricted set of type-based aliasing. For example a pointer of type `void*` can be used to access any kind to data, but a pointer of type `int*` cannot be effectively used to access data of type `double`.

A programming discipline that effectively uses the type system can help make programs both correct and efficient. Abstract representation of programs naturally enables symbolic manipulation. Here, we present an approach to correctness and performance using a high-level, fully typed abstract representation of ISO C++ programs developed as part of *The Pivot* framework. We will use the notion of parallelizable vector operation as a running example.

Why C++? For the SELL approach we need a widely-used general-purpose language for our “host language”. For type transformation and high-level work, we need a language that provides a flexible type system that can be used in a type-safe manner. For high-performance computing, we need a language that can efficiently use hardware resources and is available on high-end computers. For wide use, we need a non-proprietary and operating system neutral language.

3.1 A notion of parallelizable

Consider the classic operation

```
z = a * x + y;
```

where a is scalar, x, y and z denotes vectors, and the operations * and + are component-wise. It can be parallelized if we know that the destination z does not overlap with the sources x and y in a way that displays non-trivial data dependencies. That happens, for example, if we know no vector element has its address taken. For exposition purpose, we will simplify the notion of Parallelizable to a collection of types whose objects support the operation [] (subscription) but not & (address-of) on its elements. For instance, in the generic function

```
template<Parallelizable T>
void f(const T& v)
{
   double a = v[2];    // #1: OK
   double* p = &v[2];  // #2: NOT OK.
}
```

line #1 is valid but line #2 is an error because it uses a forbidden operation. We generalized the standard notation template<typename T> which reads "for all T", to template<Parallelizable T> meaning "for all T such that T is is Parallelizable".

We use the word *concept* to a designate collection of properties that describes usage of values and types. Basically, a concept is the type of a type. There are proposals to integrate concepts into the C++ programming languages [17, 18, 16, 13, 19]. However, using IPR we can handle concepts without waiting for the C++ standards committee to decide on the technical details 3.3.

A programmer might use Parallelizable to constrain the use of a vector:

```
vector<double> v(10000);
// ...
f(v);  // f will use v as an Parallelizable (only)
```

Here we now know that f() will not use & on v even though the standard library vector actually allows that operation. We can use f() with its no-alias guarantee for any type that supports subscripting. For example we might use a STAPL [1] pvector:

```
pvector<double> vd(100000);
// ...
f(vd);
```

The concept checking allows no assumptions about types uses beyond what the concept actually specifies (here, a Paralleizable provides []). In particular, no hierarchical ordering or run-time mechanisms are required.

Note that when defined in this way, `Parallelizable` requires no modification to C++0x or to any compiler. Furthermore, the use of `Parallelizable` is most likely to be composable with other facilities introduced as concepts – even if the facilities were developed in isolation.

Below, we will briefly present a high-level representation of C++ programs that support concept-based analysis and transformations.

3.2 Concepts in the IPR

A translation unit is represented as a graph with a distinguished root for the sequence of top-level declarations. In IPR, every entity in a C++ program is viewed as an expression possessing some type. So, types have types, which are called concepts. This becomes more useful, and maybe clearer, for a type variable as we find them in template parameter lists.

Fig. 3. IPR model `Parallelizable T`

In Fig. 3, we have drawn a view of the representation of the declaration `Parallelizable T`. The declaration of the template-parameter `T` has type `Parallelizable`. If we knew about the syntax and semantics of `Parallelizable`, that knowledge would be represented by a node referred to by the `type` field of the node with the identifier `"Parallelizable"`.

Note how `Parallelizable` fits into the IPR framework without modification or special rules. `Parallelizable` is simply a (deliberately trivial) example of what can be done with concepts in general.

Concepts are the basis for checking usage of types in templates, just like ordinary types serve to check uses of values in functions. Concept checking is done at two sites: (a) at template use site; and (b) at template definition site. If concept checking succeeds at both sites, then it is guaranteed that template arguments are used (only) according to the semantics expressed in the concepts [4]. In the particular case of `Parallelizable`, it means that no vector has its address taken, and consequently parallelization transformations can be safely applied.

3.3 Getting concepts into the IPR

How do we get concepts into our program? C++0x will most likely provide a way of specifying and checking concepts. That will provide a convenient handle for all concepts and for all SELL type-based analysis and transformation. For example:

```
concept Parallelizable<typename T> {
    // operations required by any Parallelizable type
    // only required operations will be accepted
    // for an object of a Parallelizable type
};

template<Parallelizable T>
void f(const T& v)
{
    // operate on v according to the rules
    // specified for Parallelizable
}

vector<double> v(10000);
// ...
f(v);  // f will use the vector as an Parallelizable (only)
```

Once, the `Parallelizable` concept is part of the program, a Pivot application can operate based on its understanding of it. Note that this "understaning" can be extra-linguistic based on the tool builders knowledge of the semantics of the library of which `Parallelizable` is part.

However, what do we do if we don't have a C++0x compiler that directly supports concepts? After all, C++0x won't be fully specified for another couple of years. We could rely on annotations, pragmas, language extensions, etc., but as noted in 1.2, that has serious implications and costs. In particular, our programs would almost certainly not be composable with extensions defined and implemented by another group. The obvious alternative is to rely on convention: Traditionally, C++ programmers name template parameters to indicate their intended use. For example:

```
template<class Parallelizable>
void f(const Parallelizable& v)
{
    // operate on v according to the rules of Parallelizable
}
```

A Pivot application (tool) can easily recognize the type name Parallelizable and connect it to the definition of the concept Parallelizable as defined by the tool. From the point of view SELL and the Pivot, C++0x concepts is a significant convenience that provides a major advantage in notation and checking. However, it is only a (major) convenience because a Pivot-based tool can manipulate the IPR directly. For example, we could take code using the C++ standard library `accumulate`

```
template<class InputIterator, class T>
T accumulate(InputIterator first,
             InputIterator last, const T& initalizer);
```

and transform every use into its equivalent parallel STAPL p-algorithm if (and only if) the STAPL requirements for its arguments are met. That is, the transformation takes place iff in addition to being an `InputIterator` the argument tyoe is a `BidirectionalIterator` or a `RandomAccessIterator`. This general approach to semantics-based transformation applies to all C++ standard algorithms described in terms of "abstract sequences".

The concept-based techniques rely critically on the use of templates, so that we can type template paraments with concepts to get a handle on their semantic properties. So, what do we do with code that doesn't use templates? Given an abstract syntax tree that represents a function declaration, we can transform it into a templated version and concept-check it. Consequently, we can check and transform a whole program as if it was fully templated.

4 Conclusion

We presented a rationale for the SELL, *Semantically Enhanced Library Language*, approach to supporting special-purpose languages, arguing that this approach has practical and economic advantages compared to conventional approaches to providing special-purpose facilities. In particular, SELL can yield extension that are composable and portable. We presented our main tool for supporting the "semantic part" of that approach, *The Pivot*. The Pivot provides a general framework for analysis and transformation of C++ programs with an emphasis on high-level and type sensitive approaches. Our semantics-based analysis and transformation do not require modification to a host language and is minimally invasive to tool chains. It relies on a high-level program representation, the IPR, with emphasis on types and concepts. Using the IPR we can perform analysis and transformation for high-performance computing (as well as other forms of computing) that traditionally required special-purpose languages or ownership of a specialized compiler and related tool chain.

References

1. Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *Proceeedings of the International Workshop on Languages and Compilers for Parallel Computation (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, August 2001.
2. Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, October 1998.
3. O. Bagge, K. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis*

and Manipulation (SCAM 2003), pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

4. Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. 2005. Accepted for publication at POPL06.
5. Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
6. International Organization for Standards. *International Standard ISO/IEC 9899. Programming Languages — C*, 1999.
7. International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.
8. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
9. Diane Litman, Peter F. Patel-Schneider, and Anil Mishra. Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules. In *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*. ACM Press, October 1997. `http://www.research.att.com/sw/tools/r++/`.
10. Georges C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Tranformations of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 2002. `http://manju.cs.berkeley.edu/cil/`.
11. M. Schordan and D. Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In *Proceeding of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer-Verlag, 2003.
12. S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.
13. Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concept for C++0x. Technical Report N1758=05-0018, ISO / IEC SC22 / JTC1 / WG21, January 2005.
14. Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO / IEC SC22 / JTC1 / WG21, May 1994.
15. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.
16. Bjarne Stroustrup. Concept checking — A more abstract complement to type checking. Technical Report N1510, ISO / IEC SC22 / JTC1 / WG21, September 2003.
17. Bjarne Stroustrup and Gabriel Dos Reis. Concepts — Design choices for template argument checking. Technical Report N1522, ISO / IEC SC22 / JTC1 / WG21, September 2003.
18. Bjarne Stroustrup and Gabriel Dos Reis. Concepts — syntax and composition. Technical Report N1536, ISO / IEC SC22 / JTC1 / WG21, September 2003.
19. Bjarne Stroustrup and Gabriel Dos Reis. A Concept Design (rev.1). Technical Report N1782=05-0042, ISO / IEC SC22 / JTC1 / WG21, April 2005.
20. Gregory V. Wilson and Paul Lu, editors. *Parallel Programming using C++*. Scientific and Engineering Computation. MIT Press, 1996.

A rationale for semantically enhanced library languages

Bjarne Stroustrup
Department of Computer Science
Texas A&M University
College station, TX-77843
and AT&T Labs – Research
bs@cs.tamu.edu

Abstract

This paper presents the rationale for a novel approach to providing expressive, teachable, maintainable, and cost-effective special-purpose languages: A *Semantically Enhanced Library Language* (a *SEL language* or a *SELL*) is a dialect created by supersetting a language using a library and then subsetting the result using a tool that "understands" the syntax and semantics of both the underlying language and the library. The resulting language can be about as expressive as a special-purpose language and provide as good semantic guarantees as a special-purpose language. However, a SELL can rely on the tool chain and user community of a major general-purpose programming language. The examples of SELLs presented here (*Safe C++*, *Parallel C++*, and *Real-time C++*) are based on C++ and the Pivot program analysis and transformation infrastructure. As part of the rationale, the paper discusses practical problems with various popular approaches to providing special-purpose features, such as compiler options and preprocessors.

1 Introduction

We often need specialized languages. Researchers need to experiment with new language features, such as concurrency features [24], facilities for integration with databases [5], and graphics [4] . Developers can sometimes gain a couple of orders of magnitude reductions in source code size with corresponding reductions in development time and defect rates, by using such special-purpose languages in their intended domains. Unfortunately, such special-purpose languages are typically hard to design, tedious to implement, expensive to maintain, and — despite their obvious utility — tend to die young.

Using a (special-purpose) library is an obvious alternative to a special-purpose language. However, a library cannot express or exploit semantic guarantees beyond what its host language provides. The basic idea of *Semantically Enhanced Library Languages* (SEL Languages or simply SELLs) is that when augmented by a library, a general-purpose language can be about as expressive as a special-purpose language and by subsetting that extended language, a tool can provide about as good semantic guarantees. Such guarantees can be used to provide better code, better representations, and more sophisticated transformations than would be possible for the full base language. For example, we can provide support for parallel operations on containers as a library. We can then analyze the program to ensure that no undesirable access to elements of those containers occurs — a task that could be simplified by enforcing a ban of languages features that happened to be undesirable in this context. Finally we can perform high-level transformations (such as parallelizing) by taking advantage of the known semantics of the libraries.

Like a library, a SELL can benefit from the extensive educational, tools, and library infrastructure of the base language. Therefore, the cost of designing, implementing, and using a SELL is minuscule compared with a special-purpose language with a small user base. Examples will be based on ISO standard C++ supported by the Pivot infrastructure for program analysis and transformation (§5.2). The focus will be on templates because they provide the key mechanism for statically type-safe expression of advanced ideas in C++.

What is called a "special-purpose language" here is often called a domain-specific language (e.g. [10]). Distinctions can be made between the two terms, but none that appear relevant to the discussion here, so please consider those two terms as equivalent in this context.

The organization of this paper is

1. Introduction
2. State some ideals for support of software development and maintenance.
3. Present some of the — usually fatal — problems that face new programming languages.
4. Discuss a few alternative approaches, such as dialects and macro languages.
5. Focus on the SELL approach and the way it can be supported in C++ using the Pivot.
6. Sketch the design of a few SELLs: type-safe C++, Parallel C++, and Real-time C++.
7. Conclusions

Submitted to LCSD'05

2 Ideals

For every specific problem area, we can design a special-purpose language that exactly matches the desired syntax and semantics of the domain and the desires of the programmers that will use that language. In an ideal world, no general- purpose language can match such a special-purpose language when applied in its specific problem area. When a special-purpose language has been done perfectly, there is a one-to-one correspondence between the fundamental concepts of the application domain and the language constructs. Given that, the language constructs can be minimal and directly reflect the terminology of the field as found in common use and major textbooks.

This is not a new ideal. Fortran did a good job at that task for arithmetic in the 1950s and COBOL successfully attacked the business processing needs of the time. Since then, thousands of languages have been designed for specific domains and almost as many have been designed to try to be able to effectively express that ideal for less specific domain. Lisp and Simula originate the two main approaches to more directly express application domain concepts directly in code: the functional and object-oriented approaches. In these languages, and in their numerous offspring, a set of concepts is represented as a library of related functions or classes. In such general-purpose and near-general-purpose languages the ideal of the perfect language for the task takes the form of libraries.

What do we expect from a well-designed special-purpose language? Concise notation is the beginning. Consider a simple, common, and useful example:

```
A = k*B + C
```

First note the algebraic notation using operators. Notation is important for concise expression of key ideas in a community. This particular notation is based on almost 400 years of history in the mathematics / scientific community.

Essentially all languages can handle `A=k*B+C` when the variables denote scalar values, such as integers and floating point numbers. For vectors and matrices, things get more difficult for a general-purpose language (that doesn't have built-in vector and matrix types) because people who write that kind of code expect performance that can be achieved only if we do not introduce temporary variables for `k*B` and `k*B+C`. We probably also need loop fusion (that is doing the element * , +, and = operations in a minimal number of loops). When the matrices and vectors are sparse or we want to take advantage of known properties of the vectors (e.g. `B` is upper-triangular), the library code needed to make such code work pushes modern general purpose language to their limit [17, 23] or beyond — most mainstream languages can't efficiently handle that last example. Move further and require the computation of `A=k*B+C` for large vectors and matrices to be computed in parallel on hundreds of processors. Now, even an advanced library requires the support of a non-trivial run-time support system [1]. We can go further still and take advantage of semantic properties of operations, such as "remembering" that C was the result of an operation that leaves all its elements identical. Then, we can use much simpler add operation that doesn't involve reading all the elements of `C`. For other examples, preceding the numerical calculation with a symbolic evaluation phase, say doing a symbolic differentiation, can lead to immense improvements in accuracy and performance. Here, we leave the domain where libraries have been considered useful. Reasoning like that and examples like that (and many more realistic ones) have led to the creation of a host of special-purpose languages for various forms of scientific calculation [24].

So, the ideal notation offered by a general purpose language is just the beginning. It can be the basis for comprehension, for fast compilation, for performance (exploiting type information and semantic properties), for reasoning about programs (by the implementation or associated tools), for programmer productivity, for making facilities accessible to professionals who need to program in their field of expertise, yet don't want to become professional programmers (e.g., physicists, engineers, animators, and graphical designers). Finally, the clarity of the code can greatly ease maintenance.

Note that the ideals and strengths of special-purpose and general-purpose languages can conflict. By definition, a general-purpose language aims at allowing the programmer to express just about anything. On the other hand, a special-purpose language gains much of its strength from allowing a programmer to express only what makes sense in its specific domain. When it comes to program analysis and optimization, this is a great strength of a special- purpose language. For example, if an optimizer tries to do a symbolic differentiation of a program in a language focused exclusively on scientific computation, it does not have to worry about a programmer trying to differentiate the draw function of a graphics system.

Convenient graphical interfaces are often associated with special-purpose languages. They can be used as an extreme example of direct representation of ideas or as a special- purpose language. However, such interfaces can be used to equal effect for code in a general-purpose language, so GUIs will not be examined further here.

3 Problems

It is fun to design a new programming language. Doing the initial implementation and trying the new language with clever examples can be most exhilarating. However, it is plain hard work to bring the implementation up to the level needed for users who care nothing about language design subtleties. Building supporting tools, such as debuggers and profilers, is hard work and not intellectually stimulating for most people who design programming languages. Real users also need basic numeric libraries, basic graphical facilities, libraries for interfacing with code written in other languages, "hand holding" tutorials, detailed manuals, etc. Doing each of those things once can be interesting and most educational, doing them all or repeatedly is tedious and often expensive. Porting the implementation, tool base, and key applications to new machines, platforms, and compilers repeatedly is not only tedious, but also career death for many people. Basically, designing, implementing, maintaining and supporting a language is tremendously expensive. Only a large user community can shoulder the long-term parts of that.

The net effect is that on the order of 200 new languages are developed each year and that about 200 languages become

unsupported each year. "Language death" doesn't just happen to bad languages. For example, you can find a collection of 16 languages for high-performance computing in *Parallel programming using C++* [24]. Most have very appealing aspects, many are based on brilliant insights, all were supported by an enthusiastic research group, and all had years of stable funding. None are in major use today. None are supported by an organization outside the one that developed them. All but one are dead[1]. Interestingly, the one survivor (Charm++) is more of a library than a language.

In addition to the really ambitious language design projects, thousands of researchers work on dialects and associated tools for their research. Such dialects are not built from scratch; instead, a compiler and key support tools are modified to serve the new dialect. Essentially all become unsupported upon graduation, funding expiration, tenure, promotion, transfer of maintenance responsibilities, change of fashion, change of any part of the tool chain, change of management, consolidation of IT operations, etc.

Some of these languages are designed for research only (or claim to be), but many are aimed at non-research use (or claim to be) and most language designers harbor dreams of wide use for their languages. However, most of these new languages and dialects never see non-research use. The ones that do, are generally unloved by maintenance organizations. That is not just prejudice and unwillingness to learn or to change. There are perfectly good reasons for the lack of enthusiasm in maintenance organizations. For example, the supply of reasonably priced support personnel tends to be severely limited. Good designers and good researchers (typically with PhDs) rarely want to become maintainers with a typical maintainer's salary, work conditions, and career prospects.

Each new language and dialect has its own tool chain that needs to be kept current and in sync with other tools. The cost of doing so for a minor dialect is typically higher than for a major language — because the cost of the latter is amortized over millions of users. These reasons are often solid in economic and management terms, even though they can be heartbreaking for the proponents of a new language or dialect. For example, the largest application using ML within AT&T was rewritten in a non-research language and so was the few uses of a very interesting rule-based language R++ that can be seen as an early precursor of aspect-oriented programming [11].

Tool chain problems don't just happen to "Mom&Pop languages". I have seen major organizations abandon Ada for just this reason. Similarly, education can be a major problem. If a language isn't taught in universities (or only in a few schools), good programmers become scarce and most organizations cannot afford to re-train new hires. Furthermore, new programmers are sometimes overly impressed by their favorite language and resists training. I have seen organizations abandon Fortran for that reason. The two effects are mutually reinforcing.

However, most special-purpose languages, proprietary dialects, etc. never get a large enough user base and tool set to worry about decline. Most minor and research languages simply never gain the tool support and availability on a wide range of platforms that users of mainstream languages take for granted. Unless a new language is really a minor dialect of an existing language, almost all of the design and implementation effort is recreating facilities — such as debuggers, profilers, database interfaces, and GUI interfaces — that tend to lie outside the main interest of the language designers. This repettetive reconstruction of "standard facilities" provided for other languages breeds lots of "good little ideas" as people add improvements. Unfortunately, such "little improvements" tend to further isolate users. Since "further isolates" can be read as "locks-in users" as well as "provides better support than the competition", there is often little resistence to gratuitous replication and incompatibility. Compatibility is just hard work, and typically unrewarding.

How many users does it take to sustain an infrastructure? Of course, that depends on a lot of things, but generally it requires more people than work on a single application. In fact, it typically takes at least a small company. That is more — often significantly more — people than it took to create the initial design and implementation of a language. If — as is usual — these people have to be paid from the revenues from sales and teaching, a special-purpose language now comes under pressure to become more widely useful. That is, the special-purpose language starts to offer facilities for general computation, general data structures, access to "external systems", database facilities, graphics facilities, etc. The result can be summarized as "Every special-purpose programming language wants to grow up and become a general-purpose programming language." Typically, this is a precursor to "language death" (becase of instability, lack of design focus, and added cost) or to a retreat into a commercially viable niche that covers only a small part of the special-purpose language's natural application domain. This withdrawl is often accompanied with a lot of commercial hype and a tendency to hide and obscure genuine technical information.

Many (probably most) special-purpose languages suffer from "edge effect" problems. The "edge effect" (also more evocatively known as the "falling off the cliff" effect) comes when a programmer needs to do something that isn't supported by the special-purpose language. For example, a programmer using a language for specifying interactive graphics might want to say "when viewed from a sufficient distance, groups of objects may be considered one object". The graphics system could have provided such a feature, but in this case it didn't (and the difference in real-time response was about a factor of 100). What does the programmer do? By definition, every special-purpose language has such "edges". For students and novices, the effect can be a nuisance; for professionals working on large projects (such as the airline control application from which this graphics example was chosen), the result can be the abandonment of the special-purpose language in favor of an alternative, such as a graphics library written in a general-purpose language. But what does a programmer do if changing tools isn't an option? In a "pure" special-purpose language, a new primitive operation or object must be added. That's not something every application programmer can do because it may effect the basic model of the special-purpose language. I have seen the time for adding a simple feature vary from one day (ask a local expert and wait for the overnight tool build) to half a year (wait for the

[1] I'd love to be proven wrong on this, so if you have a counter example, please tell me and we'll celebrate this exceptional success together.

next release) or more. This kind of delay can kill a project, so it must be considered among the risks when choosing or designing tools. For a library — and for any tool that allows a programmer to add code written in a general-purpose language — the problem is minor.

The final nail in the coffin of many special-purpose languages is that once it is designed and in use, it is relatively easy to "emulate" its facilities in a general-purpose language. Often, the value of a special-purpose language is not really in the language implementation or its particular syntax (though programmers can be passionately devoted to a syntax). The value is in the design, the programming model, the techniques for use, and possibly some special algoritms or data structures sustaining applications. Typically, those special-purpose language "implementation details" can be separated from the language and used directly from a general-purpose language. This is all the easier because these key components are written in some general-purpose language. All that is needed for their direct use is a nice programming interface in that general-purpose language. The definition of "nice" will reflects the experience gained from the use of the special purpose language.

Please note that a language is rarely "killed" by any one of the problems mentioned. Typically, the language succumb to a combination of problems. Also, this list is not intended to be complete or necessary "fatal": some special-purpose languages do survive and some fail because of reasons not listed here. An exhaustive list of problems probably couldn't be compiled, and if it could it would be beyond the scope of this paper.

3.1 Case study: R++

A detailed study of a few hundred new languages to provide solid evidence for the observations made here would be useful. However, I doubt it would dampen the enthusiasm for designing new languages. Here, I'll just present one small example, and then proceed to an alternative approach to providing new facilities for programmers.

R++[11] is an unrecognized precursor to aspect-oriented programming. Basically, it is an extension of C++ in which you can define actions and triggers for actions. For example, a retirement policy can be associated with an `Employee` class like this:

```
rule Employee::retirement_policy {
     age>=65 && status!=retired
=>
     cout << name << " must retire...";
}
```

This is simple enough to be easy to teach. Furthermore, the implementation was a small enough increment on C++ that it was relatively easy to maintain. Since R++ is a superset of the general-purpose language C++ there are no edge effects. It was used in a reasonably large telecom operations system application. Tutorials, academic papers, manuals, experience reports, implementation, etc. were provided. You can find them on the web [11].

For all practical purposes, it died in 1996. The reasons were basically that the porting and training costs were too high compared to the benefits. What do I mean by dead? Completely unused? Not necessarily. Ever so often, I see a reference to R++ and I'd be surprised if there wasn't a project somewhere using it. Probably, there are also a couple of research groups trying it out. However, despite ideas that appear fundamentally sound, despite avoiding edge effects by being embedded in a general-purpose language, and despite having an implementation that did sustain a major application, R++ still suffered many of the various problems mentioned in this section and failed to gain major use outside its originating organization.

4 Alternatives

So, in most cases, designing a new language is not an economically viable solution to the problem of how to provide special-purpose facilities. A language often looks good for a few years but maintenance, porting, education, etc. is too expensive and the result is death or at best stagnation of the tool chain and the user community. As a technical/economical choice, designing a new language most often is a mistake. Most language design efforts soak up resources reinventing a few wheels and then die having provided a poor return on investment. The resources could have been better spent on improvements to an existing major language and its libraries and tools. Furthermore, most new languages divide a community by creating barriers to communication of new ideas and not infrequently by generating hype that trigger language wars and distrust of new ideas. Not all of the problems are the fault of the new language and new ideas must be explored and exploited. So, what else can we do to bring the ideal of direct expression of ideas in code into wider use?

So, let's assume that we are in one of the many situations where designing a new language is likely to be uneconomical and to have undesirable effects on the spread of ideas. What alternatives do we have when our task is to provide programmers with improved tools for expression ideas in code?

Here are some popular approaches:

1. Compiler options and pragmas
2. Libraries
3. Preprocessed languages
4. Dialects

Each can be an effective approach in some cases and each has been used in ways that have been deemed successful. Here, we must consider their fundamental and practical strengths and weaknesses.

These are not the only possible approaches. For example, one might consider:

1. Dynamically typed languages
2. A new, more general, general-purpose language

Dynamically typed languages are not considered here. The the main reason is an interest in compile-time guarantees. Basically, dynamically -typed languages constitutes a different world from the statically typed world that I focus on here. Dealing with that world is beyond the scope of this paper.

One might consider building a new general purpose language providing facilities that are so complete that every special-purpose language can be expressed directly through the mechanism of the general purpose language. That's one of the holy grails of general-purpose language design. In fact, over the last 30 years or so, there has been a stream of such langugages offering facilities for defining extended syntax (e.g. through embedded parsers) and associating semantics with the newly defined constructs. Such languages are also beyond the scope of this paper. Part of the reason is that providing such a language is beyond the means of most organizations needing a special-purpose language. Another problem is that (ironically) such languages themselves suffer from the problems of being special-purpose languages with small user communities and insufficient support. The success rate for general-purpose languages is even lower than the rate for special-purpose languages.

4.1 Compiler options and pragmas

People who add compiler options and/or pragmas rarely think that as language design. In particular, (in the C and C++ worlds) a `#pragma` can be ignored by a compiler. However, every new `#pragma` and compiler option introduces a new dialect. It is something to consider when building a system, when specifying a system configuration, when porting a system, when documenting a system, and when trying to understand application code. Assume for a moment that options and `#pragma`s are not used for back-door language extension. Then, they are simply insufficient for doing anything really interesting in the direction of better expression of ideas. Most special-purpose languages require additions. Also, they often require restriction of use of certain undesirable language features. That makes compiler options a too crude a mechanism. Options tend to apply indiscriminately; for example, we might want to eliminate the use of `goto`. However, the option will then eliminate all `goto`s — even the acceptable ones for breaking out of loops in a highly optimized matrix implementation and the essential ones in implementation of the state machines generated from a high-level modeling library/language. What is needed is to distinguish between uses of an undesirable language feature in user code and their use in the implementation of trusted components. Compiler options are best left for conventional uses, such as backwards compatibility switches; `#pragma`s are best avoided.

4.2 Libraries

Libraries can provide expressive power and notational convenience that approximate that of built-in language features. However, it is hard to ensure consistent use of a library (or a set of libraries). It is even harder to ensure consistent use of a subset of a library when — as is common — too much has been bundled into a single unit of distribution. Other language features can interfere with what a library attempts to achieve. The C++ standard library is a classical example. It provides well-behaved containers, but some programmers use arrays instead and thereby prevents any meaningful guarantees to be made for the program as a whole.

When ambitious in what they try to achieve in terms of generality or performance, libraries can become very elaborate and brittle. For example, some C++ template metaprogramming libraries aiming at very general support for high-performance numerical computation reach their goal at the cost of complete obscurity of implementation details that becomes visible to users during debugging. Often, a library breaks the zero-overhead principle in search for generality.

A library cannot, by itself, eliminate basic problems with host language semantics. For example, in C and C++, aliasing problems persists so that a library cannot provide guarantees needed for confidence, transformations, and optimizations. Often, a library is (at least partially) defined in terms of its implementation; it is not specified as an entity separate from its host language implementation. This is not a fundamental problem, but it is a common problem, and often a serious one in comparison to a special-purpose language.

4.3 Preprocessed languages

Generating code from a higher-level language into a lower-level one has been popular for decades. For example early C compilers generated assembly code; early C++ compilers generated C code; GUI builders, CAD systems, IDL processors, modeling languages, etc., generate code in languages such as C, C++, Java, C#. That is, the language source is preprocessed into a host language. The resulting languages and language processors are referred to by many names, such as preprocessors, macros, genrators, wizards, builders, and meta-languages. One way of distinguishing an implementation of a language implemented by such techniques from a facility defined by such translation techniques is whether you can ever get an error message from the target language compiler. If you can it's a preprocessor; if not it's a compiler. For example, by that criteria, the original C and C++ translators (into assembler and C) were compilers whereas Ratfor, C macros, and Microsoft "wizards" rely on preprocessors. C++ templates are "right on the edge" in that they receive some compiler support (and will recieve significantly more in the future: concepts [21, 20]). However, compiler error messages sometimes fail to refer to the original template source and often do so spectacularly badly. In consequence, some programmers consider templates "like macros" and avoid them; many more avoid uses they consider nontrivial. Here, we consider preprocessed languages, rather than abstraction facilities integrated within a language.

The language (generator, macro-language, modeling language, whatever) defined by a preprocessor becomes yet another special-purpose language. It requires documentation, training, tool support. In particular, you need to use a preprocessor together with a matching tool chain and compiler. Unless the preprocessor is integrated into the tool chain and shipped with every implementation, this implies lock-in and slow upgrades. It is not uncommon for the preprocessor not to work with the most current version of the compilers and tools or the underlying language. The main reason is that the preprocessor implementer doesn't get access to those compilers significantly before their own users. This commonly leads to users having to make a painful choice between using the preprocessor or the latest and greatest compiler and other tools. This creates friction between the preprocessor users and any non-preprocessor users they collaborate with. The debugging, compatibility, and portability problems persist because old compilers don't just die. It can take a large organization the better part of a decade to get everyone up-

graded to the latest version (of something), just to fall behind again at the next release. For example, it took "forever" (almost a decade) to get C++ template implementations good enough for mainstream use. However, some users still rely on decade old compilers.

A preprocessed language tends to have problems interacting the type system of the host language. Having the same type system as the host language is often not good enough — after all, the purpose of a preprocessed language is to elegantly express things that cannot be expressed elegantly in the host language. Error detection and error reporting problems are just the most obvious examples of this. Concepts (a type system for types) [21, 18], as being developed to improve C++ templates' support for generic programming and template metaprogramming, is an example of a mechanism addressing the problem of mismatch of the type systems of a higher-level language and a lower-level host language. Higher order types fills some of the same role in the specification of abstractions in functional languages.

So, a preprocessed language share many problems of with a special-purpose language with a stand-alone implementation. In fact, as their tools become more complete and their definition more precise and separate from the host language, they grow into special-purpose languages. Conversely, if their implementation and type system support becomes more integrated with the host language, they cease to be separate languages and become abstraction mechanisms of the host language (C++ templates is a prominent example). In addition, preprocessing languages tend to suffer the problems of libraries: Unless all code conforms to the conventions of the preprocessed language, the guarantees the language can rely on and offer weakens.

4.4 Dialects

Take a popular general-purpose language, add desired features to a compiler and/or a run-time support system, and you have your own private dialect. This may be the most popular way of creating a new language. The result is not quite a special-purpose language, but it has special-purpose features embedded in a general-purpose language. Working in a production-quality general-purpose language implementation is hard, though. Many people will simultaneously be making modifications in such an implementation. Furthermore, compilers, debuggers, libraries, tools are required parts of such implementations and major implementations target many platforms. Consequently, most people who extend a language in this way do so in a minor — less messy — implementation, modifying only the part of the tool chain they need, and target only the platforms they care about. This is reasonable — in many cases even essential — to allow people to focus their efforts on the design and implementation of the new facilities they want. Unfortunately, the effect is that unless the major vendors adopt the new dialect, its designers are left with a private language. This implies all the usual private language costs — and the usual mortality rate. In addition, it is essentially impossible to remove undesirable features from a dialect. Doing so would destroy compatibility and basically move the language away from the dialect classification and into the special-purpose language classification.

5 The SELL approach

The analysis in sections 3 and 4 paints a grim picture of the problems of applying language design and implementation techniques to support software development. One conclusion would be to leave the field to big corporations with deep pockets: Let them do the design, development, and apply their marketing muscle; then we live with the results, whatever they may be. An alternative conclusion is to withdraw into some cosy ghetto of our own design and let the rest of the world do what it likes without interference or input from us. I like neither alternative and point to a way to dodge the horns of this dilemma:

1. superset: Add libraries to provide application-specific facilities, then
2. subset: Subtract features (outside the library implementation) to provide semantic guarantees

The result is a subset of a superset of a language called a *Semantically Enhanced Library Language*. When subsetting we can aim at a "clean and regular" language. Since a SELL will aim for a narrower application domain than its host languages, we have a good chance of the result being simpler than its host.

We must consider this approach in terms of expressiveness ("can we really express things as well in a library as in a special-purpose language?") and tools ("will we get stuck developing and maintaining a messy tool chain?"). The claim is that the answers can be "yes" and "no" for a large enough range of problems and a low enough cost to prefer the SELL approach over the traditional approaches mentioned in sections 3 and 4. Obviously, the SELL approach is not completely new — in fact, it is an attempt to synthesize what has worked best in the traditional approaches and dodge the worst problems. Please also note that I don't claim that the other approaches to making special-purpose features available never work or that there are no other alternatives. That would be absurd. What I do claim is that the success rate for new languages — if measured by survival of a language for a decade and use outside the group that originated it — is very low and the costs higher than often realized.

The argument about expressiveness of libraries is based on a pair of old Bell Labs sayings:

1. Library design is language design
2. Language design is library design

We need both. In other words, the expressiveness of a library depends on the ability of a general-purpose language to define libraries. Functional programming, object-oriented programming, and generic programming are prominent schools of thought that give a prominent role to library building.

The skills needed to write a good library are very similar to the skills needed for all high-end systems programming or application building. Furthermore, when we write a library, we can rely on existing infrastructure (compilers, debuggers, libraries, education, etc.). The result is that libraries are cheap to produce compared to alternatives.

However, the tools part could easily lead us into the debugging, tool chain, and maintenance problems characteristic of

dialects and preprosessors. To avoid that we need a tool for expressing constraints and high-level transformations that is minimally invasive into the tool chain. To further keep the tool problems under control, we need a general tool for doing that and one that will fit into all tool chains. That is, we need a general-purpose tool for analysing source code and performing source-level transformations that relies on a standard interface to compilers.

5.1 C++

In principle, any general-purpose programming language can be the host language for the SELL approach. Unsurprisingly, my favorite / chosen host language is C++ [19, 8]

C++ has the virtues of stretching to a very broad range of application areas, good performance, a large and lively user community, and support for compilers, libraries, and tools for essentially all platforms [22].

C++'s abstraction facilities provide adequate support for object-oriented programming, generic programming, traditional procedural programming, and multi-paradigm programing combining elements of those. Classes plus templates plus overloading is the basis of expressiveness and performance.

Obviously improvements are possible — even given the Draconian compatibility constraints imposed by the huge user community and the wide range of application areas. In particular, we hope that the next standard (C++0x) will offer concepts (a type system for types), more general and flexible facilities for initilization, and remedies for many minor annoyances [20]. Unfortunately, the compatibility constraints and the use of C++ for very low-level system components precludes remedying obvious weaknesses, such as overly agressive implicit conversions (incl. the array-to-pointer conversion) and unchecked unions.

5.2 A brief overview of the Pivot

The Pivot is a general framework for the analysis and transformation of C++ programs[13]. The Pivot is designed to handle the complete ISO C++, especially more advanced uses of templates and including some proposed C++0x features. It is compiler independent.

There are lots of (more than 20) tools for static analysis and transformation of C++ programs, e.g. [15, 2, 16, 12]. However, few — if any — handle all of ISO Standard C++ [8, 19], most are specialized to particular forms of analysis or transformation, and few will work well in combination with other tools. The design of the Pivot is focussed on advanced uses of templates as used in generic programming, template metaprogramming, and experimental use of libraries as the basis of language extension. Since (static) types is central to such libraries, the SELL approach requires a representation that deals with types as first-class citizens and allows analysis and transformation based on their properties. In the C++ community, this is discussed under the heading of *concepts* and is likely to receive some language support in the next ISO C++ standard (C++0x) [21, 18, 20].

The central part of the Pivot is a fully typed abstract syntax tree called IPR (*Internal Program Representation*):

Figure 1. An overview of *The Pivot* infrastructure

To get IPR from a program, we need a compiler — only a compiler "knows" enough about a C++ program to represent it completely with syntactic and type information in a useful form. In particular, a simple parser doesn't understand types well enough to do a credible general job. We interface to a compiler in some appropriate (to a specific compiler) and minimally invasive fashion. A compiler-specific IPR generator produces IPR on a per-translation-unit basis. Applications interface to "code" through the IPR interface. So as not to run the compiler all the time and to be able to store and merge translation units without compiler intervention, we can produce a persistent form of IPR called XPR (*eXternal Program Representation*).

The IPR is complete and arguably minimal. Traversal of C++ code represented as IPR can be done in several ways, including "ordinary graph traversal code", visitors [6], iterators [19], or tools such as Rose [15]. The needs of the application — rather than the IPR — determines what traversal method is most suitable.

Currently, the Pivot does not support an annotation language. Pivot programs can annotate IPR nodes, but there is no facility for the programmer to embed annotations in the C++ source text. Providing such a facility is easy, but once programmers starts to depend on such annotations, they have created a new special-purpose language. We want to explore how much can be done with the SELL approach, relying only on standard conforming C++ source text.

6 Examples of SELLs

The proof of the pudding is in the eating, but this is not a paper presenting you with a SELL for use; it is a presentation of the general idea of SELLs. Therefore, I present only details that will illustrate the idea of a SELL, not complete SELLs.

6.1 Safe C++

C++ inherits a host of opportunities for type violations from C and adds a few of its own. It is possible — and not very hard — to write type-safe code in C++. However, it is not

easy to know that no type violations exist in a program, especially in a large program written and maintained by many programmers with a variety of backgrounds and a variety of ideas of what constitutes safe code. So, how would we support a type-safe dialect of C++ that maintains the essential expressiveness and efficiency of C++? In particular, we want to be sure that there are no type violations in the code. We can only be really sure if we can provide a tool (or combination of tools) that will detect all violations. In the absence of tools, we must rely on humans to follow rules. That would probably be better than the state of the art in most software development organizations, but it would only be second best.

Consider the major insecurities in C++ code:

1. Buffer overruns — i.e., reading or writing outsider the range of an array
2. Dereferencing an uninitialized pointer, a zero-valued pointer, or a pointer to a deleted object
3. Misuse of a union — i.e. write a union variable as one type and read it as another
4. Misuse of a cast — e.g. cast an int to a pointer type where no object of that type exist where the new pointer points
5. Misuse of `void*` — e.g. assign an `int*` to a `void*` and cast that `void*` to a `double*`
6. Deleting an object twice, not deleting an object after use, or using a pointer after deletion.

The obvious approach for avoiding these problems is to provide a library (or a set of libraries) that saves the programmer from having to use these error-prone features. For example, instead of using arrays, the programmer can use a range-checked `vector` and instead of a `union` a user can use a tagged `union` or an `Any` type. Casts (with exception of the dynamicaly type-safe `dynamic_cast`) and `void*`s are rarely useful outside low-level and easily encapsulated uses, so they can simply be avoided. If we use counted pointers, memory leaks won't happen (depending on how cyclic data structures are handled). Since pointers are checked, we don't access through invalid pointers and double deletion are easily detected.

Basically, errors that cannot be detected until run-time are systematically turned into exceptions, making *Safe C++* a dynamically type safe language. Exceptions may not be your favorite language feature, but they are useful in most contexts and are universally used for reporting run-time type violations in languages deemed type-safe.

So, we can fairly easily write code that doesn't suffer from the obvious type-safety problems. What is outlined here is a SELL where the superset is created by adding checked `vector`s, "smart" checked pointers, a tagged `union` (or an `Any` type). However, nothing has been gained if users persist using the unsafe-features in unsafe ways. For example, we can write safe code, but someone might just do something like this:

```
double* horrible(int i)
{
    int v[80];
    char* p = new char[200];
    double* q = new double[200];
    Shape* pc = new Circle(Point(10,20),20);
    delete[] p;
    p[100] = 'c';
    p[i] = 'x';
    v[100] = 666;
    pc->rotate(45);
    pc->draw();
    f(pc);
    void* vp = v;
    delete vp;
    delete[] p;
    return q;
}
```

Obviously, the subsetting (enforcement) part of the SELL design must be to detect and eliminate the unsafe uses of the host language. Please note that the tool that does that must distinguish between the use of the "banned" features or uses of features within the implementation of the extensions and direct use by the user. In this case, a dumb tool (such as a compiler option) banning all uses of pointer would prevent the use of `vector` that uses pointers internally. Instead, we could use the Pivot to catch only the uses of pointers outside our supporting classes. That done, our code would have to be rewritten to look something like:

```
unique_ptr<vector<double>> messy(int i)
{
    vector <int> v(80);
    string p(200);
    vector<double> q(200);
    scoped_ptr<Shape> pc(new Circle(Point(10,20),20));
    p[100] = 'c';      // ok
    p[i] = 'x';        // checked at run time
    v[100] = 666;      // caught at run time
    pc->rotate(45);
    pc->draw();
    f(pc);
    return unique_ptr<vector<double>>(q);
}
```

This is much better (ignoring the messy use of "magic constants"), but *Safe C++* could have problems for real-world programming in many areas where C++ is used: We have not dealt with performance and compatibility. Actually, this code hints of a very significant concern for performance in the library design: `scoped_ptr` deletes its object at the end of scope and prevents `f` from keeping a reference to that object. Similarly, `unique_ptr` cooperates with `vector` to ensure that the elements of `q` are transferred out of `messy` and not destroyed as part of `q` upon exit. We didn't just rely on counted pointers of a garbage collector to deal with resource problems.

Using the Pivot, we could do better, though. By default, both uses of `pc` in `messy` must be checked for validity (assuming that a `scoped_ptr` can be a null pointer). However, a bit of simple flow analysis can eliminate the second check, and a slighly more clever analysis will reveal that no checking is actually necessary: We can see that `pc` has not been properly initialized and not assigned to — and so can the Pivot. This kind of analysis has been used experimentally for private languages and dialects [9]. Given the Pivot, we can apply this for a library or for "raw C++".

Compatibility is a harder problem. What if `f` is not known to be safe? What if we can't rewrite or recompile all the code of a system? What if layout compatibility of some data structures

is required? *Safe C++* as presented here is just an illustration, not a full-blown SELL.

6.2 Parallel C++

With the emergence of cheap multiprocessors, clusters, and multi-core chips, concurrency is increasingly important. Many languages and dialects have been designed to address the concurrency needs of high-performance scientific computing. Here I will build on a library, STAPL [1] [14], that offers parallel operations on containers in the spirit of the STL. For example:

```
void f(pvector<double>& v)
{
     prange<double> r = find_all(v.range(),criteria);
     sort(r);
     cout << r;  // ordinary serial output of elements
}
```

Imagine that `v` has 500 million elements and that the program runs on a serious supercomputer, such as Blue Gene\L[3] (where STAPL is in fact used). The `find_all` will execute in parallel on as many processors as the STAPL run-time system deems reasonable finding elements that meets `criteria`. If `find_all` finds lots of elements, then `sort` will also use many processors.

Here we have a sophisticated library combined with an even more advanced run-time support system. What can the Pivot do to help? For starters, it can produce the information that the run-time support system needs to function well. Secondly, it can provide classical flow analysis and aliasing information. Finally, it can be programmed to recognize usage patterns to allow algorithm substitution (as in the initial matrix algebra example) and alert the programmer to likely problems or opportunities.

6.3 Real-time C++

The problems of real-time code for embedded systems combine concerns for correnctness, reliability, and performance in constrained circumstances. Some problems and solutions overlap with those of *Safe C++* but others are unique in that they require that every operation is performed in a known constant time (or less). Naturally, not all real-time and embedded systems are written under this Draconian rule, but let's see how we can address those that do. Some C++ operatons become unusable:

1. free store (general `new` and `delete`)
2. exceptions (assuming inability to easily predict the cost of a `throw`)
3. class hierarchy navigation (`dynamic_cast` in the absence of a constant time implementation [7])

First, we add a suitable support library:

1. a fixed size `Array` class (no conversion to pointer, knows its own size)
2. some safe pointer classes
3. memory allocation classes that guarantee constant time allocation (and deallocation if allowed) — pools, stacks, etc.
4. ...

Next, we use the Pivot to eliminate dangerous operations (as listed in §6.1) from user code.

In principle, this will do the job. However, we can do more. For most programs of this sort, we can do whole-program analysis. Such programs tend to be relatively small and not allow dynamic linking. Thus, the Pivot could be used to allow exceptions for error reporting: we can verify that every exception is caught and calculate the upper bound for each throw. This is a special — and espicially hard — example of using a tool to verify that resource consumption is within acceptable bounds.

In general, there are lots more that the Pivot can do in the context of embedded systems. Some depends on a specific application, so the boundary between SELL and application support blurrs. For example, it is not uncommon for an embedded program to be more permissive about the facilities that can be used during a startup phase. The SELL can define what "startup" means (e.g. called from `start_up`) and only apply the stringent rules outside that.

7 Conclusions

The first half of this paper outlines the problems facing programmers providing and using a special-purpose language defined in the most common ways: as a separate language, as compiler options, as libraries, using a pre-processor for a general-purpose language, and as a dialect. The picture painted is bleak, leading to a suggested alternative: *Semantically Enhanced Library Languages* (SELLs) The SELL approach offers a practical and economical alternative to the more common ways of implementing extensions, dialects, and special-purpose languages. By using libraries, it limits the problems with compatibility and tool chains. By adding tool support, it enhances the appeal of libraries.

8 Acknowlegements

Thanks to Gabriel Dos Reis and my students for reading drafts of this paper and making many detailed suggestions. Also thanks to the anonymous reviewers for making more major suggestions, some of which will require several more papers.

9 References

[1] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library* In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.

[2] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, Eelco Visser: *Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs*. `http://www.codeboost.org/`.

[3] IBM: `http://www.research.ibm.com/bluegene/`.

[4] William R. Mark, et al: *Cg: A System for Programming Graphics Hardware in a C-like Language*. Proceedings of SIGGRAPH 2003.

[5] M. Fernandez, et al: *SilkRoute: A framework for publishing relational data in XML*. ACM Trans. Database Syst. 27(4): 438-493 (2002)

[6] Erich Gamma, et al: *Design Patterns*. Addison-Wesley, 1994.

[7] Michael Gibbs and Bjarne Stroustrup: *Fast Dynamic Casting*. Software—Practice & Experience. Vol 35, Issue 686. 2005.

[8] International Organization for Standards, *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd ed., 2003. Wiley 2003. ISBN 0-470-84674-7.

[9] Trevor Jim, et al: *Cyclone: A Safe Dialect of C*. USENIX Annual Technical Conference, pages 275–288, Monterey, CA, June 2002.

[10] Lengauer, et al: *Domain-specific program generation*. Revised papers deom Dagstuhl seminar. March 2003. LNCS 3016.

[11] Diane J. Litman, Anil K. Mishra, and Peter F. Patel-Schneider: *Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules*. Proc. 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97). October 1997. `http://www.research.att.com/sw/tools/r++/` and `http://www.bell-labs.com/project/r++/`.

[12] George C. Necula, et al: *CIL: Intermediate Language and Tools for Analysis and Transformation*. `http://manju.cs.berkeley.edu/cil/`.

[13] The pivot is a program analysis and transformation infrastructure being developed at Texas A&M University.

[14] Steven Saunders, Lawrence Rauchwerger: *ARMI: An Adaptive, Platform Independent Communication Library* In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), pp. 12, San Diego, CA, Jun 2003. `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`.

[15] Markus Schordan and and Daniel Quinlan. *A Source-to-Source Architecture for User-Defined Optimizations*. In Proc. of the Joint Modular Languages Conference (JMLC'03), Volume 2789 of Lecture Notes in Computer Science, pp. 214-223, Springer Verlag, June 2003. (Rose).

[16] S. Schupp, D. P. Gregor, D. R. Musser, and S.-M. Liu. *Semantic and behavioral library transformations*. Information and Software Technology, 44(13):797 810, October 2002. (Simplicissimus).

[17] Jeremy G. Siek Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra*. ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[18] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. *Concept for C++0x*. Technical Report N1758=05-0018, ISO/IEC SC22/JTC1/WG21, January 2005.

[19] B. Stroustrup, *The C++ Programming Language*, special ed., Addison-Wesley, 2000. ISBN 0-201-70073-5 .

[20] B. Stroustrup: *The design of C++0x*. The C/C++ Users Journal. May 2005.

[21] B. Stroustrup, G. Dos Reis: *A concept design*. Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21, April 2005.

[22] B. Stroustrup: *Examples of C++ applications*: `http://www.research.att.com/~bs/applications.html`. *Some C++ compilers*: `http://www.research.att.com/~bs/compilers.html`.

[23] Tod Veldhiusen: *Arrays in Blitz++* ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[24] Wilson and Lu (editors): *Parallel programming using C++*. Addison-Wesley. 1996. ISBN 0-262-73118-5.

Sibling Rivalry: C and C++

Bjarne Stroustrup

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

This article presents a view of the relationship between K&R C's most prominent descendants: ISO C and ISO C++. It gives a rough chronology of the exchanges of features between the various versions of C and C++ and presents some technical details related to their most significant current incompatibilities. My focus here is the areas where C and C++ differ slightly (''the incompatibilities''), rather than the large area of commonality or the areas where one language provide facilities not offered by the other. In addition to presenting incompatibilities, this paper briefly discusses some implications of these incompatibilities, reflects on the ''Spirit of C'' and ''Spirit of C++'' notions, and states some opinions about the relationship between C and C++. This article is written in support of the view that C/C++ incompatibilities can and should be eliminated.

1 Introduction

Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. For example, C89 [C89] [Kernighan,1988] has no Boolean type, so people define their own, C++ [C++98] [Stroustrup,2000] addresses that problem by defining the type ***bool***, whereas C99 [C99] provides a type ***_Bool*** and a macro ***bool***. Such incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of tools for C and C++.

I write this memo to illustrate the current compatibility problems, to help people appreciate the origins of these problems, and to support a merger of C and C++ as the way to maximize the degree of compatibility, portability, and growth within the C/C++ community. It is my claim that the current incompatibilities arose from ''historical accident,'' short-term concerns, and overly narrow focus, rather than being rooted in fundamental differences between C and C++. Please note that I know full well that short-term concerns can be compelling, that a focus can sometimes be recognized as overly narrow only in retrospect, and that a ''historical accident'' can be made after seriously and competently considering all factors that appear relevant on the day. Please also note that I do not claim that I didn't contribute to incompatibilities in this way; neither individuals nor committees are infallible. This paper does not attmpt to demonstrate the value of C/C++ compatibility nor does it suggest resolutions for the incompatibilities presented.

My focus here is the areas where C and C++ differ slightly (''the incompatibilities''), rather than on the large area of commonality or the areas where on language provide facilities not offered by the other.

2 A Family Tree

How can I call C and C++ siblings? Clearly, C++ is a descendant of C. However, look at a family tree:

A solid line means a massive inheritance of features, a dashed line borrowing of major features, a dotted line borrowing of minor features.

From this, ISO C and ISO C++ emerge as the two major descendants of K&R C, and as siblings. Each carries with it the key aspects of Classic C, and neither is 100% compatible with Classic C. For example, both siblings consider ***const*** a keyword and both deem this famous Classic C program non-standard-compliant:

```
main()
{
    printf("Hello, world\n");
}
```

As a C89 program, this has one error. As a C++98 program, it has two errors. As a C99 program, it has the same two errors, and if those were fixed, the meaning would be subtly different from the identical C++ program.

To simplify, I have left influences that appeared almost simultaneously in both language during standardization unrepresented on the chart. Examples of that are ***void**** for C++ and C89, and the ban of ‘‘implicit ***int***’’ in C++ and C99.

Classic C is basically K&R C [Kernighan,1978] plus structure assignment, enumerations, and ***void***. I picked the term ‘‘Classic C’’ from a sticker that used to be affixed to Dennis Ritchie’s terminal.

When it comes to early influences on C89, C with Classes [Stroustrup,1982] and the earliest versions of C++ are indistinguishable. Similarly, the effects of the C standards effort was felt on C++ earlier than the publication of *The Annotated C++ Reference Manual* [ARM] in 1989, but I see little advantage in elaborating the chart with Cfront version numbers. Consequently, I simply lump together the 1984-1988 C++ releases as ‘‘Early C++’’ [Stroustrup,1986].

Incompatibilities are nasty for programmers in part because they create a combinatorial explosion of alternatives. Leaving out Classic C for simplicity, consider a simple Venn diagram:

There are features belonging to each of the seven areas:

C89 only	call of undeclared function
C99 only	variable length arrays (VLAs)
C++ only	templates
C89 and C99	Algol-style function definitions
C89 and C++	use of the C99 keyword ***restrict*** as an identifier
C++ and C99	*//* comments
C89, C++, and C99	***struct***s

For each language feature, a programmer must remember to which language the feature belongs and what its meaning is. That is a cause of confusion and bugs. For each feature, an implementor must allow it for the appropriate language only. This becomes much worse when various proprietary extensions and compiler switches are taken into account.

One of the big questions for the C/C++ community is whether the next phase of standardization (potentially adding two more circles to the diagram) will pull the languages together or tear them further apart. In ten years, there will be large and thriving C and C++ communities. However, if the languages are allowed to drift further apart, there will not be a C/C++ community, sharing tools, implementations, techniques, headers, code, etc. My nightmare scenario looks a bit like this:

Each separate area of the diagram represents a set of incompatibilities that an implementer must address and that a programmer may have to be aware of.

In the following sections, I present the major influences among C and C++ versions. I do not try to be comprehensive, but to highlight issues that affect compatibility. This discussion clearly reflects a C++ point of view. I was there when the C++ decisions were made, so I can give reasons. I did not attend C standards committee meetings, so my knowledge about decisions there are second hand. However, this article is not an attempt to demonstrate that the C++ design decisions are preferable to the C design decisions. Should someone decide to eliminate some or all of the current compatibility problems then ‘‘just adopt the C++ rules’’ is as unrealistic a policy as ‘‘just follow the C rules.’’

For the discussion of features, I rely primarily on memory†, checked by lookup in [C89], [C++98],

† I have been a member of the C/C++ community for more than 25 years. I used BCPL [Richards,1980] on and off from 1973 until 1979. I first used C in 1975 and worked in Bell Labs’ computer science research center alongside people such as Dennis Ritchie and

[C99] [Kernighan,1978] [D&E], and various notes.

The differences between C++ and C89 are documented in Appendix C of the ISO C++ standard [C++98]. The differences between C++ and C99 are not officially documented because the ISO C committee had neither the time nor the expertise to do so, and documenting C++/C99 incompatibilities was not required by the C99 committee's charter [Benito,1998]. An unofficial, but extensive list of incompatibilities can be found on the web [Tribble,2001]. See also Appendix B of [Stroustrup,2000].

2.1 From K&R C to Classic C

Pre-ANSI C is often referred to as K&R C. However, that is slightly incorrect. The C described in [Kernighan,1978] lacks three features of the language used by almost all C programmers before the emergence of C89: ***void***, enumerations, and structure assignment. These three features were added in PCC, the Portable C Compiler, developed by Steve Johnson and distributed as *the* C compiler by Bell Labs (with the ''blessing'' of Dennis Ritchie).

Adding ***void*** (used as a possible return type for functions only) allows a programmer to directly express that a function doesn't return a value, and allows the compiler to check that.

Similarly, adding enumerations allows a programmer to directly express that a group of values in some way belong together. It also supports the notion of manifest constants in a way that does not rely on macros.

Adding structure assignment (and also structure copy initialization, argument passing, and function return) makes ***struct*** values first-class citizens of C.

Thus, two of the three last additions to Classic C add to the expressive power of the type system without actually allowing a programmer to express any new computations. The third makes user-defined types, as then existing, equal to built-in types. In addition, one of the additions provides an alternative to the use of macros. These are all themes that recur in the design of C++.

2.2 From Classic C to C with Classes

''C with Classes'' [Stroustrup,1982] [Stroustrup,1983] was an almost completely compatible dialect of C that briefly flourished in the early 1980s before evolving into C++. The only incompatibility with Classic C was that new keywords, such as ***class***, ***new***, and ***public***, could no longer be used as identifiers. Strongly influenced by Simula [Birtwistle,1979], C with Classes introduced key C++ facilities such as classes, derived classes, access control, constructors, destructors, and the memory management operators ***new*** and ***delete***. It also introduced the notion of inlining and the rudiments of function and operator overloading.

C with Classes provided optional function argument type checking and argument conversion through function declarations in which argument types could be specified. For example:

```
void f(double d, class shape*);

void g(int i, class circle* p)
{
    f(i,p);     // invokes f((double)i,(class shape*)p) when circle is derived from shape
    f(p,i);     // error: wrong argument types
    f(i);       // error: 2nd argument missing
}
```

To distinguish between the Classic C ''*f* takes any number of arguments of any type'' and ''*f* takes no arguments,'' I introduced new notation:

```
int f1();          /* K&R C and C with Classes: f1 takes any number of arguments of any type */
int f2(void);      // C with Classes: f2 takes no arguments
int f3(...);       // C with Classes: f3 takes zero or more of arguments of any type
int f4(int ...);   // C with Classes: f4 takes an int followed by zero or more arguments of any type
```

Like in Classic C, the use of function declarations were optional.

C with Classes introduced the BCPL //-comments.

Brian Kernignan 1979-1996. I took part in the internal Bell Labs standardization of C in the early 1980s together with Larry Rosler, and I took part in the efforts to standardize C++ from a about year before that work formally started in late 1989, attending almost all meetings.

C with Classes introduced ***const***, initially called ***readonly*** [Stroustrup,1981]. It served two functions: as a way of defining a symbolic constant that obeys scope and type rules (that is, without using a macro) and as a way of deeming an object in memory immutable. The model for ***const*** was simple: In principle, ***const*** could be implemented by tagging each object by a const/non-const bit. For a ***const***, that bit is set after initialization. Once that bit is set, the object can no longer be modified. For example:

```
void f()
{
    const int c = 5;
    c = 6;                  // error
    *((int*)&c) = 6;        // error, but a compiler can't catch all such violations
}
```

To ease the use of ***const***s as symbolic constants and to make it trivial for a compiler to avoid unnecessarily storing simple ***const***s in memory, a global ***const*** was by default local to its translation unit. For example:

```
const int x;            // error: const not initialized
extern const int y;     // must be defined and initialized elsewhere
const int max = 3;
int a[max];
int b[y];               // error: y is not a constant expression; its value is not known at compile time
```

Here, ***max*** need not be stored in memory. Storing a ***const*** as an object in memory is necessary only if someone takes its address. Note that C with Classes supported ***const***s in constant expressions.

Constant pointers, using the notation ****const***, with exactly the same model of ''constness'' were adopted based on a suggestion of Dennis Ritchie.

Late in its history, C with Classes began to support the notion of a pointer to ''raw memory,'' ***void****. The origin of ***void**** is shrouded in some mystery. I vaguely remember inventing it together with Larry Rosler and Steve Johnson. However, Dave Prosser remembers first having suggested it based on something used ''somewhere in Australia.'' Possibly both versions are correct because Dave worked closely with Larry at the time. In either case, ***void**** was introduced into both languages more or less at the same time. The earliest mention of ***void**** that I can find is in a memo dated January 1, 1983, about the memory management mechanisms provided by my C++ compiler, Cfront, so the origins of ***void**** in C++ must go back at least to mid-1982. The earliest written record of ***void**** in the context of ANSI C is a proposal by Mike Meissner dated ''12 Oct 83,'' which presented ***void**** essentially as it was accepted into ANSI C in June 1984 [Prosser,2001].

Whatever the origin, what was implemented in C with Classes was a simple type-safe notion of memory holding objects of unknown type. Any pointer can be implicitly converted to ***void****, and any use of the memory referred to by a ***void**** involves a cast to some type. For example:

```
void f()
{
    int* pi = new int;
    void* pv = pi;
    double* pd = pv;            // error
    double* q = (double*)pv;    // ok: on your head be it
}
```

One of the most visible aspects of C with Classes compared to C was that memory was managed using the operators ***new*** and ***delete***, rather than by functions such as ***malloc()*** and ***free()***. The fundamental reason for operators in C with Classes came from the need to guarantee initialization of class objects. However, ***new*** also solved an old problem in C: How to express a free store allocation without a cast. For example, a typical K&R C free store allocation was handled like this:

```
char *calloc();                         /* K&R-style function declaration */
int *ip = (int *) calloc(10, sizeof(int));   /* allocate space for 10 ints */
/* ... */
free(ip);
```

In C with Classes this became

```
int* p2 = new int[10];      // allocate 10 ints
/* ... */
delete[] p2;
```

In addition to eliminating the need to cast, ***new*** eliminates the possibility of specifying the wrong size for an allocation. Characteristically, the run-time performance of ***new/delete*** and ***malloc()/free()*** were close to identical. In general, the run-time support for C with Classes was the C run-time support with a slightly different interface. The ***new*** operator closely resembles a common C practice:

```
#define ALLOC(T) ((T*)calloc(1,sizeof(T)))
/* ... */
int* p3 = ALLOC(int);
/* ... */
free(p3);
```

To allow functions to be used where Classic C tended to use macros for efficiency, C with Classes introduced the notion of inline functions. The primary use of inline was to provide maximally efficient access functions for classes, inspired by use of C with Classes in embedded systems. Member functions defined in-class are inline by default; non-member functions can be declared inline by using the ***inline*** keyword.

2.3 From Classic C and C with Classes to C89

The ANSI C standards effort (which became the ISO C standards effort) pulled together the various C dialects and prevented further language divergence. C with Classes and later C++ were not ready for standardization until after the C89 standard was cast in stone. However, several ideas from C with Classes, and later C++, were obvious candidates for C.

C89 adopted function prototypes in a form very similar to what C with Classes provided, but significantly different from what C++ offered in 1984 (see §2.4). C89 also deemed the behavior calls of undeclared varadic functions undefined. For example:

```
void f()
{
    printf("Oops!\n");          /* error, usually not caught by compiler */
}

int printf(const char *, ...); /* from <stdio.h> */

void g(i)
{
    printf("Ok!\n");            /* ok: varadic function declared */
    printf("i = %d\n", i);
}
```

C89 adopted ***const***, but in a form that differed significantly from what C With Classes and C++ provided. C's ***const*** differs from C++'s in that a global ***const*** by default has external linkage and are not allowed in constant expressions. For example:

```
const int x;                /* assume initialized elsewhere */
static const int max = 3;
int a[max];                 /* error: const not allowed in constant expression */
(*(int*)&max) = 7;          /* error, but not caught by compiler */
```

The default linkage of variables and functions in Classic C is external linkage. C89 chose consistency with that, whereas C with Classes chose rules to make ***const*** and inlines by default behave more like macros and structs in respect to linkage. By leaving the default linkage of a ***const*** external, C89 made it necessary to represent all ***const***s as objects in memory. Unfortunately, that implies that C89 ***const***s incur overheads compared to macros. Possibly for that reason, C89 ***const***s differ from C++ ***const***s in not being allowed in constant expressions.

C89 introduced ***void**** in a form that significantly differed from the one offered by C++. C89 allows implicit assignment of a ***void**** to any pointer type. For example:

```
void* malloc(size_t);        /* from standard header */
#define NULL (void*)0        /* from standard header */

int f()
{
    double* pd = malloc(sizeof(int));    /* ok: void* converts to pointer type */

    float* pf = NULL;    /* ok: void* converts to pointer type */
    int x = NULL;        /* error: void* doesn't convert to int */

    char i = 0;
    char j = 0;
    char k = 0;
    char* p = &j;
    void* q = p;
    int* pp = q;    /* unsafe, legal C, not C++ */

    *pp = -1;    /* overwrite memory starting at &j, typically including i or k */
}
```

This example illustrates both the strength and the weakness of C's ***void**** compared to C++'s ***void****. Because C++ had the ***new*** operator, it had no need to open a loophole in the type system to allow ***malloc*()** to be used conveniently (without a cast). On the other hand, C89's definition of ***void**** allows a definition of the null pointer that can't be assigned to an ***int***. I believe this to be the only point where C is more strongly typed than C++.

2.4 From C with Classes to Early C++

During the 1983-1985 period, I reimplemented C with Classes, redesigned it, renamed it (twice), and had it released [D&E]. The renaming was prompted by the relation between C with Classes and C, and the name C++ represented the emergence of C++ as a separate language – as opposed to a dialect. How being a language differs from being a dialect is not exactly clear, but the aim of being completely compatible except for new keywords was abandoned sometime in late 1983.

The primary aim of the evolution of C with Classes into C++ was to strengthen the abstraction mechanisms and to improve type checking. Some of the design changes affected C/C++ compatibility.

I abandoned the use of ***void*** as an argument type meaning ''no arguments'' after Dennis Ritchie and Doug McIlroy strongly condemned it as ''an abomination.'' Instead, I adopted the obvious notation for taking no arguments, an empty pair of parentheses. For example:

```
int f(void);    // error: abomination
int g();        // g takes no argument
```

After observing the effect of having optional function declarations on code and on programmers for about a year, I made function declarations compulsory. That is, in C++ you cannot call an undeclared function:

```
int main()
{
    double d = sqrt(2);   // error: sqrt() not declared
}
```

Clearly, this tightening of the rules caused many C programs not to be C++ programs, but the improvements in error detection and the ease of converting legal C code to C++ meant that this never became a serious problem.

During the transition to C++, the C Algol-style function definition syntax became redundant. For a short while they were accepted for compatibility only. Finally I banned them to avoid confusion†. For example:

† Note that in C89, an Algol-style definition differs semantically from a prototype-style function definition.

```
void f(a,p) char *p; /* a is an int */        // error: not C++
{
    /* ... */
}
```

Each enumeration is a separate type (in C and C++). In C++, this implies that you can overload a function for an enumeration. This again implies that the type of an enumerator must be of the type of its enumeration. For example:

```
enum E { a, b };
void f(E);
void f(int);

void g(E x, int y)
{
    f(x);       // calls f(E)
    f(y);       // calls f(int)
}
```

In C, the type of an enumerator is ***int***. This distinction is not significant in C, but it is in C++. Trying to assign an ***int*** to a enumerated type is an error in C++. For example:

```
E x = 7;        // error: int assigned to E
E x = E(7);     // ok: on your head be it
int i = a;      // ok: enumerators converts to int
x++;            // error: attempt to assign the int x+1 to the E x
```

Thus, the rules for enumerations can lead to compatibility problems.

As shown, C++ does not require the keyword ***enum*** to be used in front of enumeration names. Similarly, ***struct*** is not required in front of structure names. Interestingly, this has not led to compatibility problems. The reason is that considerable effort and ingenuity was expended on this potential problem. If a name is defined as both a structure tag and as an ordinary identifier in a scope, an unqualified use of the name is taken to refer to the non-***struct***. For example:

```
struct X { /* ... */ };
int X(int);
struct Y { /* ... */ };

void f()
{
    struct X a;     // the struct
    int i = X(2);   // the function
    Y b;            // the struct
}
```

It is possible to construct a C program that is not a C++ program based on structure tags [Stroustrup,2000 Appendix B], but it is not something you often see.

2.5 From Early C++ and C89 to ARM C++

The Annotated C++ Reference Manual [ARM] was published in 1989 and became the base document for the ANSI C++ standards effort†, starting with its first technical meeting in 1990. Before that, the various drafts of the ANSI C standard had been available for years and I had been able to take the first steps to increase C/C++ compatibility. The most important actions were simply to follow the draft C standard wherever there were no considered decision to do something different. That way, C++ got the C89 rules for ***unsigned***, the ***volatile*** keyword, etc. The ‘‘abomination’’

```
int f(void);    // f() takes no arguments
```

was re-incorporated and a redundant comma in declarations of varadic functions introduced into C89 was also accepted:

† More precisely: the reference manual of [Stroustrup,1991], including templates and exceptions, was the base document.

```
int f(int ...);        // C++
int g(int, ...);       /* C and C++ */
```

Thus, after an initial increase in diversion between the languages in the early 1980s, the degree of practical compatibility increased in the late 1980s. However, the foundation for further diversion was laid by C's introduction of a ***void**** that didn't require casting and by C++'s introduction of exceptions.

C++ exception handling complicates the run time support system and encourages people to rely on styles of error handling that are not supported by C. However, people concerned with compatibility could and did ignore exceptions, often relying on compiler options disabling exceptions and using only traditional (C style) run-time support.

Similarly, the introduction of templates tended not to affect compatibility beyond the introduction of the keyword ***template***. Template use was simply kept out of interfaces and code shared with C programmers.

In C, structure scopes that appear to be nested aren't, because structure names declared inside are considered to be in the outer scope. This proved to be unmanageable in C++ where nested classes were often used as implementation details. Consequently, C++ adopted nested structure scopes. For example:

```
struct X {
    struct Y {
        /* ... */
    };
    /* ... */
};

struct Y a;      /* ok in C, error in C++ */
X::Y b;          // error in C, ok in C++
```

ARM C++ and C89 are almost exactly contemporary. We missed an opportunity by the C and C++ communities not joining up to evaluate the incompatibilities and to jointly decide what to do about them. Instead, each language embarked on separate courses for the 1990s, leading to much confusion and some understandable, but unnecessary, sibling rivalry.

2.6 From ARM C++ to C++98

Most of the evolution from ARM C++ to ISO C++ [C++98] [Stroustrup,2000] focussed on the abstraction facilities, such as templates, namespaces, and run-time type information, and had little impact on C compatibility. C/C++ compatibility was taken very seriously by the C++ standards committees, every incompatibility with C89 was documented (Appendix C of [C++98]), and care was taken not to accidentally or unnecessarily increase the number of incompatibilities or their degree of seriousness.

In several cases, the text of the standard was adjusted to reflect the C standard in an attempt to avoid unintended incompatibilities arising from differences in wording, and in at least one case a rule was changed with no other purpose than to achieve compatibility. Declarations that differ only in ***const*** at the highest level of an argument type are considered identical. For example:

```
void f(const int);
void f(int);          // the same f() as "void f(const int)"
```

A Boolean type with associated keywords ***bool***, ***true***, and ***false*** was introduced. Similarly, the wide character type, ***wchar_t***, first introduced as a typedef in C89, was added. The fundamental reason for introducing new types was in both cases a desire to improve type checking and to use overloading based on those types. In particular, having ***wchar_t*** or ***bool*** as a typedef would not allow ***iostream*** operations to be properly implemented.

To ease the use of equipment with limited character sets, C++ introduced several new keywords, such as ***or***, ***and***, ***not***, and ***xor*** [Simonsen,1989]. For example:

```
if (a and (b or c))  // a && (b || c)
```

The use of ***static*** to mean ''local to this translation unit'' was deprecated in favor of the use of unnamed namespaces, creating a potential future C/C++ incompatibility.

Finally, after years of debate, ''implicit ***int***'' was banned. That is, every declaration must contain a type. The rule that the absence of a type implies ***int*** is gone. This simplifies parsing, eliminate some errors, and improves error messages.

One effect of making C++'s declarations more expressive than Classic C declarations was that much more information moved into header files. Consequently, the old issue of how to keep declarations in different translation units consistent became increasingly central to the definition of C++ features. In particular, the question ''what does it mean to be consistent?'' is key. When I first started the design of C with Classes, I had found K&R [Kernighan,1978] insufficiently clear about this and asked Dennis Ritchie about his intent. His answer was brief and clear: ''programs should behave as if there were exactly one definition for each function and structure'' [Ritchie,1981]. The Classic C policy was that if you couldn't enforce a simple rule, you kept the rule simple rather than changing it to something complicated that could be enforced with current technology. The rules then allowed compiler and linker implementors to fail to detect violations – leaving the task of better enforcement to specialized programs, such as lint, and future improved implementations.

I adopted both the rule and the policy for C++. The rule for separate compilation, the ''One Definition Rule'' (the ODR) insists that a program is illegal if a type is defined inconsistently in different translation units. The exact details are hard to express in a standard and impossible for a traditional C++ compiler to check.

A practical effect of this is that C++98 requires inline functions to be consistent across compilation units, and that entities referred to by templates and inline functions must also be consistently defined.

Exceptions had been introduced in ARM C++, but were labeled ''experimental.'' During the transition from ARM C++ to ISO C++, the design of the exception handling facilities remained remarkably stable. The main effort was to properly integrate exceptions with other facilities. In particular, the ***new*** operator was redefined to throw an exception if unable to allocate the memory required. The implication for most C++ programs was minimal; they don't deal with memory exhaustion beyond (more or less gracefully) exiting anyway. But the implication for C compatibility is profound. A program using the default ***new*** now requires the full exception handling run-time support. The support for exception handling can be orders of magnitude larger than the C run-time support, which is still sufficient to support all other parts of C++.

For people running applications on a full-blown operating system, this is still insignificant because the size of the operating system facilities are yet another couple of magnitudes larger. However, if you are working on a resource constrained embedded system or writing a device driver, the overhead can be prohibitive. Note that for many resource constrained systems, exception handling is neither prohibitively expensive nor too unpredictable. Often, the alternative mechanisms needed in the absence of exception handling are at least as costly and as hard to analyze. However, from a C/C++ compatibility point of view exception handling is unique in imposing a different model of error handling for C++ that requires a form of language run-time support not required by C.

If somebody decides that exception handling is undesirable for an application, there are three choices:

[1] Avoid C++

[2] Use only facilities in ISO C++ that do not require exception support. This is possible. For example, ***new*(*nothrow*)** will allocate objects on free store just like ***new***, but it returns ***0*** if it cannot allocate memory. Such avoidance of exceptions is sometimes simplified because the kind of system that must avoid using exceptions is often the kind of system where free store allocation is banned or severely limited.

[3] Use a (non-standard) compiler option that makes ***new*** work like ***new*(*nothrow*)**.

Usually, avoiding exception handling facilities is not sufficient to avoid exception support from being part of the language run-time support; some compiler/linker option is also needed to awaken the implementation to the opportunity to cut back on the run-time support.

2.7 From C89 and ARM C++ to C99

Compared to C89, C99 provides many small language changes and a few more substantial ones†. The general thrust of the changes from C89 to C99 is a massive increase in the support for conventional (Fortran-style) numerical computation. Most language and library changes have little directly to do with C++, but could have if C++ chooses to take that direction. Examples of this kind of extensions are variable length arrays (VLAs) and designated initializers. However, the focus here is the C99 extensions that have substantial overlap with C++ features.

† The foreword to the C99 standard lists 53 changes, mostly extensions.

C99 introduces ***bool*** as a macro for a built-in type ***_Bool***. Like ***bool***, ***true*** and ***false*** are macros defined in the standard header ***<stdbool.h>***.

C99 provides a header ***<iso646.h>*** with macros for the C++ keywords ***or***, ***and***, ***xor***, etc. (see 2.6).

C99 introduces a new keyword ***restrict*** (a revised version of the much-condemned ***noalias*** [Ritchie,1988]) to improve optimization. For example:

```
void munge(double *restrict p, double *restrict q);  // an optimizer may assume that p and q
                                                     //  point to non-overlapping arrays
```

C99 introduces support for complex arithmetic through several built-in types identified by keywords such as ***_Complex*** and ***_Imaginary***. The header ***<complex.h>*** provides macros, such as ***complex***, ***imaginary***, and ***I***, as the primary interface to complex types. The usual arithmetic functions are provided for complex numbers. To distinguish complex mathematical functions from floating-point mathematical functions the prefix ***c*** is used. Differences in scalar types are indicated by a suffix. For example:

```
double complex csin(double complex);          /* from <complex.h> */
float complex csinf(float complex);
long double complex csinl(long double complex);

double sin(double);                           /* from <math.h> */
float sinf(float);
long double sinl(long double);
```

Unfortunately, the name of the complex log function ***clog()*** now clashes with the name of the C++ logging stream, ***clog***. To use the conventional names for these functions, overloading is needed. However, C doesn't support overloading except for built-in operators, such as + and *, and for type generic math macros. If the type generic macro header ***<tgmath.h>*** is included, the standard mathematical functions become macros, and overload resolution is done almost as in C++. For example:

```
#include<tgmath.h>

void g(float f, double d, long double ld,
          float complex fz, complex z, long double complex lz)
{
     sin(f);     // sinf(f)
     sin(d);     // sin(d)
     sin(ld);    // sinl(ld)
     sin(fz);    // csinf(fz)
     sin(z);     // csin(z)
     sin(lz);    // csinl(lz)
}
```

These macros require ''compiler magic'' for their implementation because C99 does not provide facilities for specifying overloading. That is, C99 does not offer the facilities used to implement ***<tgmath.h>*** to ordinary users to deal with their own overloading needs.

The complex facilities are defined in a standard header ***<complex.h>***.

Unfortunately, both the model of complex numbers and the interfaces offered to them differ from both the traditional C++ class ***complex*** defined in ***<complex.h>*** and the standard library templated ***complex*** defined in ***<complex>***. For example, the C++ version of the C99 code above is:

```
complex<double> sin(const complex<double>&);          // from <complex>
complex<float> sin(const complex<float>&);
complex<long double> sin(const complex<long double>&);

double sin(double);                                   // from <cmath>
float sin(float);
long double sin(long double);
```

```
void g(float f, double d, long double ld,
        complex<float> fz, complex<double> z, complex<long double> lz)
{
    sin(f);     // call sin(float)
    sin(d);     // call sin(double)
    sin(ld);    // call sin(long double)
    sin(fz);    // call sin(complex<float>&)
    sin(z);     // call sin(complex<double>&)
    sin(lz);    // call sin(complex<long double>&)
}
```

Some of the syntactic differences between C++ complex and C99 complex can be plastered over using ''thin bindings'' (see §6.2) or ''compatibility headers'' (see the Appendix). However, subtle details of complex arithmetic (such as the treatment of ***imaginary***) and some conversion rules also differ.

C99 introduces //-comments , just like in C++.

As in C++, a declaration can be used where a statement is allowed.

As in C++, a for-initializer may be a declaration. However, C++'s use of definitions in conditions was not adopted. For example:

```
void f(int max)
{
    int sum = 0;
    for (int i = 0; i< max; ++i) {   // C++ and C99: declaration in for-initializer
        // ...
    }
    int s2 = sum+sum;                // C++ and C99: declaration as statement
    while (int x = check(s2)) {      // C++, not C99: declaration in condition
        // ...
    }
}
```

C99 adopted ***inline***, but with a linkage model that differs significantly from C++'s ODR (§2.6). For example, some language constructs are disallowed in C inlines (but not in C++):

```
// use of static variables in/from inlines ok in C++, errors in C:

static int a;
extern inline int count() { return ++a; }

extern inline int count2() { static int b = 0; b+=2; return b; }
```

This implies that a programmer wanting to write portable code must know the rules for ***inline*** in both C and C++.

In C, an inline function is by default local to its declaration unit. For example:

```
// in file x.c:
inline int f(int i) { return i+1; }

// in file y.c:
inline int f(int i) { return i+2; }
```

However, C++ requires global inlines to be identically defined in all translation units that define them.
The implication of this incompatibility is that ***inline*** cannot safely be used in headers shared between C++ and C99 – at least not by non-language lawyers†.

Like C++98, C99 bans ''implicit ***int***.'' For example:

† This is particularly sad because I have been assured by members of the C committee that their aim for C99 ***inline*** was C/C++ compatibility. Other members have assured me that these incompatibilities were deliberate and follow from fundamental differences between C and C++ views of linkage. Personally, I don't see these fundamental differences and am of the opinion that the *inline* incompatibilities could have been avoided by an application of Dennis Ritchie's ''if you can't enforce a simple rule, kept the rule simple rather than changing it to something complicated that can be enforced with current technology'' rule of thumb (§2.6).

```
const a = 10;    // error: no type in declaration of a
f(int);          // error: no type in declaration of f
```

The ban of ''implicit ***int***'' is a rare case where close cooperation between the C and C++ standards committees resulted in an almost simultaneous identical change in both languages. It is also an example of the committees finally removing a wart from the languages despite having to break backwards compatibility to do that.

3 Deprecation/Obsolescence

Both the C and C++ standards committees try to support a transition away from facilities deemed undesirable by including a list of deprecated/obsolescent features in the standard documents. Examples are:

[1] call of undeclared function (C89)
[2] the use of ***static*** to mean ''local to this translation unit'' (C++)
[3] use of Algol-style function definition (C99)
[4] use of empty argument list in function declaration (C89, C99)
[5] the ability to undefine the standard-library macros ***bool***, ***true***, and ***false*** (C99)

These lists express a committee's hope for the future and serve as a warning to programmers to avoid those features. Both K&R1 [Kernighan,1978] and early C++ manuals [Stroustrup,1986] effectively used such lists to doom undesirable features. For example, [1] and [3] can be found under ''anachronisms'' in [Stroustrup,1986]. Clearly, deprecation/obsolescence can be used to both to encourage and discourage C/C++ compatibility.

Banning calls of undeclared functions brings C99 and C++ into line. Banning Algol-style function definitions in the next revision of ISO C, C0x, would bring C99 and C++ into line. Banning the use of ***static*** to mean ''local to this translation unit'' (in favor of use of unnamed name spaces) in C++0x would increase the degree of C/C++ incompatibility. Banning

```
int f();    // f takes no argument (C++, deemed obsolescent in C89 and C99)
```

in C0x would be a disaster for C/C++ compatibility because every major C++ program contains such declarations.

Does deprecation work? That is, does a standard committee's wishes for the future really help the community to accept change? It can help where the gain from avoiding a feature is clear to the community, and it appears to fail when a gain is not clear. Deprecation is a valuable tool and could be more so if systematically supported by warnings and/or compatibility switches. Bringing C and C++ together will require an effort in this direction. Clearly, achieving C/C++ compatibility will require breaking backwards compatibility in some cases. This can be done only with the help of a mechanism for detecting obsolescent features and a mechanism for prohibiting their use in code meant to be long lived.

4 The Spirit of C

The phrase ''The spirit of C'' is brandished around, as is the complementary phrase ''The spirit of C++.'' These phrases are often used as weapons to condemn notions supposedly not in the right spirit and therefore deemed illegitimate. More reasonably, they can be used to distinguish languages aimed at supporting low-level systems programming, such as C and C++, from languages without such support. However, I find these notions poisonous when thoughtlessly applied in debates within the C/C++ community. For example, some condemn classes as ''not in the spirit of C'' and others condemn C-style strings as ''not in the spirit of C++.'' More often than not, these phrases dress up personal likes and dislikes as philosophies supposedly backed by ''the fathers of C'' or ''the fathers of C++.'' This can be amusing and occasionally embarrassing to Dennis Ritchie and me. We are still alive and do hold opinions, though Dennis – being the older and wiser – is better able to keep quiet.

Here are a few slogans often claimed to be or be part of ''the spirit of C:''

[1] keep the built-in operations close to the machine (and efficient)
[2] keep the built-in data types close to the machine (and efficient)
[3] no built-in operations on composite objects
[4] don't do in the language what can be done in a library
[5] the standard library can be written in the language itself

[6] trust the programmer
[7] the compiler is simple
[8] the run-time support is very simple
[9] in principle type-safe, but not automatically checked (use lint for checking)
[10] the language isn't perfect because practical concerns are taken seriously

All can be supported by quotes from the opening pages of K&R-1 [Kernighan,1978].

Naturally, Classic C is a good approximation to ''the spirit of C.'' C99 and C++ are less so, but they still approximate those ideals. This is significant because most languages don't. From the perspective of Ada, Java, or Python, C and C++ appear as twins. Only in discussions within the C/C++ community do the differences appear to overwhelm the commonalities.

In the spirit of [10], Classic C breaks [3] by adding structure assignment and structure argument passing to K&R C.

C++ starts out by breaking [7]: A greater emphasis on type and scope distinguishes C++ compared to C. Consequently, a C++ compiler front-end must do much more than Classic C front-end does. The introduction of exceptions complicates C++'s run-time support, violating [8]. However, that may be defended on the grounds that if you don't need exceptions, you can avoid using them (§2.6). After 20 years, it is more remarkable that C++ closely follows the remaining eight criteria. In particular, C++ can be seen as the result of following [1] to [5] to their logical conclusion by allowing the user to define general and efficient types and libraries.

Compared to early C compilers, modern C implementations cannot be called simple, so C99 also breaks [7]. Since ***<tgmath.h>*** cannot be written in C (though something almost identical could be written in C++), C99 breaks [5]. Arguably, C99's ***complex*** facilities violate [1], [2], and [3].

Contrary to popular myths, there is no more tolerance of time and space overheads in C++ than there is in C. The emphasis on run-time performance varies more between different communities using the languages than between the languages themselves. In other words, overheads are found in some uses of the languages rather than in the language features.

Why is ''the spirit of C'' of interest? It is worth calmly discussing ''the spirit of C'' because this topic has been used to inflame language wars and especially because what underlies those flame wars is often a genuine concern for the direction of evolution of C and/or C++. That is, a consistent aim/philosophy is needed for a coherent language to emerge from a set of changes and extensions.

In their evolution from Classic C, C99 and C++ differ in philosophy. C++ has a clearly stated philosophy of language: the emphasis in the selection of new facilities is on mechanisms for defining and using new types safely and efficiently. Basic facilities for computation were inherited, as far as possible unchanged, from Classic C and later from C89. C++ will go a long way to avoid introducing a new fundamental type. The prevailing view is that if you need one type then many programmers will need similar types†. Consequently, providing mechanisms for expressing such types in the language will serve many more programmers than would providing the one type as a built-in. In other words, the emphasis is on facilities for organizing code and building libraries (often referred to as ''abstraction mechanisms'').

To contrast, the emphasis in the evolution of C89 into C99 has been on the direct support for traditional (Fortran-style) numerical computation. Consequently, the major extensions of C99 compared to C89 are in new built-in numeric types, new mathematical functions and macros, new facilities for I/O of numbers, and extensions to the notion of an array. The contrasting approaches to complex numbers and to ***vector***s/VLAs illustrate the difference in C++'s and C99's design philosophies: C adds built-in facilities where C++ add to the standard library [Stroustrup,2002].

Ideally, C's emphasis on built-in facilities and C++'s emphasis on abstraction mechanisms are complementary. However, for that to work smoothly, the emphasis on built-in facilities must be on fundamental computational issues (that is, facilities that cannot elegantly and efficiently be provided by composing already existing facilities) and care must be taken not to increase reliance on mechanisms known to cause problems for the abstraction mechanisms (such as macros, uneven support for built-in types, and type violations).

---

† This is a technological variant of the proverb: ''Give a man a fish and he'll eat for a day; teach a man to fish and he'll never go hungry''.

4.1 Macros

Typical C and C++ programmers view macros very differently. The difference is so great that it can be considered philosophical. C++ programmers typically avoid macros wherever possible, preferring facilities that obey type and scope rules. In most cases, C programmers don't have such alternatives and use macros. For example, a C++ programmer might write something like this:

```
const int mx = 7;

template<class T> inline T abs(T a) { return (a<0)?-a:a; }

namespace N {
    void f(int i) { /* ... */ }
};

class X {
public:
    X(int);
    ~X();
    // ...
};
```

A C programmer facing a similar task might write something like this:

```
#define MX 7

#define abs(a) (((a)<0)?-(a):(a))

void N_f(int i) { /* ... */ }

struct X { /* ... */ };
void init_X(struct X *p, int i);
void cleanup_X(struct X *p);
```

For a variety of reasons, eliminating the use of macros to express ideas in code has been a constant aim of C++. For example, see Chapter 18 of [D&E]. This implies that a C++ programmer tend to view a solution involving a macro with suspicion, and at best as a lesser evil. On the other hand, a C programmer often view that same solution as natural, and the most elegant. Both can be right – in their respective languages – and this is a source of some misunderstanding.

At the core of many C++ programmers' distrust of macros lies the fact that macros transform the program text before tools such as compilers see it. Namespaces, class scopes, and function scopes provide no protection against a macro. Because macro substitution follow rules that doesn't involve scope and semantics, surprises can result. For example:

```
abs(i++); /* unexpected and undefined result */

struct S {
    int abs;    /* syntax error */
    /* ... */
};
```

To make such surprises less likely, most C and C++ programmers prefer to name all macros with all capital letters.

Some C programmers point out that they prefer to cope with the known problems of macros rather than to deal with the complexities of their C++ alternatives. However, this is not the place for an analysis of macros and their alternatives. The point here is that any solution to a compatibility problems that involves a macro is automatically considered suspect by many C++ programmers. Thus, any use of a macro in the standard becomes a potential incompatibility as the C++ community looks for alternative solutions to avoid its use. The only macro found in the C++ standard beyond those inherited from C is ***__cplusplus***.

5 Impact of C/C++ Feature Differences

C++ provides many features not found in C, such as virtual functions and declarations in conditions. Similarly, C99 provides several features not found in C++, such as variable length arrays (VLAs) and designated initializers. When considering compatibility issues, such features can be classified into those that affect interfaces, such as virtual functions and VLAs, and those that affect only the form of the code that they are part of, such as declarations in conditions and designated initializers.

5.1 Trivial Interfaces

C++ programmers have always known that to make code accessible to C programs they must provide interfaces that avoid non-C features, such as classes with virtual functions. These C to C++ interfaces have typically been trivial. For example:

```
// C interface:
extern int f(struct X* p, int i);

// C++ implementation of C interface:
extern "C" int f(X* p, int i) { return p->f(i); }
```

C programmers have typically assumed that any C header can be used from a C++ program. This has largely been true (after someone adds suitable ‘‘*extern "C"*’’ directives), though headers that use C++ keywords as ***struct*** member names have been a constant irritant to C++ programmers (and sometimes a serious practical problem). For example:

```
class X { /* ... */ };                // not C
struct S { int class; /* ... */ };    // not C++
```

C99 introduces several features that if used in a header will prevent that header to be used from C++ program (or from a C89 program). Examples include VLAs, ***restrict***ed pointers, ***_Bool***, ***_Complex***, some ***inline*** functions, and macros with variable number of arguments. For example:

```
// C99 interface features, not found in C++ or C89:

void f1(int[const]);                  // equivalent to f(int *const);
void f2(char p[static 8]);            // p is supposed to point to at least 8 chars
void f3(double *restrict);
void f4(char p[*]);                   // p is a VLA

inline void f5(int i) { /* ... */ }   // may or may not be C++ also

void f6(_Bool);
void f7(_Complex);

#define PRINT(form ...) fprintf(form, __VA_ARGS__)
```

If a C header uses one of those features, mediation code and a C++ header must be provided for the C code to be used from C++.

The ability to share header files is an important aspect of C and C++ culture and a key to performance of programs using both languages: C and C++ programs can call libraries implemented in ‘‘the other language’’ with no data conversion overheads and no (or very minimal) call overhead.

The question of what can and cannot be used in header files will be further confused by vendors who will provide nonstandard extensions. Many programmers have a hard time distinguishing what is standard from what is vendor-specific extension.

5.2 Thin Bindings

Where language features differ so that very similar functionality is provided in different ways, approaches based on sharing declarations is insufficient to mask the language differences. One approach to dealing with this is to provide ‘‘compatibility headers’’ that, through liberal use of ***#ifdef***s, provide very different definitions for each language but allow user code to look very similar. For example:

```
// my double precision complex
```

```
#ifdef __cplusplus
    #include<complex>
    using namespace std;
    typedef complex<double> Cmplx;
    inline Cmplx Cmplx_ctor(double r, double i) { return Cmplx(r,i); }
    //...
#else
    #include<complex.h>
    typedef double complex Cmplx;
    #define Cmplx_ctor(r,i) ((double)(r)+I*(double)(i))
    //...
#endif

void f(Cmplx z)
{
    Cmplx zz = z+Cmplx_ctor(1,2);
    Cmplx z2 = sin(zz);
    // ...
}
```

Basically, this approach is for the individual programmer or organization to create a new dialect that maps into both languages. This is an example of how a user (or a library vendor) must invent a private language simply to compensate for compatibility problems. The resulting code is typically neither good C nor good C++. In particular, by using this technique the C++ programmer is restricted to use what is easily represented in C. For examples, unless exceptional effort is expended on the C mapping, arrays must be used rather than containers, overloading (beyond what is offered by *<tgmath.h>*) must be avoided, and errors cannot be reported using exceptions. Such restrictions can be acceptable when providing interfaces to other code, but are typically too constraining for a C++ programmer to use them within the implementation. Similarly, a C programmer using this technique is prevented from using C facilities not also supported by C++, such as VLAs and *restrict*ed pointers.

Real code/libraries will have much larger ‘‘thin bindings’’ with many more macros, typedefs, inlines, etc. and more conventions for their use. The likelyhood that two such ‘‘thin bindings’’ can be used in combination is slim and the effort to learn a new such binding is non-trivial. Thus, this approach doesn’t scale and fractures the community.

5.3 Competing Programming Models

Interfaces – that is, information in header files – are all that matter to people who see C and C++ as distinct languages that just happen to be able to produce code that can be linked together (like C and Fortran). However, to programmers who use both languages, to teachers, and to implementers, compatibilities of facilities used to express computations matter. This implies that C/C++ incompatibilities in areas such as statement syntax and the meaning of expressions are also best avoided.

For users of both languages, the areas where C and C++ provide alternative solutions to similar programming problems become a problem:

[1] An alternative forces programmers to choose between two sets of facilities and their associated programming techniques.

[2] An alternative more than doubles the effort for teachers and students.

[3] Code using separate alternatives can often cooperate only through specially written mediation code.

Consider the problem of manipulating a number of objects where that number is known only at run time. C++ and C99 offer alternative solutions not present in C89. Consider a C89 example:

```
void f89(int n, int m, struct Y* v)  /* C89: v points to m Ys */
{
    struct X* p = malloc(n*sizeof(struct X)); /* not Classic C; not C++ */
    struct Y* q = malloc(m*sizeof(struct Y));
    if (p==NULL || q==NULL) exit(-1);       /* memory exhausted */
    if (3<n && 4<m) p[3] = v[4];
    memcpy(q, v, v+m*sizeof(struct Y));     /* copy */
    /* ... */
    free(q);
    free(p);
}
```

Among the potential problems with this code is that ***v*** might not point to an array with at least ***m*** elements.

The obvious C99 alternative is:

```
void f99(int n, int m, struct Y v[m])    // C99: v points to m Ys
{
    struct X p[n];                          // not C89; not C++
    struct Y q[m];
    if (3<n && 4<m) p[3] = v[4];
    memcpy(q, v, v+m*sizeof(struct Y));     // copy
    // ...
}
```

The nicer syntax makes it less likely that ***v*** does not point to an array with at least ***m*** elements, but that is still possible. Unfortunately, it is undefined what happens if the array definition fails to allocate memory for the ***n*** elements required. The use of arrays automates the freeing of memory, though there could still be a memory leak if *f99( )* is exited through a ***longjmp( )***.

The obvious C++ alternative is:

```
void fpp(int n, vector<Y>& v)  // C++: v holds v.size() Ys
{
    vector<X> p(n);             // not C89; not C99
    if (3<p.size() && 4<v.size()) p[3] = v[4];
    vector<Y> q = v;            // copy
    // ...
}
```

A ***vector*** contains the number of its elements, so the programmer doesn't have to worry about keeping track of array sizes or about freeing the memory used to hold those elements.

The standard library ***vector*** is more general than a VLA. For example, ***vector*** has a copy operation, you can change the size of a ***vector***, and ***vector*** operations are exception safe (see Appendix E of [Stroustrup,2000]). This could imply a performance overhead compared to VLAs on some implementations, but so far I have not found significant overheads.

The key point here is that users have to choose and the users of more than one of these languages have to understand the different programming styles and remember where to apply them.

6 As close as possible ...

The semi-official policy for C++ in regards to C compatibility has always been ''As Close as Possible to C, but no Closer'' [Koenig,1989]. Naturally, wits have answered with ''As Close as Possible to C++, but no Closer,'' but I have never seen that in any official context nor seen any elaboration of what it means.

How close is ''as close as possible to C?'' Traditionally, it has almost been possible to equate that statement with ''compatible with C except where the C++ type system would be compromised.'' Differences such as those for ***void****, C++'s insistence on function prototypes, the use of built-in types for ***bool*** and ***wchar_t***, and even the ***inline*** rules, can be explained that way.

The ''as close as possible ...'' rules were crafted under the assumption that ''the other language'' was immutable. In reality, it has not been so: Just look as the number of cross borrowings between C and C++. I believe that it would be technically feasible for ''as close as possible'' to be ''identical in the subset supporting traditional C-style programming'' assuming that changes could be made simultaneously to both

languages systematically bringing them closer together.

What can be done about the C/C++ incompatibilities? What should be done? I hear four basic answers:

[1] *Nothing, the incompatibilities are good for you:* I simply don't believe that, having never seen a piece of code that benefited from an incompatibility in any fundamental way. However, if enough people are of that opinion, the C and C++ committees will proceed to reduce the area of compatibility and to provide competing incompatible additions. That would destroy the C/C++ community. Programmers would increasingly face a choice between a language rich in built-in facilities and a language rich in abstraction facilities. Naturally, both language communities would be busy compensating for their weaknesses by providing libraries, which in turn would further increase the areas of incompatibility. The primary beneficiaries of this would be languages outside the C/C++ family.

[2] *Nothing, it's too late:* Given that I consider the current level of C/C++ incompatibilities both a major problem and not rooted in fundamental technical or philosophical reasons, I'm most reluctant to accept that nothing can be done. However, it is possible that changes really are infeasible today. In that case, we can strive to minimize future incompatibilities and to remove incompatibilities where opportunity arises. More likely, people will draw the conclusion that compatibility is already lost so compatibility concerns should not be allowed to complicate the design of new language features and libraries. In particular, there will be pressure for each language to provide competing, incompatible, versions of popular facilities from the other.

[3] *Remove all incompatibilities:* This is my ideal. This is what I believe to be the best long-term solution for the C/C++ community. We ought to try for that. Clearly, this would involve changes to both languages and compromises would have to be crafted to minimize the impact on users of both languages. Silent changes – that is, changes that are not easily diagnosed by a compiler – should be minimized. Wherever possible, the compromises should be crafted to increase the consistency of the resulting set of features and to simplify the language rules. It will be difficult to remove all incompatibilities. However, the amount of work required from the C/C++ community to reach compatibility will be far less that that required from it to live with increasingly incompatible languages.

[4] *Remove most of the incompatibilities; removing all is impossible:* Unfortunately, we can't always get all we want. In that case, we should figure out which incompatibilities can be removed and get rid of those. After that exercise, maybe the remaining incompatibilities won't look so impossible to remove or to live with, and maybe the exercise would discourage the growth of new incompatibilities.

Whatever is (or isn't) done must be considered in light of the fact that the world changes rapidly and that users expect programming languages to evolve to meet new challenges. Thus, compatibility issues must be considered in the wider context of language evolution. I think the most promising approach is to consider C and C++ close to complete in language support for their respective kinds of programming. If that is so, increasing C/C++ compatibility could be seen as part of a consolidation and cleanup of basic facilities. Most ''extensions'' belong in standard and non-standard libraries.

7 Acknowledgements

Thanks to Matt Austern, John Benito, Walter Brown, Don Caldwell, Brian Kernighan, Andrew Koenig, Jens Maurer Dennis Ritchie, and Tom Plum for helpful comments on drafts of this paper.

8 References

[ARM] See [Ellis,1990].

[Benito,1998] John Benito, the ISO C committee liaison to the ISO C++ committee in response to a request to document C++/C99 incompatibilities in the way C89/C++ incompatibilities are.

[Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Björn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.

[C89] ISO/IEC 9899:1990, Programming Languages – C.

[C99] ISO/IEIC 9899:1999, Programming Languages – C.

[C++98] ISO/IEC 14882, Standard for the C++ Language.

[D&E] See [Stroustrup,1994].

[Ellis,1990] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA. 1990. ISBN 0-201-51459-1.

[Kernighan,1978] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ. 1978. ISBN 0-13-110163-3.

[Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-110362-8.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. July 1989.

[Prosser,2001] Dave Prosser: Personal communication.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – the language and its compiler*. Cambridge University Press, Cambridge, England. 1980. ISBN 0-521-21965-5.

[Ritchie,1981] Dennis Ritchie. Personal communication; from memory.

[Ritchie,1988] Dennis Ritchie: *Why I do not like X3J11 type qualifiers*. Net posting. http://www.lysator.liu.se/c/dmr-on-noalias.html. January 1988.

[Simonsen,1989] K. Simonsen and B. Stroustrup: *A European Representation for ISO C*. EUUG Newsletter. Vol 9 No 2 Summer 1989.

[Stroustrup,1981] Bjarne Stroustrup: *Extensions of the C Language Type Concept*. Bell Labs internal Technical Memorandum January 5, 1981.

[Stroustrup,1982] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982.

[Stroustrup,1983] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution*. Software: Practice & Experience, Vol 13. 1983.

[Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley, Reading, MA. 1986. ISBN 0-201-12078-X.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd edition)*. Addison-Wesley, Reading, MA. 1991. ISBN 0-201-53992-6.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[Stroustrup,2000] Bjarne Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. 2000. ISBN 0-201-70073-5.

[Tribble,2001] David R. Tribble: Incompatibilities between ISO C and ISO C++. http://david.tribble.com/text/cdiffs.htm.

9 Appendix: Keywords and Compatibility Headers

When introducing a new feature, the simplest way to represent it often involves a new keyword. This was done in C++ in several cases, such as ***bool***, ***or***, and ***wchar_t***. Similarly, C99 introduced ***restrict***. The obvious snag is that every new keyword breaks some existing code. For example:

```
/* old code */

typedef int bool;

struct or {
    int restricted;
    /* ... */
};

enum { uchar_t, wchar_t };
```

Naturally, some name clashes are more likely than others, but in general there is too much code ‘‘out there’’ for a designer to reliably predict what name clashes a new keyword will cause.

An alternative approach is to avoid introducing a keyword that might clash and defining a macro or a typedef in a header instead. In places, C99 takes that approach, defining ***or*** as a macro in ***<iso646.h>*** and ***bool*** as a macro in ***<stdbool.h>***. Unfortunately, this has its own problems:

[1] The backwards compatibility which is the main reason to avoid new keywords can prove elusive.

[2] There are C++/C99 compatibility problems where the two languages took separate approaches.

To examine these problems, consider the treatment of ***bool*** in C89, C++, and C99.

9.1 No standard

C89 doesn't define ***bool***, so many people define their own. For example:

```
/* some old code */

#define bool int
#define false 0
#define true ~0

bool b = true;
```

or

```
/* other old code */

typedef char bool;
enum { false, true };

bool f(bool,bool);
```

Naturally, the lack of a standard for ***bool*** led to problems with incompatible definitions, and many (most?) library builders abandoned plain ***bool*** for schemes involving names less likely to clash. For example:

```
/* library code protecting itself against name clashes */

#define my_bool int
#define my_false 0
#define my_true ~0

my_bool b = my_true;
```

9.2 New Keyword

C++ addressed the demand for a standard Boolean type by providing a new type ***bool*** with two named values ***true*** and ***false***. Any source file using one of those three keywords as an identifier will not compile. Furthermore, a translation unit that uses an identifier ***bool*** in a way that affects external linkage will not link to translation units that has been compiled with a compiler supporting ***bool***. For example:

```
// cannot be linked to code using standard bool:

enum bool { false, true };

extern bool f(bool,bool);
```

Basically, backwards compatibility is reduced to never mixing code using a non-standard ***bool*** with (new) code that does. All violations of this rule are caught by the compiler or the linker. The introduction of a new keyword, such as ***bool***, forces a revision of source code to meet the standard.

Such revision is often resented by the programmers who have to do the work, and is occasionally deemed at least temporarily infeasible. Compatibility switches are usually provided to help users avoid, delay, or aid the transition.

There is no effective way within C++ to ''disable'' the keyword ***bool*** to accommodate code that already has its own (pre-standard) definition of ***bool***.

9.3 New Standard Header

C99 addressed the demand for providing a standard Boolean type by providing a Boolean type named ***_Bool*** and a standard header ***<stdbool.h>*** with a contents roughly equivalent to:

```
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

The obvious advantage of this approach is that translation units that do not include ***<stdbool.h>*** compile unchanged. To use the standard ***bool***, ***true***, and ***false***, ***<stdbool.h>*** must be included.

In C99, a translation unit that uses an identifier ***bool*** as an argument or return type can still link to translation units compiled with ***<stdbool.h>***. Unfortunately, the meaning of the resulting program is in general undefined. What actually happens depends on the representation of ***_Bool*** on a given implementation and the representation of the pre-standard ***bool***. A lint-like utility could catch such problems.

As a C99 programmer, I have the choice between including and not including ***<stdbool.h>***. Can I use ***bool*** as an identifier? Consider:

```
#include "foo.h"

int bool;   // OK?
```

Unless I know for certain that ***foo.h*** doesn't directly or indirectly include ***<stdbool.h>***, I must avoid defining ***bool***. Conversely, if I'm the author of ***foo.h***, I cannot safely use ***<stdbool.h>*** unless I know for certain that nothing that includes my header defines ***bool***. Therefore, authors of C99 header files must be careful to avoid including ***<stdbool.h>*** unless they know the context in which that header is used. Alternatively, the writer of a header must make the decision that users really should be able to cope with standard headers.

The former conclusion seems too Draconian, so let's explore the latter. By (possibly) including ***<stdbool.h>*** the author of ***foo.h*** presents its users with a choice:

[1] modify all code to live with the definitions in ***<stdbool.h>***, or

[2] protect all code from the definitions in ***<stdbool.h>***.

The former alternative makes the ''standard header approach'' equivalent to the ''keyword approach'' as far as code maintenance is concerned: basically all code needs to be updated to avoid uses that clash with the standard ***bool***. For example:

```
#include "foo.h"      // may or may not include <stdbool.h>

int xbool; // used to be bool
```

This is messy and – as noted for the ''keyword approach'' – such revision is resented by many programmers and is occasionally deemed at least temporarily infeasible. Let's therefore explore the possibility of protecting the code from ***<stdbool.h>***. For example:

```
#include "foo.h"

#undef bool      // protect against <stdbool.h>
#undef true
#undef false

int bool;   // OK
```

This is ugly and against the spirit of standardization, and deemed obsolescent in C99. It also involves modification of old code (which we were trying to avoid), and it is basically unmanageable because code would be littered with (repeated) protection against standard (and non-standard) macros.

A more realistic example would be:

```
#include "foo.h"

#ifndef bool
#include "my_bool.h"
#endif

bool b = true;
```

The problem here is that ***bool*** will now mean one thing in ***foo.h*** and another in the rest of the translation unit (and ***<stdbool.h>*** could of course be included again ''further down''). Also, if the ***bool*** from ***my_bool.h*** is used as an argument type or a return type of an externally linked function then the resulting program has undefined behavior if that function is linked to a function using ***<stdbool.h>***. The resulting code will work only if ***my_bool.h*** happens to be compatible with C99's ***_Bool*** or if declaration order ensures that one or the other ***bool*** is used consistently.

A slightly different problem occurs for programmers who wants to use the standard ***bool***. Consider writing this:

```
#include "foo.h"

bool b = true;   // ok?
```

Unless the programmer knows for certain that *foo.h* directly or indirectly includes *<stdbool.h>*, this code is not safe and *<stdbool.h>* must be explicitly included. For example:

```
#include "foo.h"
#ifndef bool
#include<stdbool.h>
#endif

bool b = true;   // ok
```

However, now I must worry about to possibility of *foo.h* using a non-standard *bool* in a way that could lead to inconsistencies.

I conclude that it is simpler and safer to consistently use *_Bool* directly and never include *<stdbool.h>*. For example:

```
_Bool b = 1;    // note: 1 rather than true

_Bool f(_Bool, _Bool);
```

However, this is ugly and I'm assured that this is not what was intended.

In conclusion: As the world moves on and people start using *<stdbool.h>* in new code, it is just a matter of time before

[1] the old definitions of *bool* and code using them must be removed, or

[2] the old definitions of *bool* and code using them must be protected by *#ifndef*s, or

[3] users must completely avoid *bool*, reverting to C89 practise.

For example:

```
// new code (avoid clashes between standard and non-standard uses of bool)

#define my_bool int
#define my_false 0
#define my_true ~0

my_bool b = my_true;
```

Thus, the ''header approach'' is effectively equivalent to the ''keyword approach.'' Occam's razor favors the simpler ''keyword approach,'' as used by the C committee for *restrict* and by the C++ committee for *bool*.

Please note that *bool* is just an example. The points made apply generally to the introduction of new keywords in a language descended from Classic C and to alternatives to introducing keywords.

9.4 C/C++ Compatibility Problems

Unfortunately, the C++ and C99 committees chose different approaches for *bool*. This causes problems for people who write headers for use in both languages.

For example, this compatibility header was suggested for use with C and C++:

```
#if defined(__cplusplus)
    typedef bool _Bool; // C++ has bool, true, and false, but not _Bool
#elif defined __bool_true_false_are_defined
    /*
        either this is C99, or other headers have provided the
        macros and the feature-test flag
    */
#else
    /* provide a C99 compatible bool */
    typedef int bool;
    typedef int _Bool;
    #define false 0
    #define true 1
    #define __bool_true_false_are_defined
#endif
```

This may be sufficient in many programs. However, the problems encountered for ***<stdbool.h>*** will be seen for any ‘‘compatibility header’’ that we might devise, including this one. In addition, this header assumes that C99’s ***_Bool*** and C++’s ***bool*** are represented as ***int***s. This is not guaranteed and can therefore lead to subtle linkage and portability problems. Defining ***_Bool*** in a C++ program may also cause problems because that name is reserved to the implementation (which may, for example, have defined ***_Bool*** to ease C++/C99 compatibility).

Finally, these issues and headers all seems rather complex for ***bool***. After all, ***bool*** ought to be the simplest possible data type.

Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software

Damian Dechev
dechev@tamu.edu
Texas A&M University
College Station, TX 77843-3112

Bjarne Stroustrup
bs@cs.tamu.edu
Texas A&M University
College Station, TX 77843-3112

Abstract

The high degree of autonomy and increased complexity of future robotic spacecraft pose significant challenges in assuring their reliability and efficiency. To achieve fast and safe concurrent interactions in mission critical code, we survey the practical state-of-the-art nonblocking programming techniques. We study in detail two nonblocking approaches: (1) CAS-based algorithms and (2) Software Transactional Memory. We evaluate the strengths and weaknesses of each approach by applying each methodology for engineering the design and implementation of a nonblocking shared vector. Our study investigates how the application of nonblocking synchronization can help eliminate the problems of deadlock, livelock, and priority inversion and at the same time deliver a performance improvement in embedded real-time software.

1 Objectives

[1] Future space exploration projects, such as Mars Science Laboratory (MSL) [23], demand the engineering of some of the most complex embedded software systems. The notion of concurrency is of critical importance for the design and implementation of such systems. The process-oriented software development and certification protocols (such as [20]) often do not reach the level of detail of providing guidelines for the engineering of reliable concurrent software. In this work, we present a detailed survey of the state-of-the-art *nonblocking* programming techniques that can help in implementing *efficient* and *safe* concurrent interactions in mission critical embedded code.

[1] This is the authors' version of the work. It is posted here by permission of the publisher. Not for redistribution. The definitive version is published in Proceedings of 3rd IEEE International Conference on Space Mission Challenges for Information Technology (IEEE SMC-IT 2009), Pasadena, California, July 2009.

1.1 Parallelism and Complexity

The most common technique for controlling the interactions of concurrent processes is the use of mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads trying to access it except the one holding the lock. In scenarios of high contention on the shared data, such an approach can seriously affect the performance of the system and significantly diminish its parallelism. For the majority of applications, the problem with locks is one of difficulty of providing *correctness* more than one of performance. The application of mutually exclusive locks poses significant safety hazards and incurs high complexity in the testing and validation of mission-critical software. Locks can be optimized in some scenarios by utilizing fine-grained locks or context-switching. Often due to the resource limitations of flight-qualified hardware, optimized lock mechanisms are not a desirable alternative [18]. Even for efficient locks, the interdependence of processes implied by the use of mutual exclusion introduces the dangers of *deadlock*, *livelock*, and *priority inversion*. The incorrect application of locks is hard to determine with the traditional testing procedures and a program can be deployed and used for a long period of time before the flaws become evident and eventually cause anomalous behavior.

1.2 Nonblocking Synchronization

To achieve higher safety and gain performance, we suggest the application of *nonblocking synchronization*. A concurrent object is *nonblocking* if it guarantees that *some* process in the system will make progress in a *finite* amount of steps [14]. An object that guarantees that *each* process will make progress in a *finite* number of steps is defined as *wait-free*. *Obstruction-freedom* [13] is an alternative nonblocking condition that ensures progress if a thread eventually executes in *isolation*. It is the weakest nonblocking property and obstruction-free objects require the support of

a contention manager to prevent livelocking. Nonblocking designs most commonly are founded on a set of atomic primitives supported by the hardware architecture.

The most ubiquitous and versatile data structure in the ISO C++ Standard Template Library [22] is *vector*, offering a combination of dynamic memory management and constant-time random access. Because of the vector's wide use and challenging parallel implementation of its nonblocking dynamic operations, we illustrate the efficiency of each nonblocking approach discussed in this work with respect to its applicability for the design and implementation of a shared nonblocking vector. A number of pivotal concurrent applications in the Mission Data System [15] framework employ a shared STL vector (in all scenarios protected by mutually exclusive locks). Such is the Data Management Service library described by Wagner in [24].

2 Engineering a Nonblocking Data Structure

Lock-free and wait-free algorithms exploit a set of portable atomic primitives such as the word-size Compare-and-Swap (CAS) instruction [9]. The design of nonblocking data structures poses significant challenges and their development and optimization is a current topic of research [8], [14]. The Compare-And-Swap (CAS) atomic primitive (commonly known as Compare and Exchange, *CMPXCHG*, on the Intel *x86* and *Itanium* architectures [16]) is a CPU instruction that allows a processor to atomically test and modify a single-word memory location. CAS requires three arguments: a memory location (L_i), an old value (A_i), and a new value (B_i). The instruction atomically exchanges the value stored at L_i with B_i, provided that L_i's current value equals A_i. The result indicates whether the exchange was performed. For the majority of implementations the return value is the value last read from L_i (that is B_i if the exchange succeeded). Some CAS variants, often called Compare-And-Set, have a return value of type boolean. The hardware architecture ensures the atomicity of the operation by applying a fine-grained hardware lock such as a cache or a bus lock (as is the case for IA-32 [16]). The application of a CAS-controlled speculative manipulation of a shared location (L_i) is a fundamental programming technique in the engineering of nonblocking algorithms [14] (an example is shown in Algorithm 1). In our pseudocode we use

Algorithm 1 CAS-controlled speculative manipulation of L_i

```
1: repeat
2:     value_type A_i = L_i^
3:     value_type B_i = fComputeB
4: until CAS(L_i, A_i, B_i) == Bi
```

the symbols ^, &, and . to indicate pointer dereferencing, obtaining an object's address, and integrated pointer dereferencing and field access. When the value stored at L_i is the control value of a CAS-based speculative manipulation, we call L_i and L_i^ *control location* and *control value*, respectively. We indicate the control value's type with the string $value_type$. The size of $value_type$ must be equal or less than the maximum number of bits that a hardware CAS instruction can exchange atomically (typically the size of a single memory word). In the most common cases, $value_type$ is either an integer or a pointer value. In the latter case, the implementor might reserve two extra bits per each control value and use them for implementation-specific value marking [8]. This is possible if we assume that the pointer values stored at L_i are aligned and the two low-order bits have been cleared. In Algorithm 1, the function $fComputeB$ yields the new value B_i.

Linearizability [14] is an important correctness condition for concurrent objects: a concurrent operation is linearizable if it appears to execute instantaneously in a given point of time between the time τ_1 of its invocation and the time τ_2 of its completion. The implementations of many nonblocking data structures require the update of two or more memory locations in a linearizable fashion [3], [8]. The engineering of such operations (e.g. *push_back* and *resize* in a shared dynamically resizable array) is critical and particularly challenging in a CAS-based design. Harris et al. propose in [10] a software implementation of a multiple-compare-and-swap (MCAS) algorithm based on CAS. This software-based MCAS algorithm has been applied by Fraser in the implementation of a number of lock-free containers such as binary search trees and skip lists [7]. The cost of the MCAS operation is expensive requiring $2M + 1$ CAS instructions. Consequently, the direct application of the MCAS scheme is not an optimal approach for the design of lock-free algorithms. However, the MCAS implementation employs a number of techniques (such as pointer bit marking and the use of Descriptors) that are useful for the design of practical lock-free systems. A common programming technique applied for the implementation of the complex nonblocking operations is the use of a *Descriptor Object* [3], [8]. A Descriptor is an object that allows an interrupting thread help an interrupted thread complete successfully.

A number of advanced Software Transactional Memory (STM) libraries provide nonblocking transactions with dynamic linearizable operations [5], [21]. Such transactions can be utilized for the design of nonblocking containers [21]. As our performance evaluation demonstrates, the high cost of the extra level of indirection and the conflict detection and validation schemes in STM systems does not allow performance comparable to that of a hand-crafted lock-free container that relies solely on the application of portable atomic primitives. Sections 2.3 and 2.5 describe in detail

the implementation of a nonblocking shared vector using CAS-based techniques and STM, respectively. Section 3 provides analysis of the suggested implementation strategies and discusses the performance evaluation of the two approaches.

2.1 Design Goals

In this section we synthesize the most desirable characteristics of a nonblocking shared vector.

a. thread-safety: the data should be accessible to multiple processors at all times

b. lock-freedom: guarantee (at least) lock-free progress of the container's operations

c. portability: do not rely on uncommon architecture-specific instructions

d. easy-to-use interfaces: offer the interfaces, functionality, and guarantees available in the sequential STL vector

e. high level of parallelism: concurrent completion of non-conflicting operations should be possible

f. minimal overhead: achieve lock-freedom without excessive copying, levels of indirection, and costly conflict detection and validation schemes, minimize the time spent on redundant and speculative computations and the number of calls to costly atomic primitives

2.2 Implementation Concerns

We provide a brief summary of the most important implementation concerns for the practical and portable design of a nonblocking dynamic array. The following sections discuss the implementation issues related to guaranteeing portability, meeting the requirements for linearizability, preventing race conditions, coping with the ABA problem, and incorporating nonblocking memory management and allocation schemes.

2.2.1 Portability

Virtually at the core of every known synchronization technique is the application of a number of hardware atomic primitives. The semantics of such primitives vary depending on the specific hardware platform. There are a number of architectures that support some hardware atomic instructions that can provide greater flexibility such as the Load-Link/Store Conditional (LL/SC) supported by the PowerPC, Alpha, MIPS, and the ARM architectures or instructions that perform atomic writes to more than a single word in memory, such as the double-compare-and-swap instruction (DCAS) [4]. The hardware support for such atomic instructions can vastly simplify the design of a nonblocking algorithm as well as offer immediate solutions to a number of challenging problems such as the ABA problem [19]. However, to maintain portability across a large number of hardware platforms, the design and implementation of a nonblocking algorithm cannot rely on the support of such atomic primitives. The most common atomic primitive that is supported by a large majority of hardware platforms is the single-word CAS instruction.

2.2.2 Linearizability Guarantee

In a CAS-based design, a major difficulty is meeting the linearizability requirements for operations that require the update of more than a single-word in the system's shared memory. To cope with this problem, it is possible to apply a combination of a number of known techniques:

a. *Extra Level of Indirection:* Reference semantics [22] must be assumed in case the data being manipulated is larger than a memory word size or the approach relies on the application of smart pointers or garbage collection for each individual element in the shared container

b. *Barnes-style announcement [3]:* Often referred to as a *Descriptor Object*, a Barnes-style announcement stores a description of a pending operation on a given memory location. It allows the interrupting threads help the interrupted thread complete an operation rather than wait for its completion

c. *Descriptive Log:* At the core of virtually all Software Transactional Memory implementations, the Descriptive Log stores a description of all pending reads and writes to the shared data. It is used for conflict detection, validation, and optimistic speculation

d. *Transactional Memory:* A duplicate memory copy used to perform speculative updates that are invisible to all other threads until the linearization point of the entire transaction

e. *Optimisitic Speculation:* Complex nonblocking operations often employ optimistic speculative execution in order to carry out the memory updates on a local or duplicate memory copy and commit once there are no conflicts with interfering operations. It is necessary to employ a methodology for unrolling all changes performed by the speculating operation, should there be conflicts during the commit phase

To illustrate the complexity of a nonblocking design of a shared vector, Table 1 provides an analysis of the number of memory locations that need to be updated upon the execution of some of its basic operations.

2.2.3 Interfaces of the Concurrent Operations

According to the ISO C++ Standard [17], the STL containers' interfaces are inherently sequential. The next ISO C++ Standard [1] is going to include a concurrent memory model [2] and possibly a blocking threading library. In Table 2 we

	Operations	Memory Locations
push_back	$Vector \times Elem \rightarrow void$	*2: element, size*
pop_back	$Vector \rightarrow Elem$	*1: size*
reserve	$Vector \times size_t \rightarrow Vector$	*n: all elements*
read	$Vector \times size_t \rightarrow Elem$	*none*
write	$Vector \times size_t \times Elem \rightarrow Vector$	*1: element*
size	$Vector \rightarrow size_t$	*none*

Table 1. Vector - Operations

Operation	Description
size_type $size()$ const	Number of elements in the vector
size_type $capacity()$ const	Number of available memory slots
void $reserve(size_type\ n)$	Allocation of memory with capacity n
bool $empty()$ const	true when $size = 0$
T* $operator[]$ $(size_type\ n)$ const	returns the element at position n
T* $front()$	returns the first element
T* $back()$	returns the last element
void $push_back(constT\&)$	inserts a new element at the tail
void $pop_back()$	removes the element at the tail
void $resize(n, t = T())$	modifies the tail, making $size = n$

Table 2. Interfaces of STL Vector

show a brief overview of some of the basic operations of an STL vector. Consider the sequence of operations applied to an instance, vec, of the STL vector: *vec[vec.size()-1]; vec.pop_back();*. In an environment with *concurrent* operations, we cannot have the guarantee that the element being deleted by the *pop_back* is going to be the element that had been read earlier by the invocation of *operator[]*. Such a *sequential* history is just one of the several legal sequential histories that can be derived from the concurrent execution of the above operations. While the STL interfaces have proven to be efficient and flexible for a large number of applications [22], to preserve the semantic behavior implied by the sequential definition of the STL semantics, one can either rely on a library with atomic transactions [5], [21] or alternatively define concurrent STL interfaces adequate with respect to the applied consistency model. In the example we have shown, it might be appropriate to modify the interface of the *pop_back* operation and return the element being deleted instead of the *void* return type specified in STL. Such an implementation efficiently combines two operations: reading the element to be removed from the container and removing the element. Should we prefer to keep the STL standard interface of $void$ *pop_back()*, the task of obtaining the value of the removed element in a concurrent nonblocking execution might be quite costly and difficult to implement. Based on the shared containers' usage, observing the possibilities for such combinations can deliver better usability and performance advantages in a nonblocking implementation. Other possibly beneficial combinations of operations are 1) CAS-based *read-modify-write* at location L_i that unifies a random access read and write at location L_i and 2) the $push_back$ of a block of tail elements.

2.3 Overview of the Lock-free Operations

In this section we present a brief overview of the most critical lock-free algorithms employed by a CAS-based shared vector (see [3] for the full set of the operations of the first lock-free dynamically resizable array). To help tail operations update the size and the tail of the vector (in a linearizable manner), the design presented in [3] suggests the application of of a helper object, named "`Write Descriptor (WD)`" that announces a pending tail modifications and allows interrupting threads help the interrupted thread complete its operations. A pointer to the WD object is stored in the "`Descriptor`" together with the container's size and a reference counter required by the applied memory management scheme. The approach avoids storage relocation and its synchronization hazards by utilizing a two-level array. Whenever `push_back` exceeds the current capacity, a new memory block twice the size of the previous one is added. The remaining part of this section presents the pseudo-code of the tail operations (*push_back* and *pop_back*) and the random access operations (*read* and *write* at a given location within the vector's bounds).

Algorithm 2 $push_back\ vector, elem$

```
1: repeat
2:     desc_current ← vector.desc
3:     CompleteWrite(vector, desc_current.pending)
4:     if vector.memory[bucket] = NULL then
5:         AllocBucket(vector, bucket)
6:     end if
7:     wop ←
       new WriteDesc(At(desc_current.size)^, elem, desc_current.size)
8:     desc_next ← new Descriptor(desc_current.size + 1, wop)
9: until CAS(&vector.desc, desc_current, desc_next)
10: CompleteWrite(vector, desc_next.pending)
```

Algorithm 3 Read $vector, i$

```
1: return At(vector, i)^
```

Algorithm 4 Write $vector, i, elem$

```
1: At(vector, i)^ ← elem
```

Algorithm 5 $pop_back\ vector$

```
1: repeat
2:     desc_current ← vector.desc
3:     CompleteWrite(vector, desc_current.pending)
4:     elem ← At(vector, desc_current.size − 1)^
5:     desc_next ← new Descriptor(desc_current.size − 1, NULL)
6: until CAS(&vector.desc, desc_current, desc_next)
7: return elem
```

Algorithm 6 CompleteWrite $vector, wop$

```
1: if wop.pending then
2:     CAS(At(vector, wop.pos), wop.value_old, wop.value_new)
3:     wop.pending ← false
4: end if
```

Push_back (add one element to end) The first step is to complete a pending operation that the current descriptor might hold. In case that the storage capacity has reached its limit, new memory is allocated for the next memory bucket. Then, `push_back` defines a new "`Descriptor`" object and announces the current write operation. Finally, `push_back` uses CAS to swap the previous "`Descriptor`" object with the new one. Should CAS fail, the routine is re-executed. After succeeding, `push_back` finishes by writing the element.

Pop_back (remove one element from end) Unlike `push_back`, `pop_back` does not utilize a "`Write Descriptor`". It completes any pending operation of the current descriptor, reads the last element, defines a new descriptor, and attempts a CAS on the descriptor object.

Non-bound checking Read and Write at position i The random access `read` and `write` do not utilize the descriptor and their success is independent of the descriptor's value.

2.3.1 The ABA Problem

The ABA problem [19] is fundamental to all CAS-based systems. The ABA problem can occur in the CAS-based design of a nonblocking dynamic array in a number of possible ways. One possible hazardous execution can happen like this: assume a thread T_0 attempts to perform a `push_back`; in the vector's "`Descriptor`", `push_back` stores an announcement declaring that the value of the object at position i should be changed from A to B. Then a thread T_1 interrupts and reads the *Descriptor Object*. Later, after T_0 resumes and successfully completes the operation, a third thread T_2 can modify the value at position i from B back to A. When T_1 resumes its CAS is going to succeed and erroneously execute the update from A to B. As a common technique for overcoming the ABA problem it has been suggested to use a version tag attached to each value. Such an approach demands the application of an atomic instruction such as a CAS2 (compare-and-swap two co-located words), a hardware primitive that is available on some modern Intel architectures. For our nonblocking implementation we cannot assume the availability of such atomic primitives since they are specific to a limited number of hardware platforms. ABA avoidance on CAS-based architectures has been typically limited to two possible approaches:

a. split a 32-bit memory word into a value and a counter portions (thus significantly limiting the usable address space or the range of values that can be stored) [6]

b. apply value semantics (by utilizing an extra level of indirection, i.e. create a unique pointer to each value to be stored) in combination with a memory management approach that disallows the reuse of potentially hazardous memory locations [12], [19] (thus impose a significant performance overhead)

To eliminate the ABA problem of (2), the authors in [3] suggest the application of a memory management scheme such as Herlihy et al.'s *Pass The Buck* algorithm [11] that utilizes a separate thread to periodically reclaim unguarded objects. The vector's vulnerability to (1) (in the absence of CAS2 or LL/SC), can be eliminated by requiring the data structure to copy all elements and store pointers to them.

2.4 STM-based Nonblocking Design

A variety of recent STM approaches [5], [21] claim safe and easy to use concurrent interfaces. The most advanced STM implementations allow the definition of efficient "large-scale" transactions, i.e. *dynamic* and *unbounded* transactions. *Dynamic transactions* are able to access memory locations that are not statically known. *Unbounded transactions* pose no limits on the number of locations being accessed. The basic techniques applied are the utilization of public records of concurrent operations and a number of conflict detection and validation algorithms that prevent side-effects and race conditions. To guarantee progress transactions help those ahead of them by examining the public log record. The availability of nonblocking, unbounded, and dynamic transactions provides an alternative to CAS-based designs for the implementation of nonblocking data structures. The complex designs of such advanced STMs often come with an associated cost:

a. *Two Levels of Indirection:* A large number of the nonblocking designs require two levels of indirection in accessing data

b. *Linearizability:* The linearizability requirements are hard to meet for an unbounded and dynamic STM. To achieve efficiency and reduce the complexity, many STMs offer the less demanding *obstruction-free* synchronization [13]

c. *STM-oriented Programming Model:* The use of STM requires the developer to be aware of the STM implementation and apply an STM-oriented Programming Model. The effectiveness of such programming models is a topic of current discussions in the research community

d. *Closed Memory Usage:* Both nonblocking and lock-based STMs often require a *closed memory system*

e. *Vulnerability of Large Transactions:* In a nonblocking implementation large transactions are a subject to interference from contending threads and are more likely to encounter conflicts. Large blocking transactions can be subject to time-outs, requests to abort or introduce a bottleneck for the computation

f. *Validation:* A validation scheme is an algorithm that ensures that none of the transactional code produces side-effects. Code containing I/O and exceptions needs to be reworked as well as

some class methods might require special attention. Consider a class hierarchy with a base class *A* and two derived classes *B* and *C*. Assume *B* and *C* inherit a virtual method *f* and B's implementation is side-effect free while C's is not. A validation scheme needs to disallow a call to C's method *f*

With respect to our design goals, the main problems associated with the application of STM are meeting the stricter requirements posed by the lock-free progress and safety guarantees and the overhead introduced by the application of an extra level of indirection and the costly conflict detection and validation schemes.

2.5 RSTM-based Vector

The Rochester Software Transactional Memory (RSTM) [21] is a word- and indirection-based C++ STM library that offers obstruction-free nonblocking transactions. As explained by the authors in [21], while helping provide lightweight committing and aborting of transactions, the extra level of indirection can cause a dramatic performance degradation due to the more frequent capacity and coherence misses in the cache. In this section we employ the RSTM library (version 4) to build an STM-based nonblocking shared vector. In Algorithms 7, 8, 9, and 10, we present the RSTM-based implementation of the $read$, $write$, pop_back, and $push_back$ operations, respectively. According to the RSTM API [21], access to shared data is achieved by utilizing four classes of shared pointers: 1) a *shared object* (class $sh_ptr < T >$) representing on object that is untouched by a transaction, 2) a *read only object* (class $rd_ptr < T >$) referring to an object that has been opened for reading, 3) a *writable object* (class $wr_ptr < T >$) pointing to a an object opened for writing by a transaction, and 4) a *privatized object* (class $un_ptr < T >$) representing an object that can be accessed by one thread at a time. These smart pointer templates can be instantiated only with data types derived from a core RSTM object class $stm :: Object$. Thus, we need to wrap each element stored in the shared vector in a class $STMVectorNode$ that derives from $stm :: Object$. Similarly, we define a Descriptor class $STMVectorDesc$ (derived from $stm :: Object$) that stores the container-specific data such as the vector's size and capacity. The tail operations need to modify (within a single transaction) the last element and the Descriptor object (of type $STMVectorDesc$) that is stored in a location L_{desc}. The vector's memory array is named with the string mem. In the pseudo-code in Algorithms 9 and 10 we omit the details related to the management of mem (such as the resizing of the shared vector should the requested size exceed the container's capacity).

Algorithm 7 RSTM vector, operation $read$ location p

```
1: BEGIN_TRANSACTION
2: rd_ptr<STMVectorNode> rp(mem[p])
3: result = rp->value
4: END_TRANSACTION
5: return result
```

Algorithm 8 RSTM vector, operation $write\ v$ at location p

```
1: BEGIN_TRANSACTION
2: wr_ptr<STMVectorNode> wp(mem[p])
3: wp->val = v
4: sh_ptr<STMVectorNode> nv =
   new sh_ptr<STMVectorNode>(wp)
5: mem[p] = nv
6: END_TRANSACTION
```

Algorithm 9 RSTM vector, operation pop_back

```
1: BEGIN_TRANSACTION
2: rd_ptr<STMVectorNode> rp(mem[L_desc->size - 1])
3: sh_ptr<STMVectorDesc> desc =
   new sh_ptr<STMVectorDesc>
   (new STMVectorDesc(L_desc->size - 1))
4: result = rp->value
5: L_desc = desc
6: END_TRANSACTION
7: return result
```

Algorithm 10 RSTM vector, operation $push_back\ v$

```
1: BEGIN_TRANSACTION
2: sh_ptr<STMVectorNode> nv =
   new sh_ptr<STMVectorNode>(new STMVectorNode(v))
3: sh_ptr<STMVectorDesc> desc =
   new sh_ptr<STMVectorDesc>
   (new STMVectorDesc(L_desc->size + 1))
4: mem[size] = nv
5: L_desc = desc
6: END_TRANSACTION
```

3 Analysis and Results

To evaluate the performance of the discussed synchronization techniques, in this section we analyze the performance of three approaches for the implementation of a shared vector:

(1) the RSTM-based nonblocking vector implementation as presented in Section 2.5

(2) an RSTM lock-based execution of the vector's transactions. RSTM provides an option of running the transactional code in a lock-based mode using redo locks [21]. Though blocking and not meeting our goals for safe and reliable synchronization, we include the lock-based RSTM vector execution to gain additional insight about the relative performance gains or penalties that the discussed nonblocking approaches offer when compared to the execution of a lock-based, STM-based container

(3) the hand-crafted CAS-based algorithms design as presented in Section 2.3

We ran performance tests on an Intel IA-32 SMP machine with two 1.83GHz processor cores with 512 MB shared memory and 2 MB L2 shared cache running the MAC 10.5.6 operating system. We designed our experiments by generating a workload of the various operations. We

Figure 1. Performance Results A

Figure 2. Performance Results B

varied the number of threads, starting from 1 and exponentially increased their number to 32. Each thread executed 500,000 lock-free operations on the shared container. We measured the execution time (in seconds) that all threads needed to complete. Each iteration of every thread executed an operation with a certain probability (`push_back` (+), `pop_back` (−), random access `write` (`w`), random access `read` (`r`)). We show the performance graph for a distribution of +:10%, −:10%, `w`:40%, `r`:40% on Figure 1. Figure 2 demonstrates the performance results in a read-many-write-rarely scenario, +:10%, −:10%, `w`:10%, `r`:70%. Figure 3 illustrates the test results with a distribution +:25%, −:25%, `w`:12%, `r`:38%. The number of threads is plotted along the x-axis, while the time needed to complete all operations is shown along the y-axis. To increase the readability of the performance graphs, the y-axis uses a logarithmic scale with a base of 10. Our test results indicate that for the large majority of scenarios the hand-crafted CAS-based approach outperforms by a significant factor the transactional memory approaches. The approach from [3] offers simple application and fast execution. The STM-based design offers a flexible programming interface and easy to comprehend concurrent semantics. The main deterrent associated with the application of STM is the overhead introduced by the extra level of indirection and the application of costly conflict detection and validation schemes. According to our performance evaluation, the nonblocking RSTM vector demonstrates poor scalability and its performance progressively deteriorates with the increased volume of operations and active threads in the system. In addition, RSTM transactions offer obstruction-free semantics. To eliminate the hazards of livelocking, the software designers need to integrate a contention manager with the use of an STM-based container. Because of the limitations present in the state of the art STM libraries [21], [5], we suggest that a shared vector design based on the utilization of nonblocking CAS-based algorithms can better serve the demands for safe and reliable concurrent synchronization in mission critical code.

Figure 3. Performance Results C

4 Impact for Space Systems

Modern robotic space exploration missions, such as the Mars Science Laboratory [23], are expected to embed a large array of advanced components and functionalities and perform a complex set of scientific experiments. The high degree of autonomy and increased complexity of such systems pose significant challenges in assuring the reliability and efficiency of their software. A survey on the challenges for the development of modern spacecraft software

by Lowry [18] reveals that in July 1997 The Mars Pathfinder mission experienced a number of anomalous system resets that caused an operational delay and loss of scientific data. The follow-up analysis identified the presence of a priority inversion problem caused by the low-priority meteorological process blocking the the high-priority bus management process. The software engineers found out that it would have been impossible to detect the problem with the black box testing applied at the time. A more appropriate priority inversion inheritance algorithm had been ignored due to its frequency of execution, the real-time requirements imposed, and its high cost incurred on the slower flight-qualified computer hardware. The subtle interactions in the concurrent applications of the modern aerospace autonomous software are of critical importance to the system's safety and operation. The presence of a large number of concurrent autonomous processes implies an increased volume of interactions that are hard to predict and validate. Allowing fast and reliable concurrent synchronization is of critical importance to the design of autonomous spacecraft software.

5 Conclusion

In this work we investigated how the application of nonblocking synchronization can help eliminate the problems of deadlock, livelock, and priority inversion in embedded real-time mission critical software. We studied the challenging process of how to design and implement a nonblocking data container by applying 1) CAS-based synchronization and 2) Software Transactional Memory. We discussed the principles of nonblocking synchronization and demonstrated the application of both approaches by showing the implementation of a lock-free shared vector. Our performance evaluation concluded that while difficult to design, CAS-based algorithms offer fast and scalable performance and in a large majority of scenarios outperform the alternative STM-based nonblocking or lock-based approaches by a significant factor. This paper aimed at delivering better understanding of the advantages (over mutual exclusion) as well as the usability and performance tradeoffs of the modern nonblocking programming techniques that can be of critical importance for the engineering of reliable and efficient concurrent flight software.

References

[1] P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21 N2009, April 2006.

[2] H. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI '08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation*. ACM Press, 2008.

[3] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-Free Dynamically Resizable Arrays. In A. A. Shvartsman, editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2006.

[4] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. In *International Symposium on Distributed Computing*, pages 59–73, 2000.

[5] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the 2007 International Symposium on Code Generation and Optimization (CGO)*, 2007.

[6] D. Dvorak and W. Reinholtz. Hard real-time: C++ versus RTSJ. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 268–274, New York, NY, USA, 2004. ACM.

[7] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[8] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[9] D. Gifford and A. Spector. Case study: IBM's system/360-370 architecture. *Commun. ACM*, 30(4):291–307, 1987.

[10] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[11] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[12] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 339–353, London, UK, 2002. Springer-Verlag.

[13] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, DC, USA, 2003. IEEE Computer Society.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[15] M. Ingham, R. Rasmussen, M. Bennett, and A. Moncada. Engineering Complex Embedded Systems with State Analysis and the Mission Data System. In *In Proceedings of First AIAA Intelligent Systems Technical Conference 2004*, 2004.

[16] Intel. Ia-32 intel architecture software developer's manual, volume 3: System programming guide, 2004.

[17] ISO/IEC 14882 International Standard. *Programming languages C++*. American National Standards Institute, September 1998.

[18] M. R. Lowry. Software Construction and Analysis Tools for Future Space Missions. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2002.

[19] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[20] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[21] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update, http://www.cs.rochester.edu/research/synchronization/rstm/v4api.shtml. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 210–220, New York, NY, USA, 2007. ACM.

[22] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[23] R. Volpe and S. Peters. Rover Technology Development and Mission Infusion for the 2009 Mars Science Laboratory Mission. In *7th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, May 2003.

[24] D. Wagner. Data Management in the Mission Data System. In *Proceedings of the IEEE System, Man, and Cybernetics Conference*, 2005.

FOREWORD: Serving the C++ Community

The ISO standard is the culmination of the early years of C++ and the basis for its further evolution. This foreword examines the role of the standard committee in the life of C++ and ponders the future of Standard C++. C++ is a general-purpose programming language with a bias toward systems programming. By design and necessity it is a language that can — and often does — provide run-time performance unmatched by anything but carefully handcrafted assembly code. Similarly, by design and necessity it is a language that is available on a wide variety of platforms from some of the tiniest processors to some of the most ambitious supercomputers. In addition, C++'s abstraction mechanisms — such as classes and templates — effectively serve users for whom high performance is merely a fringe benefit. The result has been an extraordinary diversity of application areas and user communities.

This widespread and diverse use implies centrifugal forces that could tear the language apart into competing dialects, thus destroying the portability and shared community that are among the most valuable benefits of a common general-purpose language. This danger is made more acute because C++ lacks conventional centralizing forces such as a corporation with legal ownership (and a deep war chest for marketing and development), a dominant implementation, or a centralized users' organization.

The C++ community's response to this challenge was to establish the ISO C++ standards committee (and its national counterparts) and to support its work for over a decade. The honor of taking the initiative goes to people from DEC, HP, and IBM, who — led by Larry Rosler and Dmitry Lenkov — in late 1988 contacted me about starting ANSI standardization of C++. It took them a couple of hours to convince me, but once convinced I started to prepare. The ARM (''The Annotated C++ Reference Manual'') was the first and most visible result. The ARM, or rather the reference manual in ''The C++ Programming Language (2nd edition),'' became the base document for the ANSI standardization of C++. AT&T deserves special praise for contributing this reference manual to the standards effort. A less enlightened company might have tried to retain C++ as a proprietary language.

The first and most significant effect of forming the ANSI C++ group was as obvious as it was unexpected to people unfamiliar with standardization. Suddenly, the community had an open forum where people from different parts of the community could meet to discuss all topics of importance to the language and its users. Commercial allies could talk without falling foul of cartel rules, commercial competitors could settle technical issues without getting into trouble with corporate rules, and there now was a place and time for people to meet face to face. In the committee, I finally met people I had known for years via email.

The C++ standardization always involved people from many countries. Even the ARM was reviewed by more than 100 people from half a dozen countries, and the first organizational meeting of what was to become the ANSI C++ committee (in Washington D.C. in December of 1989) had participants from several countries. In 1991, this international aspect of C++ was formalized by making the standardization an ISO process.

In retrospect, the committee's role as an open forum and as a focus of the C++ community was at least as important as the final standard. The C++ standards committee is the center of the C++ community. Had the members not left their commercial rivalries at the committee door and focused on technical issues, the C++ standards effort would have failed. I am told that this consistently civil, civic, and often friendly behavior is uncommon in a standards committee. One of the things that have made the C++ standards process worth while for me personally and professionally is that only in the rarest of cases have members interpreted their obligations narrowly or destructively. I cannot think of a meeting where I didn't learn something worthwhile. The C++ committee has been the source of many long-term friendships.

The committee membership was always large enough and dedicated enough to keep alive communication with the community at large. The image of a handful of experts meeting in some ''isolated, small, dark, smoke-filled room'' is about as far from the reality of the C++ committee as you can get. I once

From The Wiley publication of the C++ Standard(ISO/IEC 14882:2002). ISBN 0 470 84674-7.

counted 102 people in the meeting room. There are rarely less than 50 people attending a meeting and something like 200 people participate between meetings using email and websites. I have seen people providing real-time reports to newsgroups from the meeting room. In addition, several nations have their own groups and meetings. From this extended group of members, information flows to and from the rest of the C++ community through email, newsgroups, presentations, papers, and books. This open, two-way, communication has been essential for the development of C++.

The result of the committee's work so far and the basis of all future work on C++ is the current standard. Code written against this standard will work for years and even decades to come. The virtues of the standard document are precision and detail. For most readers, its problems are that it is hard to read, contains no rationale, and gives only few hints at implementation techniques. The standard is not a tutorial, not even a tutorial for experts. There are textbooks covering that need, and people who try to learn C++ from the standard usually have a terrible time figuring out the language features and — like anyone trying to learn C++ from a manual — fail to appreciate the programming techniques those features exist to support. The closest thing we have to a rationale for C++ is my ''The Design and Evolution of C++'' which discusses the motivation behind most of the major design decisions.

A standard is just a heap of paper (or a bag of bits) until someone takes it seriously enough to provide implementations that approximates it as closely as can be done given real-world constraints. The C++ community is now fortunate enough to have several high-quality implementations that closely approximate the standard and at least one that passes all conformance tests with flying colors. These implementations and their numerous ports cover more platforms than I could possibly know of. The community owes thanks to the compiler and standard library writers — many of whom are members of the committee or represented there by colleagues. Standard C++ is not an easy language to implement, yet for a given platform there are often a choice of good compilers and standard library implementations. One implication of this wide availability is a high degree of practical portability (if that's your aim) across an astounding range of systems.

The committee was never just backwards looking. Naturally, much of its work is concerned with the detailed, tedious, and necessary work of clearly stating existing rules and resolving compatibility problems. However, a living language cannot be static and unchanging, so C++ has evolved to meet challenges. For example, in my 1990 C++ reference manual, exceptions and templates were labeled ''experimental,'' yet they were obviously essential parts of what C++ was supposed to be. Today, through the work of the committee, the efforts of implementers, and the creative ideas of users, templates and exceptions are the key to some of the most powerful and innovative uses of C++.

Library development has always been one of the most vital driving forces of C++. One of my earliest principles for C++ was to prefer facilities that ease or enable library building over facilities that merely solve specific problems. This idea was sorely tested by the lack of a good standard container library. C++ was expressive enough to define a good string class, a good complex number library, and good specific vector and list classes, but by 1992, nobody had come up with a sufficiently general, flexible, and efficient container framework. Most ideas were derivatives of the original Simula containers-of-objects design. However, such containers cannot hold built-in types or classes not specifically derived from the container element base class (usually called ''Object'' or ''Link''), require ugly and inefficient casting when retrieving objects from a container, leads to overuse of heap storage, and lack any effective notion of container interchangeability; see the discussion in chapter 16 of ''The C++ Programming Language (3rd edition).'' Clearly, templates held the key to a better solution, but until Alex Stepanov came forward with his STL, nobody had constructed a sufficiently general, elegant, and efficient solution to the standard container problem.

The STL evolved into the containers, iterators, and algorithms part of the C++ standard library. Through that, it became the inspiration for a generation of C++ programmers and the starting point for modern generic and generative programming. This had repercussions beyond the C++ community. The success of templates and STL-style containers have been one of the major reasons for the inclusion of simple generics into Java and C# and for the work on comprehensive support for generic programming in dialects of functional programming languages such as Haskell and ML. In this, the impact of C++'s generic programming facilities and techniques resemble the impact that C++'s object-oriented programming facilities and techniques had on the programming community as a whole a decade before when C++'s mainstream acceptance helped inspire CORBA, COM, Java, and even object-oriented facilities for Fortran and Cobol.

So where does the C++ standards process take us from here? The current standard describing both

language and library is a better approximation of my original aims for C++ than any previous version. However, we can obviously do better still and the standards committee intends to live up to that challenge. Unfortunately, there are things that a committee cannot do well. Among the most obvious of those are innovation and design. The current standard contains a few obvious and embarrassing examples of design by committee in both the core language and in the standard library, but I consider those blemishes, potentially correctable, and non-critical. As the work on the next standard begins, the committee must and will try even harder to avoid that trap.

The most successful additions to C++ were developed outside the committee and appeared as well-developed ideas combining features and a philosophy of use. The best example is again the STL. In contrast, less successful developments tend to be minor technical enhancements that grew as the committee tried to compensate for a lack of clear vision. Examples are std::string and some of the details of templates. What the committee seems to do well is to refine facilities given a clearly articulated vision and concrete code examples.

What directions might the committee take? That's for the committee to decide, so I can only guess and I hope that we'll again be surprised by some brilliant new insight or library that can disrupt the regular proceedings and set C++ on a new and more productive track. However, in the absence of such an exciting breakthrough, the predictable directions are Improve the support for generic programming Improve the support for library building and use Make the language more regular, predictable, and teachable Standardize libraries to make C++ a better platform for systems programming

These directions will eventually become reasonably well specified and backed by specific language proposals, library proposals, explanations of programming techniques, and concrete examples. Ideally, the size and number of core language extensions will be low because facilities provided as libraries can be more easily and more thoroughly tested before their inclusion in the standard. Compatibility with the current C++ standard, improvements in the degree of static type safety, and adherence to the zero-overhead principle (''what you don't explicitly ask for, you don't pay for'') are considered very important.

These directions affirm the original guiding principles for C++, distinguish C++ among programming languages, and will serve the diverse C++ community well. Bjarne Stroustrup

Texas A&M University and AT&T Labs
bs@cs.tamu.edu

Wrapping C++ Member Function Calls

Bjarne Stroustrup

AT&T Labs - Research
Florham Park, New Jersey
USA

ABSTRACT

This paper presents a simple, general, and efficient solution to the old problem of ‘‘wrapping’’ calls to an object in pairs of prefix and suffix code. The solution is also non-intrusive, applies to existing classes, allows the use of several prefix/suffix pairs, and can be implemented in 15 simple lines of Standard C++. A robust version of the wrapper is also presented. The claim of efficiency is backed by measurement.

The paper is organized around a series of examples evolving the basic idea into a final robust template class.

1 Introduction

Often, we want to ‘‘wrap’’ a piece of code with some prefix code and some suffix code. For example:

```
grab-lock do-something release-lock
begin-transaction do-something end-transaction
trace-entry do-something trace-exit
```

Typically, there is a prefix/suffix pair that should be applied to many different sections of code. In particular, in a language with classes – such as C++ – the problem often becomes one of ensuring that a ***prefix*( )** is called before a call of a member function and ***suffix*( )** after the call. This is easily achieved explicitly for a few calls. For example:

```
void fct(X* p)
{
    prefix();
    p->f();
    suffix();

    prefix()
    p->g();
    suffix();
}
```

However, application programmers must remember to bracket each call in its proper ***prefix*( )/*suffix*( )** calls. This is tedious and error prone.

The obvious alternative is to add the ***prefix*( )/*suffix*( )** calls to the definitions of the member functions that need them. For example:

```
class X {
public:
    void f() { prefix(); /* f's own stuff */ suffix(); }
    void g() { prefix(); /* g's own stuff */ suffix(); }
    // ...
};
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
void fct(X* p)
{
    p->f();
    p->g();
}
```

This solves the problem for the class user, but is still tedious for the class implementer. Worst of all, this solution requires foresight by the class implementer. Because the solution is intrusive – the class member function code must be modified to add, remove, or change a suffix – wrapping can only be done by someone able and willing to modify the source code.

One advantage of this approach is that it allows some, but not all functions to be wrapped. This can sometimes be a significant advantage. Consider the case where the prefix/suffix provides locking. That is, the class is a form of monitor [Hoare,1974]. In that case, it is not uncommon that a few functions can be performed without locking because they don't modify shared data or access only data that is accessed and modified atomically [Mitchell,1979].

This paper will concentrate on the case where every operation on a class needs to be wrapped.

2 History

''Wrapping'' is an old problem that has been solved many times in various languages and contexts. Monitors provide a solution to the problem of controlling access to a resource in a concurrent system by wrapping calls in a acquire-lock-or-wait and release-lock pair. Many languages provide primitives for wrapping code in acquire/release lock pairs (for example, Mesa's ***MONITOR*** [Mitchell,1979], Java's ***synchronized*** [Lea,1997], and Modula-3's ***LOCK*** [Nelson,1991]).

More general solutions are provided by CLOS where a pair of ***:before*** and ***:after*** methods can wrap calls to objects of classes derived from the one providing the ***:before*** and ***:after*** methods [Keene,1989]. I briefly adopted a variant of that idea for C++'s direct ancestor C with Classes [Stroustrup,1994]. There, one could define a function that would implicitly be called before every call of every member function (except the constructor) and another that would be implicitly called before every return from every member function (except the destructor). The functions providing this prefix/suffix semantics were called ***call()*** and ***return()***. They were used to provide synchronization for the monitor class in the original task library [Stroustrup,1980]:

```
class monitor {
    // ...
    call()   { /* grab lock */ }
    return() { /* release lock */ }
    // ...
};

class X : public monitor {
public:
    void f();
    void g();
    // ...
};

void fct(X* p)
{
    p->f();    // monitor::call(); f()'s own stuff; monitor::return()
    p->g();    // monitor::call(); g()'s own stuff; monitor::return()
}
```

Call and return functions were removed from the language because nobody (but me) used them and because I completely failed to convince people that ***call()*** and ***return()*** had important uses.

In 1988, Mike Tiemann suggested an alternative solution called ''wrappers'' [Tiemann,1988]:

```
struct foo {
    foo();
    ~foo();
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
        virtual int ()foo(int(foo::* pmf)(int), int i)    // virtual wrapper
        {
            prefix();
            int r = (this->*pmf)(i);          // invoke a function
            suffix();
            return r;
        }

        int g(int);
        virtual int h(int);
    };
```

Such a wrapper function was syntactically identified by a pair of parentheses in front of the class name and would wrap calls to functions of its class. The first argument of a wrapper was the member function to invoke, the second and subsequent arguments were the arguments to that function. The basic idea was that a member function called by a user was transformed a call of the wrapper with the arguments needed for the wrapper to do the actual call. For example:

```
void fct(foo* p)
{
    int i = p->g(1);      // p->()foo(&foo::f,1)
    int j = p->h(2);      // p->()foo(&foo::g,2)
    // ...
}
```

This proposal died – after some experimental use – because of complexities of handling argument and return types, and because it was intrusive. That is, you wrapped a class by deriving it from a wrapper base class and that base class had to be written to deal with all combinations of argument and return types needed by the derived classes. This required too much foresight.

This proposal had the interesting property that the wrapper had access to the arguments and the return value of a call. Also, different wrapper functions could be provided for functions with different types.

3 Prefix

In the following sections, a solution to the wrapping problem is introduced in stages. The solution does not require language changes; it consists of two simple template classes.

Overloading the `->` has long been a popular way of specifying a prefix. For example:

```
template<class T>
class Prefix {
    T* p;
public:
    Prefix(T* pp) :p(pp) { }
    T* operator->() { /* prefix code */ return p; }
};

X my_object;
Prefix<X> pref_obj(&my_object);      // available for prefixed use

void fct(Prefix<X> p)
{
    p->f();    // prefix code; f()
    p->g();    // prefix code; g()
}
```

Note how the prefix is attached to the object (and its functions) non-intrusively. I can use ***Prefix*** to control access to classes that I cannot change. I can also use ***Prefix*** to control access to classes that have not been specifically been designed for controlled access. In this, the templated ***Prefix*** approach is more general and more flexible than the C with Classes approach and the Tiemann wrapper approach. These approaches required that controlled classes be derived from a specific base class. The templated ***Prefix*** approach requires less foresight on the part of the programmer.

Published in the June 2000 issue of "The C++ Report". All rights reserved

4 Suffix

Unfortunately, it was not obvious how to extend the templated ***Prefix*** approach to deal with suffixes. However, it can be done. One non-intrusive way of getting something done ‘‘later’’ is to create an object with a class with a destructor that does that something. The destructor will be executed at the end of the object’s lifetime. For example:

```
class Suffix {
public:
    ~Suffix() { /* suffix code */ }
};

void fct(X* p)
{
    Suffix suf;
    p->f();
    // ...
    // the suffix code is implicitly executed here
}
```

This is a simplified variant of the technique commonly known as ‘‘resource acquisition is initialization’’ [Stroustrup,2000]. This technique has the nice property that the suffix code is executed even if *f*( ) throws an exception.

The problems with the ‘‘resource acquisition is initialization’’ technique in this context are:

[1] the user must explicitly declare an object

[2] the user must name the ***Suffix*** object

[3] the suffix isn’t executed until the end of the block

5 Prefix and Suffix

These problems can be solved by combining the use of a destructor to invoke a suffix with the use of an *operator->* to invoke a prefix. As before, *operator->* executes the prefix and returns something that identifies the object for which the function is to be called:

```
template<class T>
class Wrap {
    T* p;
public:
    Wrap(T* pp) :p(pp) { }
    Call_proxy<T> operator->() { prefix(); return Call_proxy<T>(p); }
};
```

What is different here is that the value returned by *operator->( )* is an object holding the pointer to the object to be called rather that the pointer itself. We can define *Call_proxy*, with a destructor that calls the suffix:

```
template<class T>
class Call_proxy {
    T* p;
public:
    Call_proxy(T* pp) :p(pp){ }
    ~Call_proxy() { suffix(); }
    T* operator->() { return p; }
};
```

We can now write:

```
#include<iostream>
using namespace std;

void prefix() { cout << "prefix "; }
void suffix() { cout << " suffix\n"; }
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
template<class T> class Call_proxy { /* ... */ };

template<class T> class Wrap { /* ... */ };

class X {  // one user class
public:
    X() { cout << "make an X\n"; }
    int f() const { cout << "f()"; return 1; }
    void g() const { cout << "g()"; }
};

class Y {  // another user class
public:
    Y() { cout << "make a Y\n"; }
    void h() const { cout << "h()"; }
};

int main()      // simple test code
{
    Wrap<X> xx(new X);
    Wrap<Y> yy(new Y);

    if (xx->f()) cout << "done\n";
    xx->g();
    yy->h();
    return 0;
}
```

Each call of *xx* and *yy* is bracketed by a pair of prefix()/suffix() calls, so the program produced:

```
make an X
make a Y
prefix f() suffix
done
prefix g() suffix
prefix h() suffix
```

Note that wrapping of all calls of all functions is not a simple transformation. For example, you cannot write a function or a macro to do so. Even writing a macro to wrap a single call is nontrivial because of the need to perform the suffix before returning the result.

It is important that *Wrap* is implemented completely in Standard C++ [Stroustrup,2000]. No language extensions or preprocessor magic are required. This implies that variations of the idea can be used to suit needs and tastes. Also, the idea is immediately applicable using current C++ implementations.

6 Ownership

The simple *Wrap* and *Call_proxy* classes do not implement a general and safe model of ownership. In particular, they do not define copy constructors and assignment operators that ensure that objects pointed to are deleted exactly once.

Also, *Call_proxy* isn't intended as a general class, so we can ensure that *Call_proxy*s are created by *Wrap* only and their lifetimes are limited to a single wrapped call:

```
template<class T> class Wrap;

template<class T>
class Call_proxy {
    T* p;
    mutable bool own;

    Call_proxy(T* pp) :p(pp), own(true) { }        // restrict creation
    Call_proxy(const Call_proxy& a) : p(a.p), own(true) { a.own=false; }
    Call_proxy& operator=(const Call_proxy&);      // prevent assignment
public:
    template<class U> friend class Wrap;
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
        ~Call_proxy() { if (own) suffix(); }

        T* operator->() const { return p; }
};
```

The *own* variable is needed to ensure that only the last object in a chain of copies executes *suffix*( ). The way *Wrap* uses *Call_proxy*, no copying is needed, and most (all?) compilers are clever enough to avoid spurious copies. However, to be cautious, I introduced *own* for the benefit of compilers with poor optimizing skills.

The wrapper itself is intended for general use. Objects with a lifetime controlled by the user are passed to *Wrap* as object (by reference). Objects that *Wrap* is supposed to delete are passed as pointers. In the latter case, the wrapper maintains a use count:

```
template<class T>
class Wrap {
        T* p;
        int* owned;
        void incr_owned() const { if (owned) ++*owned; }
        void decr_owned() const { if (owned && --*owned == 0) { delete p; delete owned; } }
public:
        Wrap(T& x) :p(&x), owned(0) { }
        Wrap(T* pp) :p(pp), owned(new int(1)) { }

        Wrap(const Wrap& a) :p(a.p), owned(a.owned) { incr_owned(); }
        Wrap& operator=(const Wrap& a)
        {
                a.incr_owned();
                decr_owned();
                p = a.p;
                owned = a.owned;
                return *this;
        }

        ~Wrap() { decr_owned(); }

        Call_proxy<T> operator->() const { prefix(); return Call_proxy<T>(p); }

        T* direct() const { return p; } // extract pointer to wrapped object
};
```

See [Stroustrup,2000] for a discussion of the non-intrusive use-count scheme used.

7 Parameterization

So far, *Wrap* has called the global functions *suffix*( ) and *prefix*( ). That is not flexible enough or general enough. We'd like to wrap different objects with different prefix/suffix pairs and to wrap a single object in more than one pair. This is simply handled by making *prefix* and *suffix* parameters.

When parameterizing *Wrap*, the major question is whether to provide per-class parameterization or per-object parameterization. I chose the latter because it is the most flexible solution.

Call_proxy must have a suffix parameter:

```
template<class T, class Pref, class Suf> class Wrap;

template<class T, class Suf>
class Call_proxy {
        T* p;
        mutable bool own;
        Suf suffix;

        Call_proxy(T* pp, Suf su) :p(pp), own(true), suffix(su) { }          // restrict creation

        Call_proxy& operator=(const Call_proxy&);       // prevent assignment
public:
        template<class U, class P, class S> friend class Wrap;
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
    Call_proxy(const Call_proxy& a) : p(a.p), own(true), suffix(a.suffix) { a.own=false; }
    ~Call_proxy() { if (own) suffix(); }

    T* operator->() const { return p; }
};
```

Wrap needs a prefix and a suffix parameter:

```
template<class T, class Pref, class Suf>
class Wrap {
    T* p;
    int* owned;
    void incr_owned() const { if (owned) ++*owned; }
    void decr_owned() const { if (owned && --*owned == 0) { delete p; delete owned; } }
    Pref prefix;
    Suf suffix;
public:
    Wrap(T& x, Pref pr, Suf su) :p(&x), owned(0), prefix(pr), suffix(su) { }
    Wrap(T* pp, Pref pr, Suf su) :p(pp), owned(new int(1)), prefix(pr), suffix(su) { }

    Wrap(const Wrap& a)
        :p(a.p), owned(a.owned), prefix(a.prefix), suffix(a.suffix) { incr_owned(); }
    Wrap& operator=(const Wrap& a)
    {
        a.incr_owned();
        decr_owned();
        p = a.p;
        owned = a.owned;
        prefix = a.prefix;;
        suffix = a.suffix;
        return *this;
    }

    ~Wrap() { decr_owned(); }

    Call_proxy<T,Suf> operator->() const { prefix(); return Call_proxy<T,Suf>(p,suffix); }

    T* direct() const { return p; } // extract pointer to wrapped object
};
```

Given that, here is a simple test program:

```
#include<iostream>
using namespace std;

void prefix() { cout << "prefix "; }
void suffix() { cout << " suffix\n"; }

class X {  // one user class
public:
    X() { cout << "make an X\n"; }
    ~X() { cout << "destroy an X\n"; }
    int f() const { cout << "f()"; return 1; }
    void g() const { cout << "g()"; }
};

class Y {  // another user class
public:
    Y() { cout << "make a Y\n"; }
    ~Y() { cout << "destroy a Y\n"; }
    void h() const { cout << "h()"; }
};

struct Pref { void operator()() const { cout << "Pref "; } };
struct Suf { void operator()() const { cout << " Suf\n"; } };
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
int main()      // test program
{
    Wrap<X, void(*)(), void(*)()> xx(new X, prefix, suffix);
    Wrap<Y, void(*)(), void(*)()> yy(new Y, prefix, suffix);
    Wrap<X, void(*)(), void(*)()> x2 = xx;
    X x;
    Wrap<X, Pref, Suf> x3(x, Pref(), Suf());

    if (xx->f()) cout << "done\n";
    xx->g();
    x2->g();
    xx = x2;
    x2->g();
    x3->g();
    yy->h();
    return 0;
}
```

Here, I use function objects primarily to get better inlining and therefore better run-time performance on an average C++ implementation (see §9). However, function objects have an important advantage over functions in that they can be used to hold data. For example, if we want to wrap calls in a pair of lock/unlock operations, we often need to say what lock is to be used. This can be done either by having one function for each lock, or by having function objects that can be initialized with the lock to be used. For example:

```
struct Lock {
    sys_lock lck;
    Lock(sys_lock& x) : lck(x) { }
    void operator()() const { grab(lck); }
};

struct Unlock {
    sys_lock lck;
    Unlock(sys_lock& x) : lck(x) { }
    void operator()() const { release(lck); }
};

Wrap<X, Lock, Unlock> x3(x, Lock(screen_lock), Unlock(screen_lock));
Wrap<X, Lock, Unlock> x4(y, Lock(lock3a), Unlock(lock3a));
```

8 Notational Convenience

The flexibility was bought at the expense of notational convenience. I don't think anyone likes to read or write declarations like:

```
Wrap<X, void(*)(), void(*)()> xx(new X, prefix, suffix);
```

Typically, only a few forms of wrapping are used in a program and people want to refer to those with the greatest degree of simplicity. The most general notation is rarely necessary.

A derived class can be used to provide more concise notation by specialization. Deriving a class also opens the possibility of defining the most appropriate parameters to constructors:

```
template<class T>
class Shared : public Wrap<T, Lock, Unlock>
{
public:
    Shared(T& obj, sys_lock& lck) : Wrap<T, Lock, Unlock>(obj, Lock(lck), Unlock(lck)) { }
    Shared(T* ptr, sys_lock& lck) : Wrap<T, Lock, Unlock>(ptr, Lock(lck), Unlock(lck)) { }
};

Shared<X> x3(x, screen_lock);
Shared<X> x4(y, lock3);
```

Unfortunately, we have to specify both the wrapped type and the wrapped object. Because of the generality of the constructor mechanism, it is not possible to deduce the one from the other. However, we could

Published in the June 2000 issue of "The C++ Report". All rights reserved

define a function to take advantage of type deduction:

```
template<class T, class Pref, class Suf>
Wrap<T,Pref,Suf> make_wrap(T& x, Pref pr = Pref(), Suf su = Suf())
{
    return Wrap<T,Pref,Suf>(x,pr,su);
}

void f(X& x)
{
    g(make_wrap(x));
    // ...
}
```

Unfortunately, we must specify the type of the object to which we assign the result of *make_wrap*( ). In this example, we must define the argument type for *g*( ). This defeats this technique for simplifying the *Wrap* notation except where the target is itself a template.

Naturally, if we wanted to wrap only a single type, or if we wanted to make wrapping a particular type notationally simple, we could introduce a name for that particular wrapper type. For example:

```
template<class T>
struct Tracer : public Wrap<T,void(*)(),void(*)()> {
    Tracer(T& x) : Wrap<T,void(*)(),void(*)()>(x,trace_on,trace_off) { }
};

X x;
Tracer<X> xx(x);

typedef Tracer<X> Xtracer;
Xtracer xxx(x);
```

9 Efficiency

The *Wrap* class and its auxiliary *Call_proxy* class have doubled in source code size from the initial version to the version that is more flexible, general, and safer for a programmer to use. How does this evolution affect efficiency? To find out I did a simple timing test wrapping calls of an empty, noninlined member function.

```
class X {  // one user class
public:
    void g() const; // note: not even virtual
};

void X::g() const {/* cout << "g()"; */}
```

I measured direct calls of the function

```
X* p = new X;
// ...
p->f();
```

and calls using the simple *Wrap* from §5

```
X x;
Wrap<X> xx(&x);     // simple Wrap from §5
// ...
xx->f();
```

and calls using the robust and parameterized *Wrap* from §7 with pointers to inline functions as prefix and suffix

```
inline void pref() { }
inline void suf() { }
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

```
Wrap<X, void(*)(), void(*)()> xx(&x, pref, suf);    // robust and parameterized Wrap from §7
// ...
xx->f();
```

and calls using the robust ***Wrap*** from §7 using function objects as prefix and suffix to simplify inlining

```
struct Pref { void operator()() const { } };    // use to function objects to simplify inlining
struct Suf { void operator()() const { } };

Wrap<X, Pref, Suf> xx(&x, Pref(), Suf()); // robust and parameterized Wrap from §7
// ...
xx->f();
```

Thus, the measurements show the overhead of wrapping compared to the overhead of a call of an empty function.

Because the simple ***Wrap*** doesn't actually perform any additional computation, we can expect a good optimizer to eliminate all overhead. However, the robust ***Wrap*** does some computation related to ownership, so we should expect to pay a little – but only a little – for using it.

The code is available from my home pages: http://www.research.att.com/~bs/papers.html.
To get an idea of the impact of compiler technology and machine architecture, I ran the tests on three different machine architectures using a total of seven different C++ implementations.

As is typical for non-trivial C++ code, the most obvious difference was between runs using different level of optimization.

Run times (debug and optimized)						
	impl.1d	impl.1	impl.3d	impl.3	impl.4d	impl.4
no Wrap	.14	.06	.21	.09	.42	.03
simple Wrap, inline function	.47	.07	.45	.10	1.82	.06
robust Wrap, inline function	.74	.25	.68	.23	2.40	.14
robust Wrap, function object	.86	.11	.59	.15	2.52	.10

These numbers are the average of many runs and the results between runs do not show significant variation. The unit is microseconds per wrapped or unwrapped call.

I did not try to tune the code by selecting compiler and optimizer options. In each case, a default ''debug mode'' and a default ''release mode'' was used. All runs were done with all Standard C++ facilities enabled.

It may surprise some that the run-time cost of using debug mode can be a factor of 25. Disabling inlining and all non-trivial operations leave templated code very inefficient. Templates and inlining were designed with optimization in mind.

Note that for implementations 1 and 3, there is no statistical difference between the cost of a direct call and the cost of a call through the simple wrapper.

Please not compare the different implementations – they run of completely different hardware and these simple measurements are not suitable benchmarks.

Run times (optimized)							
	impl.1	impl.2	impl.3	impl.4	impl.5	impl.6	impl.7
no Wrap	.06	.07	.09	.03	.03	.04	.04
simple Wrap, inline function	.07	.07	.10	.06	.15	.14	.06
robust Wrap, inline function	.25	.24	.23	.14	.22	.28	.12
robust Wrap, function object	.11	.17	.15	.10	.15	.22	.09

Implementations 2 and 3 ran on the same machine. Similarly, implementations 4, 5, 6, and seven ran on the same machine. I conclude that the overhead depends more on compiler technology than on machine architecture.

Note that implementations 1, 2, and 3 provide proof by example that simple wrapping need not incur overhead. Implementations 1 and 3 show that the cost of robust wrapping can be less than a function call, though it typically is equivalent to almost two function calls.

Clearly, implementations are still better at inlining function objects than at optimizing calls through a

Published in the June 2000 issue of "The C++ Report". All rights reserved

pointer to an inline function.

Two implementations didn't support friend templates, so I had to modify ***Call_proxy*** to make more functions public. However, this does not affect the generated code – beyond allowing code to be generated from compilers that isn't yet quite up to the standard in this area.

10 Limitations

A fundamental limitation of this prefix/suffix approach is that neither prefix nor suffix has access to the arguments to the called function or the result produced. For example, it is not possible to provide a prefix/suffix to record the results of calls or to map calls into message send operations. In this, the prefix/suffix approach differs from the Tiemann wrappers [Tiemann,1988].

Another limitation of this approach is that the prefix and suffix are independent. For example, it is not possible to write a prefix/suffix pair that catch exceptions thrown by the wrapped functions. It is not possible to open a try-block in the prefix and close it in the suffix.

However, it is possible for a prefix and a suffix to communicate. A simple use of that is for a prefix and a suffix to access the same variable as is typically done for locks and trace state. To do the call, ***Wrap*** and ***Call_proxy*** get a pointer to the object called. Thus, they might keep a record of objects accessed, but they do not know which function is called for those objects.

11 Techniques

A wrapper can wrap every class. Consequently, it can wrap a wrapper. This can be useful. For example:

```
Tracer< Shared<X> > xx;   // trace accesses to shared data
```

Alternatively, several operations can be combined into a single prefix or suffix. For example:

```
struct TSpref {
    sys_lock lck;
    TSpref(sys_lock& x) : lck(x) { }

    void operator()() const { trace_on(); grab(lck); }
};

struct TSsuf {
    sys_lock lck;
    TSsuf(sys_lock& x) : lck(x) { }

    void operator()() const { release(lck); trace_off(); }
};

template<class T>
class Trace_lock : Wrap<T, TSpref, TSsuf> {
    Trace_lock(T& obj, sys_lock& x) : Wrap<T, TSpref, TSsuf>(obj, TSpref(x), TSsuf(x)) { }
    Trace_lock(T* ptr, sys_lock& x) : Wrap<T, TSpref, TSsuf>(ptr, TSpref(x), TSsuf(x)) { }
    // ...

};

Trace_lock xx(x, screen_lock);
```

This is more work to define, but in general more flexible. To a user, the details of how a wrapper is implemented is of little practical interest.

As mentioned in §1, it is not always a good idea to apply the prefix and suffix to every access to an object. When we need to access an object directly, we can use the original object. For example:

```
void code(X* p)
{
    Shared<X> xx(*p);

    int i = xx->f(2);          // controlled access
    int j = p->f(3);           // direct access
    // ...
}
```

Published in the June 2000 issue of "The C++ Report". All rights reserved

However, this approach has the problem that the programmer has to remember what the original object was. Typically it is simpler and less error-prone to extract a pointer to the wrapped object from the wrapper using the *direct*( ) function supplied for that purpose. For example:

```
void user(Shared<X>& xx)
{
    int i = xx->f(2);              // controlled access
    int j = xx.direct()->f(3);     // direct access
    // ...
}
```

12 Conclusions

Wrapping calls to member functions of a class by a prefix/suffix pair is simple and efficient and requires no extensions to Standard C++. Wrapping is non-intrusive and does not place requirements on the wrapped class. In particular, a wrapped class need not be derived from a particular base class. Current C++ implementations are capable of coping efficiently with the technique. The primary limitation of the prefix/suffix approach is that the arguments to a call and its return value is not accessible to the prefix/suffix.

13 Acknowledgements

My measurements were simplified by code borrowed from Andrew Koenig [Koenig,2000].

14 References

[Hoare,1974] C. A. R. Hoare: *Monitors: An Operating System Structuring Concept*. Communications of the ACM 17(10), October 1974.

[Keene,1989] Sonya A. Keene: *Object-Oriented programming in Common Lisp*. Addison-Wesley. 1989. ISBN 0-201-17589-4.

[Lea,1997] Doug Lea: *Concurrent Programming in Java*™. Addison-Wesley. 1997. ISBN 0-201-69581-2.

[Koenig,2000] Andrew Koenig and Barbara Moo: *Performance: Myths, Measurements, and Morals*. JOOP 13(1,2). January and March 2000.

[Mitchell,1979] James G. Mitchell, et.al.: *Mesa Language Manual*. XEROX PARC, Palo Alto, CA. CSL-79-3. April 1979.

[Nelson,1991] Greg Nelson (Editor): *Systems Programming with Modula-3*. Prentice Hall. 1991. ISBN 0-13-590464-1.

[Stroustrup,1980] Bjarne Stroustrup: *A Set of C Classes for Co-routine Style Programming*. Bell Laboratories Computer Science Technical Report CSTR-90. November 1980.

[Stroustrup,1995] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1995. ISBN 0-201-54330-3.

[Stroustrup,2000] Bjarne Stroustrup: *The C++ Programming language (Special Edition)*. (2nd edition, 1991; 3rd edition, 1997). Addison-Wesley. ISBN 0-201-88954-4 and 0-201-70073-5.

[Tiemann,1988] Michael Tiemann: *''Wrappers:'' Solving the RPC problem in GNU C++*. Proc. USENIX C++ Conference. Denver, CO. October 1988.

Published in the June 2000 issue of "The C++ Report". All rights reserved

Rejuvenating C++ Programs through Demacrofication

Aditya Kumar
Computer Science Engineering
Texas A&M University
College Station,Texas-77840
Email: hiraditya@neo.tamu.edu

Andrew Sutton
Computer Science Engineering
Texas A&M University
College Station,Texas-77840
Email: asutton@cse.tamu.edu

Bjarne Stroustrup
Computer Science Engineering
Texas A&M University
College Station,Texas-77840
Email: bs@cs.tamu.edu

***Abstract*—We describe how legacy C++ programs can be rejuvenated using C++11 features such as generalized constant expressions, perfect forwarding, and lambda expressions. In general, this work develops a correspondence between different kinds of macros and the C++ declarations to which they should be transformed. We have created a set of *demacrofication* tools to assist a developer in the rejuvenation of C++ programs. To evaluate the work, we have applied the rejuvenation tools to a number of C++ libraries to assess the extent to which these libraries might be improved by demacrofication. Results indicate that between 68 and 98% of potentially refactorable macros could be transformed into C++11 declarations. Additional experiments demonstrate why these numbers are not readily achieved using fully automated rejuvenation tools. We also discuss some techniques to further assist in automating rejuvenation process.**

***Index Terms*—source code rejuvenation; macros; C++11; refactoring; demacrofication**

I. INTRODUCTION

As software developers make the decision to migrate C++98 software to C++11 it becomes advantageous to embrace new libraries, idioms, and language features offered by the new language. Using C++11 programming styles can improve readability (hence maintainability), reliability, and performance. We refer to this kind of program modification as *source code rejuvenation* [1]: a one-time modification to source code that replaces deprecated language and library features and programming idioms with modern code.

In this work, we are interested in replacing C preprocessor macros with new features and idioms in the C++11 programming language. The kinds of problems engendered by the C preprocessor are many and well known [2], [3]. Because the preprocessor operates on the token stream independently from the host language's syntax, its extensive use can result in a number of unintended consequences. Bugs resulting from these conflicts can lead to hard-to-debug semantic errors.

In the C-family of languages there has been an effort to limit the use of unstructured preprocessor constructs. Java has none, C# limits preprocessing to conditional configuration, D replaces common preprocessor uses with different kinds of declarations (e.g., `version`), and most modern C++ coding standards ban most "clever" or ad hoc usage of the C preprocessor [2], [4]–[6]. In C++11, the number of reasonable uses of the C preprocessor has been reduced. The use of generalized constant expressions, type deduction, perfect forwarding, lambda expressions, and alias templates eliminate the need for many previous preprocessor-based idioms and solutions.

Older C++ programs can be rejuvenated by replacing error-prone uses of the C preprocessor with type safe C++11 declarations. In this paper, we present methods and tools to *support* the task of *demacrofying* C++ programs during rejuvenation. This task is intended to be completed with the help of a programmer; it is not a fully automated process.

In particular, we have developed a classification of macros defined in terms of their properties of completeness and dependence and how they correspond to C++11 expressions, statements, and declarations. We have built a set of tools (`cpp2cxx` that assists in macro refactoring (the transformation from macro to declaration), and another tool assists in the iterative application of those refactorings to a software build.

Note that without these new C++11 features, there would be no cause to replace the previous use of macros. The transformations we propose in this paper cannot be achieved using C++98/03. The approach is intended to be applied as developers migrate to the new programming language.

In order to evaluate our approach, we have applied it to the demacrofication of 7 different C++ libraries with over 1.5 million lines of non-comment non- blank code. The evaluation comprised three different studies. We first applied the refactoring tool to each of the libraries in order to identify the kinds of macros used and the extent to which automated transformations could remove replaceable macros. We then conducted a case study applying the process manually as we fully migrated one of those libraries to C++11. Finally, we selected two of the remaining 6 libraries and experimented with fully automating the rejuvenation using our iterative, automated refactoring tools.

II. RELATED WORK

Pirkelbauer et al. describe source code rejuvenation as a "source-to-source transformation that replaces deprecated language features and idioms with modern code" [1]. Unlike refactorings, which are recurring maintenance tasks aimed to improve software design, source code rejuvenation is a one-time transformation caused by a major evolutionary change in

978-1-4673-2312-3/12/$31.00 © 2012 IEEE

the host language or underlying platform. Our goal in this work is to (partially) automate the rejuvenation of legacy C++ programs by replacing some macros with new C++11 declarations.

Ernst et al. studied preprocessor usage in a number of C programs [3]. Most relevant to our work, is their categorization of macro bodies (the replacement text) into 28 different categories, several of which directly correlate to C (and C++11) syntax trees. The results of their study suggest that 42% of macros had replacement text which were constants, 33% were expressions, 5.1% were statements, and 2.1% were types. If one could directly associate each replacement text with a C++11 declaration, then we could expect to deprecate roughly 80% of all macros used in those programs. Obviously, this will not be the case since some macros may reference free variables, contain recursive instantiations, be defined after their first use, etc. Our classification is, in part, informed by the work of Ernst *et al*. However, we do not use their classification outright, ours is primarily based on the relationship between a macro and its corresponding C++11 declaration.

Mennie and Clarke aim to replace a number of macros with equivalent C++ declarations [7]. In his thesis, Mennie describes how simple constants and constant expressions, enumerated constants, and parameterless functions can be migrated to C++ declarations [8]. The approach taken here analyzes macro definitions and their uses within a program, while ours focuses solely on macro definitions. We also benefit from new C++11 language features such as perfect forwarding, which make it possible to generate correct definitions of correct inline functions.

Gravley and Lakhotia also describe a method for replacing sequences of constants (including #define directives) with `enum`s [9]. The approach relies on heuristics as "visual cues" to facilitate the grouping of constant expressions into enumerations. A similar application is used to transform sets of static final fields into enumerations in Java but is based on interprocedural type inference instead of heuristics [10]. These approaches focus on specific transformations of macros into declarations, whereas we try to accommodate a range of transformations. The techniques presented in these works also come with an added degree of difficulty since the macros they are transforming attempt to map multiple macro definitions into a single enumeration. We describe a 1-to-1 mapping between macro definitions and their corresponding C++11 declarations.

Most refactoring tools involving the C preprocessor involve its use for source code configuration and portability [11]. For example, Baxter and Mehlich describe automated methods for removing dead preprocessor configurations [12]. Garrido and Johnson study issues related to more general refactorings of C macros and conditionals [13]. The thesis of Vidács addresses similar refactorings at the model level [14] after abstracting from the source code through reverse engineering [15]. These works vary from ours in the domain to which they apply. We do not directly address issues related to macros used to conditionally compile source code since there is no way (in any version of C++) to represent such variations directly in the syntax of the language.

III. REJUVENATING C++ PROGRAMS

The process by which we refactor macros during rejuvenation is not a single- pass operation. The general process of demacrofication is shown in Figure 1. The reason for this is that we have not found any particular set of criteria that would allow us to apply a transformation to all macros in a C++98 program and produce a fully correct, fully functioning C++11 program. There does not seem to be a silver bullet. We have, however, found criteria that allow some transformations to be fully automated and others that require user intervention.

Fig. 1: Demacrofication is a process involving multiple transformations of the source code.

The process begins by suggesting a set of initial refactorings. These initial refactorings are based on a classification of macros in the initial source text and their relation to C++11 declarations (if any). This suggestions might be applied automatically or incrementally by a developer (this is a decision made by the developer). We note that not all automatic refactorings will result in correct programs. The conditions resulting in breakage are discussed throughout this section. Eventually, through iterative analysis and fixing of any broken transformations, the programmer can produce a final, functioning version of the source code.

IV. CLASSIFYING MACROS

The classifications of macros described by Ernst et al. was based on the structures the macros represent in terms of the C language grammar, frequency of usage and the context of usage [3]. Mennie et al. classified the macros based on the styles of macro usage in the software that they used for their case study [7]. As a result, many of the macros in those different classifications have no direct correspondence with C or C++ declarations. For example, there is no way to join two tokens into a single identifier except to use the C Preprocessor.

Because our goal is to replace macros with C++ declarations, we have a somewhat restricted view of these taxonomies. In particular, we are only interested in the kinds of macros that could eventually be replaced. To achieve this, we found the following three parameters that could serve as the classification criteria for the macros:

A. syntactic structure of the macro bodies,
B. the presence (or absence) of free variables in the replacement text, and
C. the presence of macros within conditionally compiled sections of text.

This classification is defined solely for the purpose of informing our tools about the which macros can be automatically transformed and which might require user intervention.

A. Objects and Functions

A C Preprocessor macro is essentially a named sequence of tokens, called its *replacement text*. Macros come in two flavors: object-like macros and function-like macros [16]. Their definitions have this form:

```
// object-like
#define PI 3.14

// function-like
#define SUM(A, B) ((A)+(B))
```

Here, `PI` is an object-like macro with the replacement text `3.14`, and `SUM` is a function-like macro that will expand to the expression `((A)+(B))`. Whether a macro is object-like or function-like will affect the transformation into a C++ declaration (if any transformations are viable).

B. Completeness

The syntax of a macro's replacement text is helpful in determining whether it can be expressed as declaration in the C++ programming language. Based on this criteria, macros can be classified into two categories. If it is not possible to represent the replacement text of a macro as a C++ expression we call that macro as ***partial***. These types of macros are sometimes used to make the code look more readable, concatenate or stringify tokens etc. For example:

```
#define C_MODE_START extern "C" {
#define CONCAT(a,b) a##b
```

Conversely, if the replacement text of a macro can be expressed as a C++ expression we call the macro as ***complete***. For example:

```
#define Z ((X)+(Y))
#define FUN_CALL do { f(); } while(0)
#define TYPE_CHAR (char*)
```

Note that in the case of macro like `FUN_CALL` which emulates a compound statement, we have relaxed the requirement for a terminating semicolon. This definition of a ***complete*** macro involves an assumption that macro-dependencies are not taken into account. Taking macro-dependencies into consideration would make it impossible to make such a reasoning because C Preprocessor allows token pasting etc. Therefore, based on the definition, a ***complete*** macro could be a *type-expression, value-expression, declaration, or statement*. If we consider the way the Pivot [17] represents expressions, ***complete*** macros correspond to complete AST fragments.

C. Dependency

In a macro, we consider an identifier to represent a "free variable" when it is not declared as a parameter of that macro. We say that a macro with any free variables is ***dependent*** since the final valuation of that macro depends on the context in which it is expanded. A macro that contains previously defined macros or even C++ keywords is considered to be dependent. We include keywords because preprocessing may actually redefine keywords. Examples of dependent macros include:

```
#define __GNUG__ (__GNUC__&&__cplusplus)
#define DIFF(A,B) (MAX((A),(B))-MIN((A),(B)))
#define TYPE_CHAR (char*)
```

In the future, we plan to make built-in type names not free variables, meaning that `TYPE_CHAR` would be classified as closed. This will eventually permit us to reason bout about type expressions such as `char*`.

In contrast, a ***closed*** macro does not contain previously (system/user) defined macros or C++ keywords or any other identifier not in its scope. For example, the following macros are considered to be closed.

```
#define ARCHITECTURE 32
#define CONCAT(a,b) a##b
```

The reason that `CONCAT` is closed is that `a` and `b` are local to the scope of the macro: they are parameters.

D. Configuration

Many macros are used to control the conditional compilation (and occasionally inclusion) of C++ files. Ernst *et al* claim that 6.5% of all macros are used in conditional directives [3], although the numbers reported by Sutton for C++ Libraries appear to be much higher (although not specifically reported) [11]. We say that any macro appearing in a conditional or include directive is ***configurational***. For example:

```
#ifdef BOOST_NO_NOEXCEPT
...
#endif
```

The macro `BOOST_NOEXCEPT` is categorized as configurational. Configurational macros are never transformed. Replacing a configurational macro with a C++ declaration would make it invisible to the preprocessor, thus guaranteeing a broken the build.

A macro that is not configurational is ***non-configurational***.

E. Definition Order

The C Preprocessor does not follow the usual "declare-before-use" program structure of C and C++. A macro can be referenced any time before it is defined, and expansions of undefined macros simply result in an empty sequence of replacement tokens. Consider the following program. The `CEL` macro converts a Fahrenheit value to Celsius value.

```
#define SLOPE (5.0 / 9.0)
#define CEL(T) SLOPE * (T - THRESH)
#define THRESH 32.0
```

The macro `THRESH` is referenced before it is defined. The macro is only looked up when an expansion of `CEL` is requested, so it will become a valid use. Contrast that with a corresponding C++ program:

```
double SLOPE = 5.0 / 9.0;
double CEL(double T) {
  return SLOPE * (T - THRESH); // Error!
}
double THRESH = 32.0;
```

Obviously, `THRESH` is not in scope when it is referenced in the body of the `CEL` function. This is not a hypothetical

Fig. 2: The `CEL` macro depends on `SLOPE` and `THRESH`.

problem; we found instances of macro ordering problems when conducting our experiments (e.g., wxWidgets-2.9.3 has several occurrences of this exact problem).

Solving this refactoring problem turns out to be a straightforward application of dependency analysis. We can construct a directed macro dependency graph in which each vertex represents a defined macro, and an edge (u, v) represents the use of v by the definition of u. The macro dependency graph corresponding to the program above is shown in 2. A correct ordering of definitions can be generated by topologically sorting the graph.

However, one does not simply reorder definitions in a program. Experience shows that programmers frequently reject nonlocal transformations of their source code; they want to compare the original source with the improved source on a line-for-line basis. If macro dependencies span multiple files or are lexically distant in the same file, then automated reorderings make such comparisons difficult to make or reason about.

V. MAPPING MACROS TO C++

We classified the macros across three different categories based upon the properties described above. Having done that, we can now define a partial mapping from macros to C++11 declarations. *This mapping is undefined for any macro that is* ***partial*** *or* ***configurational****.*

To construct the mapping of macros to C++11 declarations, we consider the programming elements of C++ abstractly, in terms of IPR [17]. IPR is a complete, efficient, and hierarchical representation of the C++ language. In contrast to a typical compiler's AST, IPR represents only the internal elements of the language, not its external syntax. In IPR, nearly every kind program element is an expression; names, literals, types, and statements are different kinds of expressions. A declaration is a kind of statement, examples of which include classes, functions, and variables.

The reason for using IPR as a reference model in this context is that it provides a framework for classifying different kinds of expressions that might be found in the replacement text of macros. We do not actually use IPR as a physical artifact in our implementation. Our classification of macros with program elements is simply motivated by IPR's design.

In the following sections, we describe how complete, non-configurational macros are mapped to C++11 declarations. The dependency of macros, whether or not they contain free variables, plays a significant role in this mapping. In some cases, dependent macros may not have a defined mapping to a C++11 declaration.

The basis for this the mapping is derived from the function-like or object-like nature of the macro and the classification of its replacement text as an expression, statement, declaration, or type. In this discussion we assume that complete statements can be fully parsed, resulting in a hypothetical AST and allowing us to reason about the kind of program fragments represented by replacement text. Our implementation of this assumption partially parses the expression, and makes the judgments heuristically Essentially, it is a best-effort attempt to characterize a sequence of tokens as a C++ expression.

In subsection, we present examples of macros that can be transformed and give the general mapping for those macros. We discuss issues related dependent macros and when mappings can be defined. We also present mappings that require "perfect knowledge" from a compiler front-end and how that information can be used to resolve some cases.

A. Expression Alias

An expression alias is a macro whose replacement text can be recognized as a C++ expression (but not a statement or a type). The macro may be dependent or not.

If the expression parsed from the replacement text has *literal type*, then the macro can be replaced with a constant expression declaration. By By literal type we mean that the type of the expression is the same as that of a literal values (e.g., 3 is an `int` literal, `3.14` is a double literal, etc.). For example:

```
#define PI 3.14
#define SEVEN 3 + 4
#define FILENAME "header.h"
```

Note that the string `"header.h"` has literal type because its type, `const char*`, is considered as such when referring to a string literal. The rule for transforming for these macros whose replacement text is an expression with literal type is:

```
// macro
#define A X

// C++11 declaration
constexpr auto A = X;
```

The type of the declared variable `A` is `auto`; it is deduced from the type of the initializer expression `X`. The examples above would be mapped to:

```
constexpr auto PI = 3.14;
constexpr auto SEVEN = 3 + 4;
constexpr auto FILENAME = "header.h";
```

A more advanced transformation, one that can fully type-check a C++ expression, could automatically assign a concrete type to the declaration of `A`. While our implementation does not perform this transformation (we do not do full type checking), we do want to highlight the fact that concrete types can be deduced from the type of literal expressions.

It is not always possible to guarantee a correct transformation when the macro is dependent since the scope of the free variable may differ from that of the macro's definition. However, if we determine that the free variables are previously

defined macros that are mapped to constant expression declarations, then the transformation can proceed in the manner given above. Consider:

```
#define R 10
#define PI 3.14
#define AREA_CIRCLE PI * R * R
```

The definition of AREA_CIRCLE depends on the identifiers PI and R. Here, we have seen that both PI and R are previously defined macros, and we know that both can be replaced by constexpr variable declarations. That is a sufficient condition for automatically replacing AREA_CIRCLE with a constexpr declaration. The resulting program is:

```
constexpr auto R = 10;
constexpr auto PI = 3.14;
constexpr auto AREA_CIRCLE = PI * R * R;
```

Note that if the order of definitions was such that the replacement would produce a compiler error, we choose to preserve the original macros. Definition reordering is best done in an assisted manner.

In all other cases, transformations must be considered carefully. Here, we give some examples where transformations are possible, but should be guided by the programmer. Consider the macro definition that refers to a function.

```
#define PRINT printf
```

This macro can be rewritten as a declaration.

```
auto PRINT = printf;
```

An automated transformation is possible only if printf is known to name a function declaration or some other global declaration. In other words, the transformation relies on compiler knowledge. The type of the declarator depends on the declaration. While the declaration above is suitable for functions, an alias to a global variable would most likely need to a reference. Declarations of this sort are not declared constexpr because we do not want them to be evaluated at compile time.

Consider a the following use of an expression alias macro.

```
#define SUM a + b

void summer()
{
  int a = 1, b=2;
  int c = SUM;
}
```

The dependent macro refers to identifiers that are actually variable declarations in a future scope. There is a possible transformation that can be applied to remove the macro, but its automated application would require information from a C++ front end about the variables referenced by the expanded macro and the relative locations of the variable declarations, the macro definition, and the use of the macro. Knowing all of this, the corresponding transformation is to replace SUM with a lambda function declared in the same scope as the local variables:

```
void summer()
{
  int a = 1, b=2;
  auto SUM = [&a, &b]() { return a + b; };
  int c = SUM();
}
```

Here, SUM is declared as a lambda function that captures its non-local arguments by reference. Applying the transformation also requires modification at the call site. We need to invoke SUM as a function since it no longer expands to a sequence of tokens.

Obviously, this is a complex transformation, and one that we have not fully implemented. Our demacrofication tool is capable of generating the replacement declaration, but we have not yet calculated where the it should be placed in the resulting code; the tool emits a note giving the transformed macro, and suggesting the context in which it might be placed. This allows the user to perform the transformation manually with assistance from our tools.

B. Type Alias

A type alias is an object-like macro whose replacement text can be recognized as a C++ type expressions. For example:

```
#define INT_VEC vector<int>
#define UINT unsigned int
#define UINT_PTR UINT*
```

We generally consider type aliases to be dependent even though type expressions like unsigned int obviously refer to a built-in types and could be considered closed. The distinction matters little for this set of macros. Dependent or not, each of these declarations represents a valid type expression, which can easily be modeled as a C++11 alias declaration. The general transformation for a type alias is:

```
// macro
#define A T

// C++11 declaration
using A = T;
```

Here, A is declared as an alias to the type expression T. We prefer to using declarations to typedefs because a) the syntax more clearly delineates the new type name from the aliased type expression, and b) using declarations can be parameterized over type arguments [18], a fact that we use to transform function-like macros that define parameterized type expressions (Section V-D).

This, the declarations corresponding to the macros above are:

```
using INT_VEC = vector<int>;
using UINT = unsigned int;
using UINT_PTR = UINT*;
```

Note that we have not implemented this transformation since recognizing type expressions is not straightforward. We include the mapping for completeness and reserve the implementation for future work.

C. Parameterized Expression

A parameterized expression is a function-like macro that expands to an expression or a statement. For example:

```
#define MIN(A, B) ((A) < (B) ? (A) : (B))
#define ASSIGN(A, B) { B = A; }
```

The `MIN` macro is typical of inline functions written using the C preprocessor. This classification of macros is used by both Ernst et al. [3] and Mennie and Clarke [7].

The `ASSIGN` macro is different. Although it is a parameterized compound statement, it does not compute a value. It performs an operation resulting in a side effect. This can be still be implemented as an inline function, but we need to be sure that any such argument `B` is passed by reference.

The mapping for the *closed* macros like `MIN` whose replacement text is an expression (but not a statement) is:

```
// macro
#define F(A1, ..., An) X

// C++11 declaration
template <typename T1, ..., typename Tn>
auto F(T1&& A1, ..., Tn&& An)
  -> decltype(X)
{
  return decltype(X);
}
```

Here, `F` is a function-like macro with n arguments and replacement text `X`. Because macro arguments are untyped, we must allow expressions of any conceivable type to be used as arguments to the function. Templates are capable of providing this level of flexibility. As such, the resulting C++11 declarations is a function template with n distinct template (type) parameters ($\mathtt{T}_i$) corresponding to each of the n function parameters ($\mathtt{A}_i$).

The result type of the function uses the new late-binding syntax for function return types. The result type, written as `decltype(X)`, is the deduced type of the expression `X`. Without the ability to deduce the type of the resulting expression, an automated, generic replacement would not be possible. We would have to investigate all possible uses of the macro in order to determine an appropriate result, if one existed.

The function parameters are passed by *forwarding* into the function body. Forwarding means that the actual type of the instantiated function parameter will be determined by the type of the function argument. Of all new C++11 features, perfect forwarding is the least obvious and least well understood. However, it suits this application perfectly. Consider a possible use of the replaced function `F` (assuming `F` takes only 2 arguments).

```
int x = 0;
F(2, x);
```

Because the arguments are forwarded, the deduced template argument types will preserve qualifiers on the type arguments. The templat argument type deduced for `T1` will be `int` because `2` is a literal having type `int` (there are no qualifiers). The type deduced for `T2` will be `int&` because `x` is an lvalue (it refers to a non-const variable). In other words, the specialization of `F` that is ultimately called after instantiation will be:

```
auto F<int, int&>(int A1, int& A2)
  -> decltype(X)
```

The exact mapping for the `MIN` macro is:

```
template <typename T1, typename T2>
inline auto MIN(T1&& A, T2&& B)
  -> decltype(((A) < (B) ? (A) : (B)))
{
  return ((A) < (B) ? (A) : (B));
}
```

Although an automated replacement can be used exactly like the original, we have introduced some additional complexity into the definition that we would like to remove: the extra parentheses, the deduced result type, the use of perfect forwarding. The *demacrofier* working in tandem with a compiler should be capable of fully replacing the macro with a function declared in the traditional style.

The mapping for *closed* macros like `ASSIGN` whose replacement text is classified as a statement is:

```
// macro
#define F(A1, ..., An) S

// C++11 declaration
template <typename T1, ..., typename Tn>
void F(T1&& A1, ..., Tn&& An)
{
  S;
}
```

Because a statement has no result type we can omit the deduced result type specification and the `return` statement. We simply construct a `void` function with the same pattern of template and function parameters above. The final mapping of the `ASSIGN` algorithm is:

```
template <typename T1, typename T2>
inline void ASSIGN(T1&& A, T2&& B) {
  B = A;
}
```

This illustrates another reason why perfect forwarding is so well-suited to this task. First, we cannot detect the presence of side effects in the replacement text of a macro. If an argument is modified, then it needs to be passed by lvalue reference. Second, we do not know how macros are used in the program without substantial cooperation from the front end. We do not know if arguments are intended to be passed by value, reference, or constant expression. Perfect forwarding allows us to defer decisions about parameter passing by adapting to the usage at the call site.

Finally, the use of inlining in both mappings helps ensure that the performance will be no worse than using the original macro.

A parameterized expression can be *dependent* (having free variables). In these cases, as with expression aliases, a transformation must consider the properties of the free variables in the expression. Consider the following:

```
#define OP(A, B) (X +
    MAX((A),(B))-MIN((A),(B)))
```

Here, `X`, `MIN` and `MAX` are free variables. If we know that `X` is an expression alias mapped to a declaration, and we know that `MIN` and `MAX` are parameterized expressions mapped to function templates, then the mapping is straightforward:

```
template <typename T1, typename T2>
inline auto OP(T1&& A, T1&& B)
  -> decltype((X + MAX((A),(B))-MIN((A),(B))))
{
  return (X + MAX((A),(B))-MIN((A),(B)));
}
```

This is similar to the analysis applied to dependent expression aliases. When free variables are not macros, then we must rely on information from the compiler. Consider a 2nd example:

```
#define Acc(a, b) { a += f(b); }
```

Before doing any modification we will have to ascertain certain facts about `f` and the context in which `Acc` is called. If `f` is a function or class type (i.e., `f(b)` invokes a constructor), then we can proceed in the usual fashion.

```
template <typename T1, typename T2>
inline void Acc(T1&& a, T2&& b) {
  a += f(b);
}
```

We note that the signature could be improved by examining the arguments accepted by `f`. For example, if `f` only takes its argument by value, then the function argument `b` could also be passed by value. If `f` is polymorphic, taking `const` and non-`const` arguments, then the forwarding approach is more appropriate.

If `f` is a local function object or lambda expression in the context in which `Acc` is called, then we would have to transform `Acc` into a lambda function or function object. The mechanism for doing so, and the problems inherent in the transformation, are similar to those involved in the transformation of expression aliases involving lambda function transformations.

In either case, the requirement for compiler knowledge places these transformations out of the scope of a straightforward, fully automated transformation. We defer to the user in these cases.

D. Parameterized Type Alias

A function-like macro whose replacement text can be recognized as type expression is a parameterized type alias. For example:

```
#define PTR_TYPE(T) T*
#define ValueType(I) \
      typename value_type<I>::type;
```

We note that this last example is taken from *Elements of Programming* where such macros are used to implement type functions related to concepts [19]. This is one case where macros are being used to implement "cutting edge" ideas about principled generic programming. Fortunately, there is an equivalent C++11 declaration (alias templates) that allows us to avoid macros with this style of programming.

As with non-parameterized type aliases, the general transformation is achieved using C++11 alias declarations.

```
// macro
#define A(T1, ..., Tn) X

// C++11 declaration
template <typename T1, ..., typename Tn>
using A = X;
```

The macro parameters T_i are mapped to corresponding template parameters of an alias declaration. No additional substitution is needed for `X` since the replacement text is written as the resulting type. The declarations corresponding to macros presented initially are:

```
template <typename T>
using Ptr = T*;

template <typename T>
using ValueType =
  typename value_type<T>::type;
```

As with type aliases, dependent names in parameterized type aliases are to be expected. However, because their resolution requires knowledge from the C++ compiler, we have not implemented their automated transformation. We include them here because we believe them to be an important feature of demacrofication and intend to support their transformation fully in the future.

VI. IMPLEMENTATION

Our implementation is composed of three separate tools that support the different stages of demacrofication shown in Figure 1. The `cpp2cxx-suggest` implements a straightforward translation the macros into equivalent C++ according to the criteria described in the previous sections. When automated transformations are not possible or obvious, suggestions to the user. The result of the application of this tool is a collection of intermediate source code and configuration files.

The `cpp2cxx-validate` program iteratively manipulates suggested transformations to find the largest possible set that results in a functioning build. The `cpp2cxx-finalize` program strips intermediate configuration and modifications to the source code and generates the final build. These two tools are discussed in more detail in Section VII-C.

The `cpp2cxx-suggest` program implements the initial phases of demacrofication. The basic process by which demacrofication happens is shown in Figure 3.

The goal of this process is to determine if there is a C++11 declaration that can be used to replace the definition of a macro. Ultimately, we must make a decision to preserve a macro in the original program or replace it.

The program iteratively considers macros in the order in which they appear in the source text. Configurational macros, and those whose replacement texts are partial syntax trees are preserved (i.e., no transformation is applied).

The classification step attempts to determine if the replacement text of the macro corresponds to a complete C++11

Fig. 3: The demacrofication process is a decision procedure that determines if a macro can be replaced with a C++11 declaration.

expression, statement, declaration, or type. The current implementation is a best-effort attempt to parse the sequence of tokens in the replacement text in the absence of any C++ information. Because it operates on partial information, it is not as accurate as we would like.

If the macro is dependent, the tool performs dependency analysis to determine if a mapping exists. If the dependencies can be resolved (i.e., finding previously defined macros that map to declarations), the the transformation can be applied. Otherwise, the macro is preserved, although the tool will emit a suggestion for the possible transformation.

VII. EVALUATION

Our evaluation consists of three distinct activities.

- First, we applied the `cpp2cxx-suggest` tool to a 7 different C++ libraries in order to ascertain the extent to which those libraries might be improved through demacrofication.
- Second, we selected one of those libraries and migrated it to C++11, acting as the user of our demacrofication tools. The goal of this study was to understand, in detail, the impact of our automated refactoring on a library and the amount of effort required to complete the process.
- Finally, we selected two of the remaining libraries as targets for our automated validation method. The goal of this experiment was to determine the extent to which we could reduce the amount of user intervention in the demacrofication process.

We describe each activity separately.

A. Automatic Demacrofication

In the first experiment we applied our demacrofication tool to seven C++ libraries comprising over 1.5 million lines of non-comment, non-blank code. The purpose of the experiment is to estimate the number of macros that might possibly be refactored, either automatically or assisted by the programmer. In other words, we are measuring the extent to which the program might be improved (made more maintainable) through demacrofication.

The `cpp2cxx-suggest` program is applied to each source code file in a library. Files including non-C/non-C++ code (e.g., lex/yacc files) were excluded from the test. The test was configured such that every transformation that could be identified was applied. The results of the demacrofication *were not compiled* to test if the automated transformation would preserve the build; we knew in advance that a naïve application of the tool would most likely result in a broken build. We are only interested in the extent to which macros might be removed from a library.

From the Table-I it is clear that we can remove a significant number of macros which are neither empty nor partial. Hypothetically, all macros in the ***closed*** and ***dependent*** category can be replaced using one of the transformations described in Section-VI. The results, however, are not quite 100% for these cases because of a couple of issues. Definition ordering issues prevent automatic refactoring, as do macros that would be refactored as lambda functions. Also, we have not yet implemented support for demacrofying type aliases.

B. Case Study

In this experiment we took one library (Crypto++-5.6.1) and migrated it use the C++11 programming language. We used the `cpp2cxx-suggest` tool to suggest and apply an initial set of transformations. When the transformation was not obvious or would break the build, we intervened and applied a correct transformation by hand.

There were three primary goals of the study. First, we wanted to identify the causes of compilation errors resulting from automatic transformation. Second, we want to analyze those errors to determine how we might improve our demacrofication tools by understanding the interaction between preprocessor and source code declarations. Third, we wanted to gauge the amount of effort required to fully demacrofy a library.

As a result of the study, we found three situations where we were able to correct the demacrofication manually. These were later incorporated into the automated demacrofication tool and implemented to produce suggestions (not automatic transformations).

We found two instances where function-like macros were being defined inside a function.

```
void f() {
#define S0(X) S[X]
   int x = 10;
   x += S0(x);
}
```

If we simply generate a new function definition, we will produce an obviously incorrect program; C++ does not permit function declarations inside other functions. However, the macro can be refactored the macro as a lambda function. To

TABLE I: Results before validation

			Preserved		Complete		
Package Name	KSLOC (NCNB) [1]	Total Macros	Empty Macros	Partial Macros	Closed	Dependent	Actually Demacrofied [2]
Cryoto++-5.6.1	55	1020	164	373	300	183	457 (94.6%)
p7zip-9.20.1	96	1098	284	119	660	35	656 (94.3%)
scintilla-3.0.4	66	2694	52	15	2588	39	2595 (98.7%)
poco-1.4.3p1	144	2564	889	305	857	513	987 (72.0%)
facebook-hiphop-php-git	544	4951	1033	934	2538	446	2591 (86.8%)
wxWidgets-2.9.3	741	19262	2483	1923	11223	3633	12593 (84.7%)
ACE-6.0.6	151	5969	3227	613	1587	542	1455 (68.3%)

[1] NCNB = Non-Comment, Non-Blank
[2] As a percentage of closed + dependent

do so, we need to know the types of all arguments passed to the function, and where to place the resulting declaration. For example:

```
void f() {
  int x = 10;
  auto S0 = [&S](decltype(x) X)
    { return S[X]; };
  x += S0(x);
}
```

Here, the non-local `S` is captured by reference, and the type of the function argument is given as `decltype(x)`. This is effectively the same as forwarding `x` in a function template. Determining where the declaration should be placed is not straightforward. One method has been discussed by Mennie and Clarke [7] based on the least common ancestor of each macro use. Here, however, the placement must respect both the declaration of captured local variables and the arguments used to deduce the type of `X`. Additionally, that if `S0` is invoked multiple times in the same scope but with different variables, then deducing the argument type of `X` is made much harder, but is still possible. Our modified implementation will suggest the refactoring for the lambda function but not the placement of the declaration.

There are instances of dependent macros where free variables used inside the replacement text are member variables of a `class`. For example:

```
class temp {
  public:
    int kelvin() const;
    int fahrenheit() const;
  private:
    int cel;
};

#define TEMP_KEL 273 + cel
#define TEMP_FAR cel * 9 / 5 + 32

int temp::kelvin() const
{ return TEMP_KEL; }

int temp:: fahrenheit() const
{ return TEMP_FAR; }
```

Here demacrofication will result in a compilation error because it is not possible to create a global lambda function that captures references to non- static member variables of a class. Even if we could, those members are private. Mennie et al. [7] suggest that the macro replacement be placed inside a function that used the macro:

```
int temp::get_temp_kel()
{
  constexpr auto TEMP_KEL = 273 + temp_cel;
  return TEMP_KEL;
}
```

If the macro is invoked inside multiple functions, this would lead to a replication of the macro replacement. This would be harmful to maintenance, so a better approach would be to introduce a new, `inline` member function with the correct semantics.

We modified our tool to recognize these cases and emit the different possible transformations as suggestions to the user.

Although we found and characterized only two problems of this nature, it is almost certain that there are more. We plan to continue identifying and addressing macro/code usage problems as we build and support our tools.

In addition to these problems, we also found a large number of macros that included control flow statements (`return`, `goto`, etc.), possibly nested to some level. Trying transform these into functions would break the control flow of the original program.

All told, we were able to refactor about 48.6% of the macros in the library. This is about half of what was predicted from our initial application of the `cpp2cxx-suggest` program. We only modified 15 macros; the remainder were not readily replaced with declarations. This took less than a day to complete.

Mileage will vary for each project since every project generally has its own style of macro use. The macros in Crypto++ were largely computational in nature and tended to include program fragments that could not be fully represented as C++ program elements. This will not be true for all such libraries.

C. Automatic Demacrofication with Validation

The third experiment was conducted to evaluate the extent to which we could automatically generate a functioning build. After initially demacrofying a library, it might not compile due to some of the problems discussed above. We want to revert the changes to the macros which were transformed incorrectly. To do so, we built a tool `cpp2cxx-validate` that would apply

suggested transformations one macro at a time, attempting to rebuild and retest the entire system in each iteration. The reason for building incrementally is that some macros may impact definitions in many different files.

In order to iteratively refactor macros, we wrap each generated macro in a condition that would allow it to be enabled or disabled. For example:

```
// file name is "file.cpp"
#if defined(USE_PI_filecpp_3_8)
 auto PI = 3.14;
#else
#define PI 3.14
#endif
```

The unique macro switch for the macro `PI` is `USE_PI_filecpp_3_8`. Technically, the macro should only be defined for C++11, but we omit those conditions here. A special header file, `DefinedIncludeGuards.h` contains the current list of macros introduced for each build.

A build automation script that does the following:

1) Use the 'demacrofier' to transform the program, wrapping each new declaration in the style above.
2) Create an empty file, `DefinedIncludeGuards.h`.
3) For each suggested transformation,
   a) add its macro switch to the include guard file
   b) build the library and run the test suite
   c) if the build or testing fails, remove the switch from the guard file.

When the program finally terminates—it can take many hours depending on the number of macros and size of the library—the library will be fully transformed and compiled. The order in which macro-switches were included was the same as the order the macros were demacrofied. This simple approach was taken to avoid complexity in determining the best order in which demacrofied constructs should be introduced into the program.

We applied this process to two libraries: p7zip and Scintilla. Table-II lists the total number of macros that were finally introduced into the respective libraries as compared to the total number of macros that were potentially refactorable as explained in Table-I

TABLE II: Results after validation

Package Name	Total Macros	Potentially Refactorable	Finally Validated [1]
p7zip-9.20.1	1098	695	606 (87.2%)
scintilla-3.0.4	2694	2627	2585 (98.4%)

[1] Percentage listed w.r.t. potentially refactorable macros

From the results in Table-II it is clear that some transformations could not preserve the build. This happened due to various reasons, some of which have been described above. Interestingly, the numbers here are much greater than those reported for the manual work on the Crypto++ library. We attribute this to the different styles of macro use in the libraries. The style of macro use in p7zip and scintilla is more amenable to demacrofication.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we presented an approach that can be used to demacrofy C++ programs by replacing macro definitions with C++11 declarations. We evaluated aspects of the approach and our tooling in a number of different ways. The results of the evaluation convince us that C++ programs can be effectively demacrofied, often using straightforward and simple mapping. There are, however, cases where the user is required to make an informed decision about how a macro should be replaced. We believe that the efficacy our automated approach would be greatly improved if our tools had full knowledge of the C++ program structure.

In the future, we plan to further investigate the integration of a compiler front end to supply the information needed to improve our mappings decision-making capabilities. In particular, the ability to know about C++ declarations will make it possible to automatically refactor a larger class of macros, and perhaps improve our ability to place the transformed results.

References

[1] P. Pirkelbauer, D. Dechev, and B. Stroustrup, "Source Code Rejuvenation is not Refactoring," *SOFSEM 2010: Theory and Practice of Computer Science*, pp. 639–650, 2010.

[2] B. Stroustrup, "The design and evolution of C++," *Reading, MA: Addison-Wesley,— c1994*, vol. 1, 1994.

[3] M. Ernst, G. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *Software Engineering, IEEE Transactions on*, vol. 28, no. 12, pp. 1146–1170, 2002.

[4] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landray, "Google C++ style guide," http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml, 2008.

[5] R. Seacord, *Secure Coding in C and C++*. Addison-Wesley Professional, 2005.

[6] B. Stroustrup, K. Carroll, and L. Aero, "C++ in safety-critical applications: The JSF++ coding standard," 2006.

[7] C. Mennie and C. Clarke, "Giving meaning to macros," in *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*. IEEE, 2004, pp. 79–85.

[8] C. Mennie, "Giving meaning to macros," Waterloo, Ontario, Canada, 2004.

[9] J. Gravley and A. Lakhotia, "Identifying enumeration types modeled with symbolic constants," in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*. IEEE, 1996, pp. 227–236.

[10] R. Khatchadourian, J. Sawin, and A. Rountev, "Automated refactoring of legacy java software to enumerated types," in *ICSM*. IEEE, 2007, pp. 224–233.

[11] A. Sutton and J. Maletic, "How we manage portability and configuration with the C preprocessor," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 275–284.

[12] I. D. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *WCRE*, 2001, pp. 281–290.

[13] A. Garrido and R. Johnson, "Challenges of refactoring C programs," in *Proceedings of the international workshop on Principles of software evolution*. ACM, 2002, pp. 6–14.

[14] L. Vidács, "Software maintenance methods for preprocessed languages," Ph.D. dissertation, University of Szeged, Szeged, Hungary, 2009.

[15] L. Vidács, Á. Beszédes, and R. Ferenc, "Columbus schema for c/c++ preprocessing," in *CSMR*. IEEE Computer Society, 2004, pp. 75–84.

[16] R. Stallman and Z. Weinberg, "The C preprocessor. GNU Online documentation," 2001.

[17] B. Stroustrup and G. Dos Reis, "The Pivot: A brief overview," https://parasol.tamu.edu/pivot.

[18] C++ Standards Committee and Becker, P. and others, "Programming languages-c++(final committee draft). C++ standards committee paper wg21/n3092= j16/10-0082," 2010.

[19] A. Stepanov and P. McJones, *Elements of programming*. Addison-Wesley Professional, 2009.

Open and Efficient Type Switch for C++

Yuriy Solodkyy Gabriel Dos Reis Bjarne Stroustrup

Texas A&M University
Texas, USA
{yuriys,gdr,bs}@cse.tamu.edu

Abstract

Selecting operations based on the run-time type of an object is key to many object-oriented and functional programming techniques. We present a technique for implementing open and efficient type-switching for hierarchical extensible data types. The technique is general and copes well with C++ multiple inheritance.

To simplify experimentation and gain realistic preformance using production-quality compilers and tool chains, we implement our type swich constructs as an ISO C++11 library. Our library-only implementation provides concise notation and outperforms the visitor design pattern, commonly used for type-casing scenarios in object-oriented programs. For many uses, it equals or outperforms equivalent code in languages with built-in type-switching constructs, such as OCaml and Haskell. The type-switching code is easier to use and is more expressive than hand-coded visitors. The library is non-intrusive and circumvents most of extensibility restrictions typical of visitor design pattern. It was motivated by applications involving large, typed, abstract syntax trees.

Categories and Subject Descriptors D [*1*]: 5; D [*3*]: 3

General Terms Languages, Design

Keywords Type Switch, Typecase, Visitor Design Pattern, Memoization, C++

1. Introduction

Algebraic data types as seen in functional languages are closed and their variants are disjoint, which allows for efficient implementation of case analysis on such types. Data types in object-oriented languages are extensible and hierarchical (implying that variants are not necessarily disjoint). To be general, an type-switching construct must be *open* – allow for independent extensions, modular type-checking and dynamic linking. On the other, in order to be accepted for production code, the implementation of such a construct must equal or outperform all known workarounds. However, existing approaches to case analysis on hierarchical extensible data types are either efficient or open, but not both. Truly open approaches rely on expensive class-membership testing combined with decision trees []. Efficient approaches rely on sealing either the class hierarchy or the set of functions, which loses extensibility [9, 18, 44, 51]. Consider a simple expression language:

$$exp ::= val \mid exp + exp \mid exp - exp \mid exp * exp \mid exp/exp$$

In an object-oriented language without direct support for algebraic data types, the type representing an expression-tree in the language will typically be encoded as an abstract base class, listing the (sealed set of) allowed virtual functions, with derived classes representing variants:

```
struct Expr { virtual int eval() = 0; };
struct Value : Expr { ··· int eval(); int value; };
struct Plus  : Expr { ··· Expr& e1; Expr& e2; };
```

A simple evaluator for this language can be implemented with the aid of a virtual function eval() declared in the base class Expr. The approach is *intrusive* however, as we will have to modify the base class every time we would like to add a function. Instead we offer an external introspection of objects with case analysis:

```
int eval(const Expr& e)
{
  Match(e)
    Case(const Value& x) return x.value;
    Case(const  Plus& x) return eval(x.e1) + eval(x.e2);
    Case(const Minus& x) return eval(x.e1) - eval(x.e2);
    Case(const Times& x) return eval(x.e1) * eval(x.e2);
    Case(const Divide& x) return eval(x.e1) / eval(x.e2);
  EndMatch
}
```

The syntax is provided without any external tool support or additional definitions. Instead we rely on a few C++11 features [24], template meta-programming, and macros. It runs

[Copyright notice will appear here once 'preprint' option is removed.]

about as fast as OCaml and Haskell equivalents (§4.3), and, depending on the usage scenario, compiler and underlying hardware, comes close or outperforms the handcrafted C++ code based on the *visitor design pattern* (§4).

The ideas and the library presented here were motivated by our unsatisfactory experiences working with various C++ front-ends and program analysis frameworks [1, 32, 39]. The problem was not in the frameworks per se, but in the fact that we had to use the *visitor design pattern* [18] to inspect, traverse, and elaborate abstract syntax trees to target languages. We found visitors unsuitable to express our application logic directly, surprisingly hard to teach students, and often slower than hand-crafted workaround techniques. We found users relying on dynamic casts in many places, often nested, to answer simple structural questions. That is, the users preferred shorter, cleaner, and more direct code to visitors. The consequential high performance cost was usually not discovered until later, when it was hard to remedy.

1.1 Summary

This paper makes the following contributions:

- A technique for implementing open and efficient type switching on extensible hierarchical data types as seen in object-oriented languages.
- The technique approaches the notational convenience of functional-language type-switch constructs and delivers equivalent performance to those for closed cases.
- The technique is simpler to use and outperforms the visitor pattern.
- A type-switch approach that combines subtype tests and type conversion; It delivers superior performance to approaches that combine (even constant-time) subtype tests with decision trees for even small class hierarchies.
- The type-switch construct handles multiple inheritance without workarounds.
- A constant-time function whose equivalence kernel partitions the set of objects with the same static type into equivalence classes based on inheritance path of the static type with the most-derived type.

In particular, our technique for efficient type switching:

- Comes close and often outperform various workaround techniques used in practice, e.g. visitor design pattern, without sacrificing extensibility (§4).
- Is open by construction (§3.2), non-intrusive, and avoids the control inversion typical for visitors.
- Works in the presence of multiple inheritance, both repeated and virtual, as well as in generic code (§3.7).
- Does not require any changes to the C++ object model or computations at link or load time.
- Can be used in object-oriented languages with object models similar to C++'s.
- For C++, can be implemented as a library written in ISO Standard C++11.

This is the first technique for efficiently handling type switching in the presence of general multiple inheritance. Being a library, our solution is that it can be used with any ISO C++11 compiler (e.g., Microsoft, GNU, or Clang) without requiring the installation of any additional tools or preprocessors. The solution sets a new threshold for acceptable performance, brevity, clarity and usefulness of open type-switching in C++.

2. Overview

A *heterarchy* is a partially ordered set $(H, <:)$ where H is a set of classes and $<:$ is reflexive, transitive and antisymmetric *subtyping relation* on H. We are going to use terms *class* and *type* interchangibly as the exact distinction is not important for this discussion. Given two types in a subtyping relation $D <: B$, the type D is said to be a *subtype* or a *derived class* of B, which in turn is said to be a *supertype* or a *base class* of D. When the transitive reduction $<:_d$ of $<:$ is a function, H is usually referred to as *hierarchy* to indicate single inheritance.

Subtyping effectively makes objects belong to multiple types. We call by the *most-derived type* the type used to create an object (before any conversions). By *static type* we call the type of an object as known to the compiler based on its declaration as well as any supertype of it. By *dynamic type* of an object we call any base class of the most-derived type.

2.1 Type Switch

In general, *type switch* or *typecase* is a multiway branch statement that distinguishes values based on their type. In a multi-paradigm programming language like C++ that supports in various forms parametric, ad-hoc and subtyping polymorphisms, such a broad definition subsumes numerous different type casing constructs studied in the literature [21, 23, 45]. In this work we only look at type casing scenarios based on dynamic polymorphism of C++ (nominative subtyping polymorphism based on inheritance), similar to those studied by Glew [21]. It is possible to generalize type casing (the notion, not our implementation) to static polymorphism of C++ (ad-hoc and parametric polymorphisms enabled by overloading and templates) along the line of work introduced by Harper and Morrisett [23] and studied in the context of closed and extensible solutions by Vytiniotis et al [45], but we do not address such a generalization here. We use the term *type switch* instead of a broader *typecase* to stress the run-time nature of the type analysis similar to how regular **switch**-statement of C++ performs case analysis of values at run time.

Given an object descriptor, called *subject*, of static type S (pointer or reference) referred to as *subject type*, and a list of *target types* T_i associated with the branches, a type switch statement needs to identify a suitable clause m (or absence of such) based on the most-derived type D <: S of

the subject as well as suitable conversion that csts the subject to the target type T_m. Due to multiple inheritance, types T_i in general may or may not be derived from S, however, because of the strong static type safety requirement, the type of applicable clause T_m will necessarily have to be one of subject's dynamic types: D <: T_m. A hypothetical type switch statement, not currently supported by C++, may look as following:

switch (subject) { **case** T_1: s_1; ... **case** T_n: s_n; }

There is no need for an explicit *default clause* in our setting because such a clause is semantically equivalent to a case clause guarded by the subject type: **case** S: s. The only semantic difference such a choice makes is in the treatment of null-pointers, which, one may argue, should be handled by the default clause. We disagree, because not distinguishing between invalid object and valid object of a known static but unknown dynamic type may lead to some nasty run-time errors.

Similar control structures exist in many programming languages, e.g. *match* in Scala [35], *case* in Haskell [25] and ML [33], *typecase* in Modula-3 [6] and CLOS [**?**] (as a macro), *tagcase* in CLU [30], *union case* in Algol 68 and date back to at least Simula's *Inspect* statement [11]. The statement in general can be given numerous plausible semantics:

- *First-fit* semantics will evaluate the first statement s_i such that T_i is a base class of D
- *Best-fit* semantics will evaluate the statement corresponding to the most-derived base class T_i of D if it is unique (subject to ambiguity)
- *Exact-fit* semantics will evaluate statement s_i if $T_i = D$.
- *All-fit* semantics will evaluate all statements s_i whose guard type T_i is a subtype of D (order of execution has to be defined)
- *Any-fit* semantics might choose non-deterministically one of the statements enabled by all-fit

The list is not exhaustive and depending on a language, any of these semantics can be a plausible choice. Functional languages, for example, often prefer first-fit semantics because it is similar to case analysis in mathematics. Object-oriented languages would typically be inclined to best-fit semantics due to its similarity to overload resolution and virtual dispatch, however, some do opt for first-fit semantics to mimic the functional style: e.g. Scala [35]. Exact-fit semantics can often be seen in languages supporting discriminated union types: e.g. variant records in Pascal, Ada and Modula-2, oneof and variant objects in CLU, unions in C and C++ etc. All-fit and any-fit semantics might be seen in languages based on predicate dispatching [17] or guarded commands [13], where a predicate can be seen as a characteristic function of a type, while logical implication can be seen as subtyping.

2.2 Expression Problem

Type switching is related to a more general problem manifesting the differences in functional and object-oriented programming styles.

Conventional algebraic datatypes, as found in most functional languages, allow for easy addition of new functions on existing data types. But they fall short in extending data types themselves (e.g. with new constructors), which requires modifying the source code. Object-oriented languages, on the other hand, make data type extension trivial through inheritance; but the addition of new functions operating on these classes typically requires changes to the class definition. This dilemma is known as the *expression problem* [10, 46].

Classes differ from algebraic data types in two important ways. Firstly, they are *extensible*, for new variants can be added later by inheriting from the base class. Secondly, they are *hierarchical* and thus typically *non-disjoint* since variants can be inherited from other variants and form a subtyping relation between themselves [21]. In contrast, variants in algebraic data types are *disjoint* and *closed*. Some functional languages e.g. ML2000 [2] and its predecessor, Moby, were experimenting with *hierarchical extensible sum types*, which are closer to object-oriented classes then algebraic data types are, but, interestingly, they provided neither traditional nor efficient facilities for performing case analysis on them.

Zenger and Odersky later refined the expression problem in the context of independently extensible solutions [50] as a challenge to find an implementation technique that satisfies the following requirements:

- *Extensibility in both dimensions*: It should be possible to add new data variants, while adapting the existing operations accordingly. It should also be possible to introduce new functions.
- *Strong static type safety*: It should be impossible to apply a function to a data variant, which it cannot handle.
- *No modification or duplication*: Existing code should neither be modified nor duplicated.
- *Separate compilation*: Neither datatype extensions nor addition of new functions should require re-typechecking the original datatype or existing functions. No safety checks should be deferred until link or runtime.
- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

While these requirements were formulated for extensible data type with disjoint variants, object-oriented languages primarily deal with hierarchical data types. We thus found it important to explicitly state an additional requirement based on the Liskov substitution principle [29]:

- *Substitutability*: Operations expressed on more general data variants should be applicable to more specific ones that are in a subtyping relation with them.

We will refer to a solution that satisfies all of the above requirements as *open*. Numerous solutions have been proposed to dealing with the expression problem in both functional [19, 31] and object-oriented camps [22, 27, 37, 49], but very few has made its way into one of the mainstream languages. We refer the reader to Zenger and Odersky's original manuscript for a discussion of the approaches [50]. Interestingly, most of the discussed object-oriented solutions were focusing on the visitor design pattern and its extensions, which even today seems to be the most commonly used approach to dealing with the expression problem in object-oriented languages.

A lot has been written about the visitor design pattern [18, 36, 37, 49]. Its advantages include *extensibility of functions*, *speed*, and, *being a library solution*. Nevertheless, the solution is *intrusive*, *specific to hierarchy*, and requires a lot of *boilerplate code* to be written. It also introduces *control inversion*, and, most importantly, – *hinders extensibility* of classes.

2.3 Open Type Switch

Note that the presence of a type switch in an object-oriented language alone does not solve the expression problem because the existing code may have to be modified to take new variants into account. Relying on default clause is not considered to be an acceptable solution in this context, because often times the only reasonable default behavior is to raise an exception. Zenger and Odersky note that in such cases defaults will transform type errors that should manifest statically into runtime exceptions that are thrown dynamically [50].

While we generally agree with this observation, we would like to point out that in our experience newly added variants were more often extending an existing variant than creating an entirely disjoint one. In a hypotetical compiler, for example, a new kind of type expression will typically extend a TypeExpression variant, while a new form of annotation will extend an Annotation variant, thus not extending the root ASTNode directly. Due to substitutability requirement such a new variant will be treated as a variant it extends in all the existing code. The functions that will be affected by its addition and thus have to be modified will be limited to functions directly analyzing the variant it extends and not providing a default behavior.

To account for this subtlety of extensible hierarchical data types, we use a term *open type switch* to refer to a type switch that satisfies all the requirements of an *open solution to expression problem* stated above except for the *no modification or duplication* requirement. We loosen it to allow modification of functions for which the newly added variant becomes a disjoint (orthogonal) case not handled by default clause. We believe that the loosened requirement allows us to express pragmatically interesting restrictions that developers are willing to live with. Besides, open type switch overcomes all the major shortcomings of the visitor design pattern:

- Case analysis with an open type switch is *non-intrusive* as it inspects the hierarchy externally and can be applied retroactively.
- New variants can be accounted for in the newly written code and will be seen as a base class or default in the existing code.
- The affected functions are limited to those for which the newly added variant is a disjoint case.
- The code avoids the control inversion and the need for boilerplate code that visitors introduce, and is thus a more direct expression of the intent.

2.4 C++ Specifics: Subobjects

C++ supports two kinds of inheritance: *non-virtual* [15] (also known as *replicated* [40] or *repeated* [47]) inheritance and *virtual* [15] (or *shared* [47]) inheritance. The difference between the two only arises in situations where a class indirectly inherits from the same base class via more than one path in the hierarchy. Different kinds of inheritance give raise to the notion of *subobject* in C++, which are then used to define semantics of operations like casts, virtual function dispatch etc. We give an informal introduction to them here in order to show some subtleties of the C++ inheritance model, which must be taken into account when addressing type switching or subtype testing.

Figure 1. Multiple Inheritance in C++

Consider a simple class hierarchy in Figure 1(1). Class D indirectly inherits from class A through its B and C base classes. In this case, the user may opt to keep distinct subobjects of class A (repeated inheritance) or a shared one (virtual inheritance) by specifying how B and C are inherited from

A. The kind of inheritance is thus not a property of a given class, but a property of an inheritance relation between derived and base class and it is possible to mix the two in an object of the most-derived type.

A class hierarchy, i.e. an inheritance graph gives rise to a subobject graph, where a given class node may be replicated when inherited repeatedly or left shared when inherited virtually. The edges in such a graph represent subobject containtment and are marked with whether such containtment is shared or exclusive. Every class C in the class hierarchy will have its own subobject graph representing the subobject of an object of the most-derived type C. Figure 1(2) shows subobject graph for class D obtained for the class hierarchy in (1) under repeated (a) and virtual (b) inheritance of class A by classes B and C. The shared containment is indicated with the dashed arrows, while exclusive with the solid ones.

We will use term *object descriptor* to mean either pointer or reference to an object, which we will use interchangably when not explicitly specified. An object descriptor of static type A referencing an object of the most-derived type C can be understood as any $*::A$-node in the subobject graph of C. Rosie and Friedman call A an *effective type* of object, while the node in the subobject graph representing it – its *effective subobject*. Casts in such a model can be understood as a change from one effective subobject to another. We will use terms *source subobject* and *target subobject* to refer to effective subobjects before and after the cast. Their static types will be refered to as as *source type* and *target type* respectively. C++ distinguishes 3 kinds of casts: upcasts, downcasts and crosscasts.

An *upcast* is a cast from a derived class to one of its bases. When the base class is unambiguous, such casts are implicit and require no additional annotations. When the base class is ambiguous, cast failure is manifested statically in a form of a compile-time error. This is the case for example with casting D to A under repeated multiple inheritance of A, in which case the user needs to explicitly cast the object to B or C first in order to indicate the desired subobject and resolve ambiguity. In some cases, however, introduction of such an explicit cast is not possible: e.g. in implicit conversions generated by the compiler to implement covariant return types, cross casts or conversions in generic code. This does not mean that in such cases we violate the Liskov subtitution principle though – the classes are still in subtyping relation, but an implicit conversion is not available.

A *downcast* is a cast from a base class to one of its derived classes. The cast has to determine at run-time whether the source subobject is contained by a subobject of the target type in the most-derived type's subobject graph. Failure of such a cast is manifested dynamically at run-time.

A *crosscast* is a cast between classes that are not necessarily related by inheritance. Accordingly to the C++ semantics such cast is defined to be a composition of upcast to target type and downcast to the most-derived type. While the downcast to the most-derived type is always guaranteed to succeed regardless of the source subobject, the upcast to the target type may be ambiguous, in which case the cast will fail. A cast from Y to B inside an object of the most-derived type D in Figure 1(2a,2b) will be an example of a successful cross cast. A similar cast from Y to A inside D under repeated inheritance of (2a) will fail because of ambiguous upcast from D to A.

An interesting artefact of these distinctions can be seen on an example of casting a subobject of type Z to a subobject of type A in Figure 1(2a). The subobject D::B::A::Z will be successfully cast to D::B::A, while the D::C::A::Z will be successfully cast to D::C::A. These casts do not involve downcasting to D followed by an upcast to A, which would be ambiguous, but instead take the dynamic type of a larger subobject (D::B or D::C) the source subobject is contained in into account in order to resolve the ambiguity. A similar cast from Y to A will fail and should Y have also been non-virtually derived from Z, the cast from D::C::Y::Z to A would have failed. This shows that the distinction between crosscast and downcast is not based solely on the presence of a subtyping relation between the source and target types, but also on the actual position of the source subobject in the most-derived type's subobject graph.

C++ inheritance model, presented here informally, complicates the semantics and the implementation of a type switch further. On one side we have to define the semantics of a type switch when the cast between source and target types that are in subtyping relation is not possible. On the other – an implementation of the cast between source and target subobjects will have to take into account the location of the source subobject in the subobject graph into account in addition to the most-derived and target types on which a simple subtype test would solely depend.

2.5 Previous Work

The closed nature of algebraic data types allows for their efficient implementation. The traditional compilation scheme assigns unique (and often small and sequential) tags to every variant of the algebraic data type and type switching is then simply implemented with a multi-way branch [42] (usually a jump table) over all the tags [3]. Dealing with extensible hierarchical data types makes this extremely efficient approach infeasible:

- *Extensibility* implies that the compiler may not know the exact set of all the derived classes till link-time (due to *separate compilation*) or even run-time (due to *dynamic linking*).
- *Substitutability* implies that we should be able to match tags of derived classes against case labels representing tags of base classes.
- Presence of *multiple inheritance* might require pointer adjustments that are not known at compile time (e.g. due

to virtual base classes, ambiguous base classes or cross-casting).

There are two main approaches to implementing case analysis on extensible hierarchical data types, discussed in the literature.

The first approach is based on either explicit or implicit sealing of the class hierarchy, on which type switching can be performed. In Scala, for example, the user can forbid future extensions from a given class hierarchy through the use of a sealed keyword [16, §4.3.2]. The compiler then uses the above tag allocation over all variants to implement type analysis. In some cases the sealing may happen implicitly. For example, languages that allow names with internal and external linkage may employ the fact that classes with internal linkage will not be externally accessible and thus effectively sealed. While clearly efficient, the approach is not open as it avoids the question rather than solves.

The broader problem with this approach is that techniques that rely on unique or sequential compile or link-time constants violate independent extensibility since without a centralized authority there is no guarantee same constant will not be chosen in type unsafe manner by independent extensions. Updating such constants at load time may be too costly even when possible. More often than not however such updates may require code regeneration since decision trees, lookup tables etc. may have been generated by compiler for given values.

An important practical solution that follows this approach is the visitor design pattern [18]. The set of visit methods in visitor's interface essentially seals the class hierarchy. Extensions have been proposed in the literature [49], however they have problems of their own, discussed in §5.

The second approach employs type inclusion tests combined with decision trees [5] to avoid duplicate checks. The efficiency of the approach is then entirely focused on the efficiency of type inclusion tests [7, 9, 12, 14, 20, 26, 41, 44, 48, 51].

Type inclusion tests for single inheritance were initially implemented by traversing a linked list of types, as proposed by Wirth [48]. Such encoding requires little space, but runs in time proportional to the distance between the two types in the class hierarchy. A trivial constant-time type inclusion test can be achieved with a *binary matrix*, encoding the subtyping relation on the class hierarchy [12]. While efficient in time, it has quadratic space requirements, which makes it expensive for use on large class hierarchies. In response to Wirth' original publication, Cohen proposed the first space-efficient constant-time algorithm, which, howver, could only deal with single inheritance [9]. *Hierarchical encoding* is another constant-time test that maps subtype queries into subset queries on bit-vectors [7, 26]. The approach can handle multiple inheritance, but the space and time required for a subtype test in this encoding increases with the size of the class hierarchy, also Caseau's approach is limited to class hierarchies that are lattices. Schubert's *relative numbering* [41] encodes each type with an interval $[l, r]$, effectivelly making type inclusion tests isomorphic to a simple range checking. The encoding is optimal in space and time, however it is limited to single inheritance. *PQ-Encoding* of Zibin and Gil employs PQ-trees to improve further space and time efficiency of the constant-time inclusion testing [51]. While capable of handling type inclusion queries on heterarchies, the approach makes the closed world assumption and can be costly for use with dynamic linking because it is not incremental. The approach of Gibbs and Stroustrup [20] employs divisibility of numbers to obtain a constant-time type inclusion test. The approach can handle multiple inheritance and was the first constant-time technique to addresses the problem of casts between subobjects. Unfortunately the approach limits the size of the class hierarchies that can be encoded with this technique. Ducournau proposed constant-time inclusion test based on the fact that in an open solution a class has known amount of base classes and thus perfect hashes can be used to map them to this-pointer offsets typically used to implement subobject casts[14]. Unfortunately the approach addresses only virtual multiple inheritance and similarly to other approaches relies on load-time computations that may be costly. Detailed analysis and explanation of existing constant-time type inclusion tests can be found in [44] and [51].

With the exception of work by Gibbs and Stroustrup [20], all the approaches to efficient type-inclusion testing we found in the literature were based on the assumption that *the outcome of a subtyping test as well as the subsequent cast depend only on the target type and the most-derived type of the object*. While such assumption is sound for subtyping tests and subtype casts for shared inheritance (including single), it does not reflect the relationship between subobjects in the general case multiple inheritance present in C++.

2.6 The Source of Inefficiency

While constant-time type inclusion tests are invaluable in optimizing subtype tests in programing languages, their use in implementing a type switch is inferior to some workaround techniques. This may prevent wide adoption of a language implementation of such a feature due to its inferior performance. We implemented 3 constant-time type inclusion tests: binary matrix **??**, Cohen's algorithm [9] and fast dynamic cast [20] and combined them with a decision tree to implement a type switch on a class hierarchy ideally suited for such scenario. The class hierarchy used in this comparison was a perfect binary tree with classes number $2i$ and $2i + 1$ derived from a class number i. Our workaround techniques included visitor design pattern and a switch on the sealed sequential set of tags.

The chart in Figure 2 shows the time (Y-axis) each technique took to recognize an object of the most-derived type i (X-axis). It is easy to see that the logarithmic cost associated with the decision tree very quickly surpasses the constant

Figure 2. Type switch based on const-time type inclusion tests

overhead of double dispatch present in the visitor design pattern or the jump-table implementation of the switch on all tags. The edgy shape of timing results reflects the shape of the binary tree class hierarchy used for this experiment.

3. Type Switch

C++ does not have direct support of algebraic data types, but they can be encoded with classes in a number of ways. One common such encoding is to introduce an abstract base class representing an algebraic data type with several derived classes representing variants. The variants can then be discriminated with either run-time type information (referred to as *polymorphic encoding*) or a dedicated member of a base class (referred to as *tagged encoding*).

While our library supports both encodings, it handles them differently to let the user choose between openness and efficiency. The type switch for tagged encoding is simpler and more efficient for many typical use cases, however, making it open will eradicate its performance advantages. The difference in performance is the price we pay for keeping the solution open. We describe pros and cons of each approach in §4.2.

3.1 Attractive Non-Solution

While Wirth' linked list encoding was considered slow for subtype testing it can be adopted for very efficient type switching. The idea is to combine the fast switching on closed algebraic datatypes with a loop that tries the tags of base classes when switching on derived tags fails.

For simplicity of presentation we assume a pointer to array of tags be available directly from within object's taglist data member. The array is of variable size, its first element is always the tag corresponding to the most-derived type of the object, while its end is marked with a dedicated end_of_list marker distinct from all the tags. The tags in between are topologically sorted according to the subtyping relation with incomparable siblings listed in *local precedence order* – the order of the direct base classes used in the class definition. We call such a list a *Tag Precedence List* (TPL) as it resembles the *Class Precedence List* (CPL) of object-oriented descendants of Lisp (e.g. Dylan, Flavors, LOOPS, and CLOS) used there for *linearization* of class hierarchies. TPL is just an implementation detail and the only reason we distinguish TPL from CPL is that in C++ classes are often separated into interface and implementation classes and it might so happen that the same tag is associated by the user with an interface and several implementation classes. We also assume the tag-constant associated with a class D_i be accessible through a static constant D_i::class_tag. These simplifications are not essential and the library does not rely on any of these assumptions. Instead the user can retroactively narrate to the library the specific tag encoding used throught a trait-like classes.

A type switch below, built on top of a hierarchy of tagged classes, proceeds as a regular switch on the subject's tag. If the jump succeeds, we found an exact match; otherwise, we get into a default clause that obtains the next tag in the tag precedence list and jumps back to the beginning of the switch statement for a rematch:

```
    size_t attempt = 0;
    size_t tag = object→ taglist[attempt];
ReMatch:
    switch (tag) {
    default:
        tag = object→ taglist[++attempt];
        goto ReMatch;
    case end_of_list:
        break;
    case D1::class_tag:
        D1& match = static_cast⟨D1&⟩(*object);
        s1;break;
        ...
    case Dn::class_tag:
        Dn& match = static_cast⟨Dn&⟩(*object);
        sn;break;
    }
```

The above structure, which we call *Tag Switch*, lets us dispatch to case clauses of the most-derived class with an overhead of initializing two local variables, compared to efficient switch used on algebraic data types. Dispatching to a case clause of a base class will take time roughly proportional to the distance between the matched base class and the most-derived class in the inheritance graph, thus the technique is not constant. When none of the base class tags were matched, we will necessarily reach the end_of_list marker in the tag precedence list and exit the loop. As mentioned before, the default clause of the type switch can be implemented with a case clause on subject type's tag: **case** S::class_tag:

The efficiency of the above code crucially depends on the set of tags we match against be small and sequential to justify the use of jump table instead of decision tree to implement the switch. This is usually not a problem in closed hier-

archies based on tag encoding since the user of the hierarchy hand-picks himself the tags. The use of static cast to obtain proper reference once the most specialized derived class has been established, however, essentially limits the use of this mechanism to single inheritance only. This of course only refers to the way target classes inherit from the subject type – they can freely inherit other classes as long as they do not create repeated of virtual multiple inheritance of the subject type. Due to these assumptions, the technique is not open because it may violate independent extensibility. Moving away from these assumptions in order to make the technique more open (e.g. randomizing tags, using **dynamic_cast** etc.) will also eradicate its performance advantages.

3.2 Open but Inefficient Solution

Instead of starting with an efficient solution and trying to make it open, we start with an open solution and try to make it efficient. The following cascading-if statement implements the first-fit semantics for our type switch in a truly open fashion:

if (T_1* match = **dynamic_cast**$\langle T_1* \rangle$(subject)) { s_1;} **else**
if (T_2* match = **dynamic_cast**$\langle T_2* \rangle$(subject)) { s_2;} **else**
...
if (T_n* match = **dynamic_cast**$\langle T_n* \rangle$(subject)) { s_n;}

Its main drawback is performance: a typical implementation of **dynamic_cast** takes time proportional to the distance between base and derived classes in the inheritance tree. What is worse, is that the time to uncover the type in the ith case clause is proportional to i, while failure to match will always take the longest. In a test involving a flat hierarchy of 100 variants it took 93 cycles to discover the first type and 22760 to discover the last (with linear combination of those times to discover the types in between). A visitor design pattern could uncover any type in about 55 cycles, regardless of its position among the case clauses, while a switch based on sequential tags could achieve the same in less than 20 cycles. The idea is thus to combine the openness of the above structure with the efficiency of a jump table on small sequential values.

3.3 Memoization Device

Let us look at a slightly more general problem than type switching. Consider a generalization of the switch statement that takes predicates on a subject as its clauses and executes the first statement s_i whose predicate is enabled:

switch (x) { **case** P_1(x): s_1; ... **case** P_n(x): s_n; }

Assuming that predicates depend only on x and nothing else as well as that they do not involve any side effects, we can be sure that the next time we come to such a switch with the same value, the same predicate will be enabled first. Thus, we would like to avoid evaluating predicates and jump straight to the statement it guards. In a way we would like the switch to memoize which case is enabled for a given value of x.

The idea is to generate a simple cascading-if statement interleaved with jump targets and instructions that associate the original value with enabled target. The code before the statement looks up whether the association for a given value has already been established, and, if so, jumps directly to the target; otherwise the sequential execution of the cascading-if is started. To ensure that the actual code associated with the predicates remains unaware of this optimization, the code preceeding it after the target must re-establish any invariant guaranteed by sequential execution (§3.7).

The above code can easily be produced in a compiler setting, but producing it in a library setting is a challenge. Inspired by Duff's Device [43], we devised a construct that we call *Memoization Device* that does just that in standard C++:

```
typedef decltype(x) T;
static std::unordered_map<T,size_t> jump_targets;

switch (size_t& jump_to = jump_targets[x]) {
default: // entered when we have not seen x yet
    if (P1(x)) { jump_to = 1; case 1: s1;} else
    if (P2(x)) { jump_to = 2; case 2: s2;} else
      ...
    if (Pn(x)) { jump_to = n; case n: sn;} else
                 jump_to = n+1;
case n+1: // none of the predicates is true on x
}
```

The static jump_targets hash table will be allocated upon first entry to the function. The map is initially empty and according to its logic, request for a key x not yet in the map will allocate a new entry with its associated data default initialized (to 0 for size_t). Since there is no case label 0 in the switch, the default case will be taken, which, in turn, will initiate sequential execution of the interleaved cascading-if statement. Assignments to jump_to effectively establish association between value x and corresponding predicate, since jump_to is just a reference to jump_targets[x]. The last assignment records absence of enabled predicates for the value.

To change the first-fit semantics of the above construct into *sequential all-fit*, we remove the **else**s and rely on fall-through behavior of the switch. We also make the assignments conditional to make sure only the first one gets recorded:

if (P_i(x)) { **if** (jump_to ==0) jump_to = i; **case** i: s_i;}

Note that the protocol that has to be maintained by this structure does not depend on the actual values of case labels. We only require them to be different and include a predefined default value. The default clause can be replaced with a case clause for the predefined value, however keeping the default clause results in a faster code. A more important perfor-

mance consideration is to keep the values close to each other. Not following this rule might result in a compiler choosing a decision tree over a jump table implementation of the switch, which in our experience significantly degrades the performance.

The first-fit semantics is not an inherent property of the memoization device. Assuming that the conditions are either mutually exclusive or imply one another, we can build a decision-tree-based memoization device that will effectively have *most-specific* semantics – an analog of best-fit semantics in predicate dispatching [17].

Imagine that the predicates with the numbers $2i$ and $2i + 1$ are mutually exclusive and each imply the value of the predicate with number i i.e. $\forall x \in \mathsf{Domain}(P)$

$$P_{2i+1}(x) \rightarrow P_i(x) \wedge P_{2i}(x) \rightarrow P_i(x) \wedge \neg(P_{2i+1}(x) \wedge P_{2i}(x))$$

An example of predicates that satisfy this condition are class membership tests where the truth of testing membership in a derived class implies the truth of testing membership in its base class.

The following decision-tree based memoization device will execute the statement s_i associated with the *most-specific* predicate P_i (i.e. the predicate that implies all other predicates true on x) that evaluates to true or will skip the entire statement if none of the predicates is true on x.

```
switch (size_t& jump_to = jump_targets[x]) {
default:
    if (P1(x)) {
        if (P2(x)) {
            if (P4(x)) { jump_to = 4; case 4: s4;} else
            if (P5(x)) { jump_to = 5; case 5: s5;}
            jump_to = 2; case 2: s2;
        } else
        if (P3(x)) {
            if (P6(x)) { jump_to = 6; case 6: s6;} else
            if (P7(x)) { jump_to = 7; case 7: s7;}
            jump_to = 3; case 3: s3;
        }
        jump_to = 1; case 1: s1;
    } else { jump_to = 0; case 0: ; }
}
```

Our library solution prefers the simpler cascading-if approach only because the necessary structure of the code can be laid out directly with macros. A compiler solution will use the decision-tree approach whenever possible to lower the cost of the first match from linear in case's number to logarithmic as seen in Figure**??**.

The main advantage of the memoization device is that it can be built around almost any code, providing that we can re-establish the invariants, guaranteed by sequential execution. Its main disadvantage is the size of the hash table that grows proportionally to the number of different values seen. Fortunately, the values can often be grouped into equivalence classes that do not change the outcome of the predicate. The map can then associate the equivalence class of a value with a target instead of associating the value with it.

In application to type switching, the idea is to use the memoization device to learn the outcomes of type inclusion tests (with **dynamic_cast** used as a predicate), thus avoiding calls to it on subsequent runs. It is easy to see that objects can be grouped into equivalence classes based on their most-derived type without affecting the results of predicates – the outcome of each type inclusion test will be the same on all the objects from the same equivalence class. We can use the address of class' **type_info** object obtained in constant time with **typeid**() operator as a unique identifier of each most-derived type. Presence of multiple **type_info** objects for the same class, as is often the case when dynamic linking is involved, is not a problem as we would effectively split a single equivalence class into multiple ones. This in fact would have been a solution if we were only interested in class membership. More often than not, however, we will be interesting in obtaining a reference to the target type of the subject and we saw in §**??** that proper this-pointer adjustments depend not only on the most-derived type, but also on target type and most importantly – path to the subject's static type from the most-derived type in the inheritance graph. Ideally we would like to have different equivalence classes per different paths from object's most-derived type to its static types, but there seem to be no easy way of identifying them given just an object descriptor.

3.4 Virtual Table Pointers

A class that declares or inherits a virtual function is called a *polymorphic class*. The C++ standard [24] does not prescribe any specific implementation technique for virtual function dispatch. However, in practice, all C++ compilers use a strategy based on so-called virtual function tables (or vtables for short) for efficient disptach. The vtable is part of the reification of a polymorphic class type. C++ compilers embed a pointer to a vtable (vtbl-pointer for short) in every object of polymorphic class type. CFront, the first C++ compiler, puts the vtbl-pointer at the end of an object. The so-called "common vendor C++ ABI"[8], further referred to as *C++ ABI* when not indicated otherwise, requires the vtbl-pointer to be at offset 0 of an object. The following compilers comply with the C++ ABI: GCC (3.x and up); Clang and llvm-g++; Linux versions of Intel and HP compilers, and compilers from ARM. We do not have access to the unpublished Microsoft ABI, but we have experimental evidence that Microsoft's C++ compiler also puts the vtbl-pointer at the start of an object.

While the exact offset of the vtbl-pointer within the object is not important for our discussion, it is important to realize that every object of a static type S∗ or S& pointed to or referenced by a polymorphic class S will have a vtbl-pointer at a predefined offset. Such offset may be different for different static types S, in which case the compiler will

know at which offset in type S the vtbl-pointer is located. For a library implementation we assume presence of a function **template** ⟨**typename** S⟩intptr_t vtbl(**const** S∗ s); that returns the address of the virtual table corresponding to the subobject referenced to by s. Such a function can be trivially implemented for the common C++ ABI, where the vtbl-pointer is always at offset 0:

```
template ⟨typename S⟩ std::intptr_t vtbl(const S* s) {
    static_assert(std::is_polymorphic⟨S⟩::value, "error");
    return *reinterpret_cast⟨const std::intptr_t*⟩(s);
}
```

Consider a repeated multiple inheritance hierarchy from Figure **??**(1). Each of the vtbl fields shown in Figure **??**(2) will hold a vtbl-pointer referencing a group of virtual methods known in object's static type. Figure 3(1) shows a typical layout of virtual function tables together with objects it points to for classes B and D.

Figure 3. VTable layout with and without RTTI

Entries in the vtable to the right of the address pointed to by a vtbl-pointer represent pointers to functions, while entries to the left of it represent various additional fields like: pointer to class' type information, offset to top, offsets to virtual base classes etc. In many implementations, this-pointer adjustments required to properly dispatch the call were stored in the vtable along with function pointers. Today most of the implementations prefer to use *thunks* or *trampolines* – additional entry points to a function, that adjust this-pointer before transferring the control to the function, – which was shown to be more efficient []. Thunks in general may only be needed when its virtual function gets overriden. In such case the overriden function may be called via pointer to base class or a pointer to derived class, which may not be at the same offset in the actual object.

The intuition behind our proposal is to use the values of vtbl-pointers stored inside the object to uniquely identify the subobject in it. There are several problems with the approach however. First of all the same vtbl-pointer is usually shared by multiple types, for example, the first vtbl-pointer in Figure **??**(2) will be shared by objects of static type Z∗, A∗, B∗ and D∗. This is not a problem for our purpose, because the subobjects of these types will be at the same offset in the most-derived object. Secondly, and more importantly, however, there are legitimate optimizations that let the compiler share the same vtable among multiple subobjects of often unrelated types.

Generation of the *Run-Time Type Information* (or RTTI for short) can typically be disabled with a compiler switch and the Figure 3(2) shows the same vtable layouts once the RTTI has been disabled. Since neither baz nor foo were overriden, the prefix of the vtable for the C subobject in D is exactly the same as the vtable for its B subobject, the A subobject of C or the entire vtable of A and B classes. Such layout, for example, is produced by Microsoft Visual C++ 11 when the command-line option /GR− is specified. Visual C++ compiler has been known to unify code identical on binary level, which in some cases may result in sharing of the same vtable between unrelated classes (e.g. when virtual functions are empty).

We now would like to show more formally that in the presence of RTTI, a C++ ABI compliant implementation will always have all the vtbl-pointers different. To do so, we need look closer at the notion of subobject, which has been formalized before [38, 40, 47]. We follow here the presentation of Ramamanandro et al [38].

3.5 Subobjects

In a given program P, a class B is a *direct repeated base class* of D if B is mentioned in the list of base classes of D without the **virtual** keyword ($D \prec_R B$). Similarly, a class B is a *direct shared base class* of D if B is mentioned in the list of base classes of D with the **virtual** keyword ($D \prec_S B$). A reflexive transitive closure of these relationships $\preceq^* = (\prec_R \cup \prec_S)^*$ defines the *subtyping* relation on types of program P. A base class *subobject* of a given *complete object* is represented by a pair $\sigma = (h, l)$ with $h \in \{\mathsf{Repeated}, \mathsf{Shared}\}$ representing the kind of inheritance (single inheritance is Repeated with one base class) and l representing the path in a non-virtual inheritance graph. A predicate $C \prec \sigma \succ A$ states that σ designates a subobject of static type A within the most-derived object of type C. More formally:

$$C \prec (\mathsf{Repeated}, C :: \epsilon) \succ C$$

$$\frac{C \prec_R B \qquad B \prec (\mathsf{Repeated}, l) \succ A}{C \prec (\mathsf{Repeated}, C :: l) \succ A}$$

$$\frac{C \prec_S B \qquad B \prec (h, l) \succ A}{C \prec (\mathsf{Shared}, l) \succ A}$$

ϵ indicates an empty path, but we will generally omit it in writing when understood from the context. In case of repeated inheritance in Figure 1(1), an object of the most-derived class D will have the following Repeated subobjects: D::C::Y, D::B::A::Z, D::C::A::Z, D::B::A, D::C::A, D::B, D::C, D. Similarly, in case of virtual inheritance in the same expample, an object of the most-derived class D will have the following Repeated subobjects: D::C::Y, D::B, D::C, D as well as the following Shared subobjects: D::A::Z, D::Z, D::A.

It is easy to show by structural induction on the above definition, that $C \prec \sigma \succ A \implies \sigma = (h, C :: l_1) \wedge \sigma = (h, l_2 :: A :: \epsilon)$, which simply means that any path to a subobject of static type A within the most-derived object of type C starts with C and ends with A. This objservation shows that $\sigma_\perp = (\mathsf{Shared}, \epsilon)$ does not represent a valid subobject. If Σ_P is the domain of all subobjects in the program P extended with $\sigma_\perp$, then a *cast* operation can be understood as a function $\delta : \Sigma_P \to \Sigma_P$. We use $\sigma_\perp$ to indicate an impossibility of a cast. The fact that δ is defined on subobjects as opposed to actual run-time values reflects the non-coercive nature of the operation – i.e. the underlain value remains the same. Any implementation of such a function must thus satisfy the following condition:

$$\delta(\sigma_1) = \sigma_2 \wedge C \prec \sigma_1 \succ A \implies C \prec \sigma_2 \succ B$$

i.e. the most-derived type of the value does not change during casting, only the way we reference it does. We refer to A as the *source type* and σ_1 as the *source subobject* of the cast, while to B as the *target type* and to σ_2 as the *target subobject* of it. The type C is the most-derived type of the value being casted. The C++ semantics states more requirement to the implementation of δ: e.g. $\delta(\sigma_\perp) = \sigma_\perp$ etc. but their precise modeling is out of the scope of this discussion. We would only like to point out here that since the result of the cast does not depend on the actual value and only on the source subobject and the target type, we can memoize the outcome of a cast on one instance in order to apply its results to another.

3.6 Uniqueness of vtbl-pointers under the C++ ABI

Given a reference a to polymorphic type A that points to a subobject σ of the most-derived type C (i.e. $C \prec \sigma \succ A$ is true), we will use the traditional field-access notion a.vtbl to refer to the virtual table of that subobject. The exact structure of the virtual table as mandated by the common vendor C++ ABI is immaterial for this discussion, but we mention a few fields that are important for the reasoning [8, §2.5.2]:

- rtti(a.vtbl): the *typeinfo pointer* points to the typeinfo object used for RTTI. It is always present and is shown as the first field to the left of any vtbl-pointer in Figure 3(1).
- off2top(a.vtbl): the *offset to top* holds the displacement to the top of the object from the location within the object of the vtbl-pointer that addresses this virtual table. It is always present and is shown as the second field to the left of any vtbl-pointer in Figure 3(1). The numeric value shown indicates the actual offset based on the object layout from Figure **??**(2).
- vbase(a.vtbl): *Virtual Base (vbase) offsets* are used to access the virtual bases of an object. Such an entry is required for each virtual base class. None are shown in our example in Figure 3(1) since it discussed repeated inheritance, but they will occupy further entries to the left of the vtbl-pointer, when present.

We also use the notation $\mathit{offset}(\sigma)$ to refer to the offset of the given subobject σ within C, known by the compiler.

Theorem 1. *In an object layout that adheres to the common vendor C++ ABI with enabled RTTI, equality of vtbl-pointers of two objects of the same static type implies that they both belong to subobjects with the same inheritance path in the same most-derived class.*
$\forall a_1, a_2 : A \mid a_1 \in C_1 \prec \sigma_1 \succ A \wedge a_2 \in C_2 \prec \sigma_2 \succ A$
$a_1.\mathit{vtbl} = a_2.\mathit{vtbl} \Rightarrow C_1 = C_2 \wedge \sigma_1 = \sigma_2$

Proof. Let us assume first $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$ but $C_1 \neq C_2$. In this case we have $\mathsf{rtti}(a_1.\mathit{vtbl}) = \mathsf{rtti}(a_2.\mathit{vtbl})$. By definition $\mathsf{rtti}(a_1.\mathit{vtbl}) = C_1$ while $\mathsf{rtti}(a_2.\mathit{vtbl}) = C_2$, which contradicts that $C_1 \neq C_2$. Thus $C_1 = C_2 = C$.

Let us assume now that $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$ but $\sigma_1 \neq \sigma_2$. Let $\sigma_1 = (h_1, l_1), \sigma_2 = (h_2, l_2)$

If $h_1 \neq h_2$ then one of them refers to a virtual base while the other to a repeated one. Assuming h_1 refers to a virtual base, $\mathsf{vbase}(a_1.\mathit{vtbl})$ has to be defined inside the vtable according to the ABI, while $\mathsf{vbase}(a_2.\mathit{vtbl})$ – should not. This would contradict again that both vtbl refer to the same virtual table.

We thus have $h_1 = h_2 = h$. If $h = \mathsf{Shared}$ then there is only one path to such A in C, which would contradict $\sigma_1 \neq \sigma_2$. If $h = \mathsf{Repeated}$ then we must have that $l_1 \neq l_2$. In this case let k be the first position in which they differ: $l_1^j = l_2^j \forall j < k \wedge l_1^k \neq l_2^k$. Since our class A is a base class for classes l_1^k and l_2^k, both of which are in turn base classes of C, the object identity requirement of C++ requires that the relevant subobjects of type A have different offsets within class C: $\mathit{offset}(\sigma_1) \neq \mathit{offset}(\sigma_2)$ However $\mathit{offset}(\sigma_1) = \mathsf{off2top}(a_1.\mathit{vtbl}) = \mathsf{off2top}(a_2.\mathit{vtbl}) = \mathit{offset}(\sigma_2)$ since $a_1.\mathit{vtbl} = a_2.\mathit{vtbl}$, which contradicts that the offsets are different. □

Conjecture in the other direction is not true in general as there may be duplicate vtables for the same type present at run-time. This happens in many C++ implementations in the presence of *Dynamically Linked Libraries* (or DLLs for short) as the same class compiled into executable and DLL it loads may have identical vtables inside the executable's and DLL's binaries.

Note also that we require both static types to be the same. Dropping this requirement and saying that equality of vtbl-

pointers also implies equality of the static types is not true in general because a derived class can share the vtbl-pointer with its primary base class. The theorem can be reformulated, however, stating that one static type will necessarily be a subtype of the other. The current formulation is sufficient for our purposes, while reformulation will require more elaborate discussion of the algebra of subobjects [38], which we touch only briefly.

During construction and deconstruction of an object, the value of a given vtbl-pointer may change. In particular, that value will reflect the fact that the most-derived type of the object is the type of its fully constructed part only. This, however, does not affect our reasoning, as during such transition we also treat the object to have the type of its fully constructed base only. Such interpretation is in line with the C++ semantics for virtual function calls and the use of RTTI during construction and destruction of an object. Once the complete object is fully constructed, the value of the vtbl-pointer will remain the same for the lifetime of the object.

3.7 Vtable Pointer Memoization

The C++ standard requires that information about types be available at run time for three distinct purposes:

- to support the **typeid** operator,
- to match an exception handler with a thrown object, and
- to implement the **dynamic_cast** operator.

and if any of these facilities are used in a program that was compiled with disabled RTTI, the compiler will emit an error or at least a warning. Some compilers (e.g. Visual C++) additionally let a library check presence of RTTI through a predefined macro, thus letting it report an error if its dependence on RTTI cannot be satisfied. Since our solution relies on **dynamic_cast** to perform casts at run-time, we implicitly rely on the presence of RTTI and thus fall into the setting that guarantees the preconditions of Theorem 1. Since all the objects that will be coming through a particular type switch will have the same static type, the theorem guarantees that different vtbl-pointers will correspond to different subobjects. The idea is thus to group them accordingly to the value of their vtbl-pointer and associate both jump target and the required offset with it through memoization device:

```
typedef pair⟨ptrdiff_t,size_t⟩ type_switch_info;
static unordered_map⟨intptr_t, type_switch_info⟩ jump_targets;
type_switch_info& info = jump_targets[vtbl(x)];
const void* tptr;
switch (info.second) ...
```

The code for the ith case now evaluates the required offset on the first entry and associates it and the target with the vtbl-pointer of the subject. The call to adjust_ptr$\langle T_i \rangle$ re-establishes the invariant that match is a reference to type T_i of the subject x.

```
    if (tptr = dynamic_cast⟨const T_i*⟩(x)) {
        if (info.second ==0) { // supports fall−through
            info.first = intptr_t(tptr)−intptr_t(x); // offset
            info.second = i; // jump target
        }
case i: // i is a constant here − clause's position in switch
        auto match = adjust_ptr⟨T_i⟩(x,info.first);
        s_i;
    }
```

Class std::unordered_map provides amortized constant time access on average and linear in the amount of elements in the worst case. We show in the next section that most of the time we will be bypassing traditional access to its elements. We need this extra optimization because, as-is, the type switch is still about 50% slower than the visitor design pattern.

3.8 Minimization of Conflicts

Virtual table pointers are not constant values and are not even guaranteed to be the same between different runs of the application, because techniques like *address space layout randomization* or *rebasing* of the module are likely to change them. The relative distance between them will remain the same as long as they come from the same module.

Knowing that vtbl-pointers point into an array of function pointers, we should expect them to be aligned accordingly and thus have a few lowest bits as zero. Moreover, since many derived classes do not introduce new virtual functions, the size of their virtual tables remains the same. When allocated sequentially in memory, we can expect a certain number of lowest bits in the vtbl-pointers pointing to them to be the same. These assumptions, supported by actual observations, has made virtual table pointers of classes related by inheritance ideally suitable for hashing – the values obtained by throwing away the common bits on the right were compactly distributed in small disjoint ranges. We use them to address a cache built on top of the hash table in order to eliminate a hash table lookup in most of the cases.

Let Ξ be the domain of integral representations of pointers. Given a cache with 2^k entries, we use a family of hash functions $H_{kl} : \Xi \to [0..2^k - 1]$ defined as $H_{kl}(v) = v/2^l \mod 2^k$ to index the cache, where $l \in [0..32]$ (assuming 32 bit addresses) is a parameter modeling the number of common bits on the right. Division and modulo operations are implemented with bit-shifts since denominator in each case is a power of 2, which in turn explains the choice of the cache size.

Given a hash function H_{kl}, pointers v' and v'' are said to be in *conflict* when $H_{kl}(v') = H_{kl}(v'')$. For a given set of pointers $V \in 2^{\Xi}$ we can always find such k and l that H_{kl} will render no conclicts between its elements, however the required cache size 2^k can be too large to justify the use of memory. The value K such that $2^{K-1} < |V| \leq 2^K$ is the smallest value of k under which absence of conflicts is still possible. We thus allow k to only vary in range $[K, K+1]$ to

ensure that the cache size is never more than 4 times bigger than the minimum required cache size.

Given a set $V = \{v_1, ..., v_n\}$, we would like to find a pair of parameters (k, l) such that H_{kl} will render the least number of conflicts on the elements of V. Since for a fixed set V, parameters k and l vary in a finite range, we can always find the optimal (k, l) by trying all the combinations. Let $H^V_{kl} : V \to [0..2^k-1]$ be the hash function corresponding to such optimal (k, l) for the set V.

In our setting, set V represents the set of vtbl-pointers coming through a particular type switch. While the exact values of these pointers are not known till run-time, their offset from the module's base address is. This can often be sufficient to at least estimate optimal k and l in a compiler setting. In the library setting we estimate them by recomputing them after a given amount of actual collisions happened in cache.

When H^V_{kl} is injective (renders 0 conflicts on V), the frequency of any given vtbl-pointer v_i coming through the type switch does not affect the overal performance of the switch. However when H^V_{kl} is not injective, we would prefer the conflict to happen on less frequent vtbl-pointers. Given a probability p_i of each vtbl-pointer $v_i \in V$ we can compute the probability of conflict rendered by a given H_{kl}:

$$P_{kl}(V) = \sum_{j=0}^{2^k-1} \Big(\sum_{v_i \in V^j_{kl}} p_i \Big) \Big(1 - \frac{\sum\limits_{v_i \in V^j_{kl}} p_i^2}{\big(\sum\limits_{v_i \in V^j_{kl}} p_i \big)^2} \Big)$$

where $V^j_{kl} = \{v \in V | H_{kl}(v) = j\}$. In this case, the optimal hash function H^V_{kl} can similarly be defined as H_{kl} that minimizes the above probability of conflict on V.

Probabilities p_i can be estimated in a compiler settings through profiling, while in a library setting we let the user enable tracing of frequencies of each vtbl-pointer. This introduces an overhead of an increment into the critical path of execution, and according to our tests degrades the overal performance by 1-2%. By default, we do not enable frequency tracing, however, because the significant drop in the number of actual collisions was not reflected in a noticeable decrease in execution time. This was because the total number of actual collisions, even in non-frequency based caching, was much smaller than the number of successful cache hits.

Assuming the uniform distribution of v_i and substituting the probability $p_i = \frac{1}{n}$, where $n = |V|$, into the above formula we will get:

$$P_{kl}(V) = \sum_{j=0}^{2^k-1} [|V^j_{kl}| \neq 0] \frac{|V^j_{kl}| - 1}{n}$$

The value $|V^j_{kl}| - 1$ represents the amount of "extra" pointers mapped into the entry j in cache and thus H^V_{kl} obtained by minimization of probability of conflict is the same as our original H^V_{kl} minimizing the number of conflicts. An important observation here is that the exact location of these "extra" vtbl-pointers is not important, only the total number m of them is. The probability of conflict under uniform distribution of v_i is thus always going to have form $\frac{m}{n}$, where $0 \leq m < n$.

4. Evaluation

Our evaluation methodology consisted of several independent studies of the technique in order to achieve a better confidence in its validity. Our first study involved comparison of relative performance of our approach against the visitor design pattern. The second study did a similar comparison with built-in facilities of Haskell and OCaml. In the third study we looked at how well our caching mechanisms deal with some large real-world class hierarchies. In the last study we did rewrite an existing application that was based on visitors using our approach and compared the two.

4.1 Comparison with Visitor Design Pattern

Our evaluation methodology consists of several benchmarks representing various uses of objects inspected with either visitors or type switching.

The *repetitive* benchmark (REP) performs calls on different objects of the same most-derived type. This scenario happens in object-oriented setting when a group of polymorphic objects is created and passed around (e.g. numerous particles of a given kind in a particle simulation system). We include it because double dispatch becomes twice faster (27 vs. 53 cycles) in this scenario compared to others due to cache and call target prediction mechanisms.

The *sequential* benchmark (SEQ) effectively uses an object of each derived type only once and then moves on to an object of a different type. The cache is typically reused the least in this scenario, which is typical of lookup tables, where each entry is implemented with a different derived class.

The *random* benchmark (RND) is the most representative as it randomly makes calls on random objects, which will probably be the most common usage scenario in the real world.

Presence of *forwarding* in any of these benchmarks refers to the common technique used by visitors where, for class hierarchies with multiple levels of inheritance, the visit method of a derived class will provide a default implementation of forwarding to its immediate base class, which, in turn, may forward it to its base class, etc. The use of forwarding in visitors is a way to achieve substitutability, which in type switch corresponds to the use of base classes in the case clauses.

The class hierarchy for non-forwarding test was a flat hierarchy with 100 derived classes, encoding an algebraic data type. The class hierarchy for forwarding tests had two levels of inheritance with 5 intermediate base classes and 95 derived ones. While we do not advocate here for the

closed solution of §3.1, we included it in our tests to show the performance gains a closed solution might have over the open one. Our library supports both solutions with the same surface syntax, which is why we believe many users will try them both before settling on one.

The benchmarks were executed in the following configurations refered to as *Linux Desktop* and *Windows Laptop* respectively:

- *Lnx*: Dell Dimension® desktop with Intel® Pentium® D (Dual Core) CPU at 2.80 GHz; 1GB of RAM; Fedora Core 13
  - G++ 4.4.5 executed with -O2
- *Win*: Sony VAIO® laptop with Intel® Core™i5 460M CPU at 2.53 GHz; 6GB of RAM; Windows 7 Professional
  - G++ 4.5.2 and 4.6.1 / MinGW executed with -O2; x86 binaries
  - MS Visual C++ 2010 Professional x86/x64 binaries with profile-guided optimizations

The code on the critical path of our type switch implementation benefits significantly from branch hinting as some branches are much more likely than others. We use the branch hinting in GCC to guide the compiler, but, unfortunately, Visual C++ does not have similar facilities. Microsoft suggests to use *Profile-Guided Optimization* to achieve the same, which is why the results for Visual C++ reported here have been obtained with profile-guided optimizations enabled. The results without profile-guided optimizations can be found in the accompanying technical report [4, §10].

We compare the performance of our solution relative to the performance of visitors in Figure 4. The values are given as percentages of performance increase against the slower technique. Numbers in regular font represent cases where type switching was faster, while numbers in bold indicate cases where visitors were faster.

		Open				Closed			
		G++		MS VC++		G++		MS VC++	
		x86-32	x86-32	x86-32	x86-64	x86-32	x86-32	x86-32	x86-64
	REP	16%	55%	4%	**0%**	124%	216%	124%	47%
	SEQ	56%	3%	3%	1%	640%	520%	34%	14%
	RND	56%	1%	**18%**	**27%**	603%	542%	43%	16%
Forward	REP	33%	67%	10%	**6%**	53%	79%	31%	**9%**
Forward	SEQ	55%	90%	153%	145%	86%	259%	185%	118%
Forward	RND	78%	27%	18%	**6%**	88%	31%	24%	**10%**
		Lnx	Win			Lnx	Win		

Figure 4. Relative performance of type switching versus visitors. Numbers in regular font (e.g. 67%), indicate that our type switching is faster than visitors by corresponding percentage. Numbers in bold font (e.g. **18%**), indicate that visitors are faster by corresponding percentage.

We can see that type switching wins by a good margin when implemented with tag switch as well as in the presence of at least one level of forwarding. Note that the numbers are relative, and thus the ratio depends on both the performance of virtual function calls and the performance of switch statements. Visual C++ was generating faster virtual function calls, while GCC was generating faster switch statements, which is why their relative performance seem to be much more favorable for us in the case of GCC. Similarly the code for x64 is only slower relatively: the actual time spent for both visitors and type switching was smaller than that for x86, but it was much smaller for visitors than type switching, which resulted in worse relative performance.

4.2 Open vs. Closed Type Switch

With a few exceptions for x64, it can be seen from Figure 4 that the performance of the closed tag switch dominates the performance of the open type switch. We believe that the difference, often significant, is the price one pays for the true openness of the vtable pointer memoization solution.

As we mentioned in §3.1, the use of tags, even allocated by compiler, may require integration efforts to ensure that different DLLs have not reused the same tags. Randomization of tags, similar to a proposal of Garrigue [19], will not eliminate the problem and will surely replace jump tables in switches with decision trees. This will likely significantly degrade the numbers for the part of Figure 4 representing closed tag switch, since the tags in our experiments were all sequential.

The reliance of a tag switch on static cast has severe limitations in the presence of multiple inheritance, and thus is not as versatile as open type switch. Overcoming this problem will either require the use of **dynamic_cast** or techniques similar to those used for vtable pointer memoization, which will likely degrade tag switch'es performance numbers even further.

Note also that the approach used to implement open type switch can be used to implement both first-fit and best-fit semantics, while the tag switch is only suitable for best-fit semantics. Their complexity guarantees also differ: open type switch is constant on average, but slow on the first call with given subobject. Tag switch is logarithmic in the size of the class hierarchy (assuming a balanced hierarchy), including the first call. This last point can very well be seen in Figure 4, where the performance of a closed degrades significantly in the presence of forwarding, while the performance of open solution improves.

4.3 Comparison with OCaml and Haskell

We now compare our solution to the built-in pattern-matching facility of OCaml [28] and Haskell [25]. In this test, we timed small OCaml and Haskell applications performing our sequential benchmark on an algebraic data type of 100 variants. Corresponding C++ applications were working with a flat class hierarchy of 100 derived classes. The difference be-

tween the C++ applications lies in the encoding used. Kind encoding is the same as Tag encoding, but it does not require substitutability, and thus can be implemented with a direct switch on tags. It is only supported through specialized syntax in our library as it differs from the Tag encoding only semantically.

We used optimizing OCaml compiler `ocamlopt.opt` version 3.11.0 working under the Visual C++ toolset as well as the Glasgow Haskell Compiler version 7.0.3 (with -O switch) working under the MinGW toolset. All the tests were performed on the Windows 7 laptop. The timing results presented in Figure 5 are averaged over 101 measurements and show the number of seconds it took to perform a million decompositions within our sequential benchmark.

Figure 5. Performance comparison of various encodings and syntax against OCaml code

4.4 Dealing with real-world class hierarchies

We used a class hierarchy benchmark used before to study efficiency of type inclusion testing and dispatching techniques [14, 26, 44, 51]. We use the names of the benchmarks from Vitek et al [44, Table 2], since the set of benchmarks we were working with was closest to that work.

While not all class hierarchies originated from C++, for this experiment it was more important for us that the hierarchies were man-made. While converting the hierachies into C++, we had to prune inaccessible base classes (direct base class that is already an indirect base class) when used with repeated inheritance in order to satisfy semantic requirements of the C++. We maintained the same number of virtual functions present in each class as well as the number of data members. The benchmarks, however, did not preserve the types of those. The data in Figure 6 shows various parameters of the class hierarchies in each benchmark, after their adoption to C++.

The number of paths represents the number of distinct inheritance paths from the classes in the hierarchy to the roots of the hiearchy. As we showed in §**??** this number reflects the number of possible subobjects in the hierarchy. The roots listed in the table are classes with no base classes. We will subsequently use the term *non-leaf* to refer to the possible root of a subhierarchy. Leafs are classes with no children, while *both* refers to utility classes that are both roots and leafs and thus neiter have base nor derived classes. The average for the number of parents and the number of

Library	Language	Classes	Paths	Height	Roots	Leafs	Both	Parents		Children	
								avg	max	avg	max
DG2	Smalltalk	534	534	11	2	381	1	1	1	3.48	59
DG3	Smalltalk	1356	1356	13	2	923	1	1	1	3.13	142
ET+	C++	370	370	8	87	289	79	1	1	3.49	51
GEO	Eiffel	1318	13798	14	1	732	0	1.89	16	4.75	323
JAV	Java	604	792	10	1	445	0	1.08	3	4.64	210
LOV	Eiffel	436	1846	10	1	218	0	1.72	10	3.55	78
NXT	Objective-C	310	310	7	2	246	1	1	1	4.81	142
SLF	Self	1801	36420	17	51	1134	0	1.05	9	2.76	232
UNI	C++	613	633	9	147	481	117	1.02	2	3.61	39
VA2	??	3241	3241	14	1	2582	0	1	1	4.92	249
VA2	??	2320	2320	13	1	1868	0	1	1	5.13	240
VW1	Smalltalk	387	387	9	1	246	0	1	1	2.74	87
VW2	Smalltalk	1956	1956	15	1	1332	0	1	1	3.13	181
	Overalls	15246	63963	17	298	10877	199	1.11	16	3.89	323

Figure 6. Benchmarks Class Hierarchies

children were computed only among the classes having at least one parent or at least one children correspondingly.

With few useful exceptions, it generally makes sence to only apply type switch to non-leaf nodes of the class hierarchy. 71% of the classes in the entire benchmarks suite were leaf classes. Out of the 4369 non-leaf classes 36% were spawning a subhierarchy of only 2 classes (including the root), 15% – a subhierarchy of 3 classes, 10% of 4, 7% of 5 and so forth. Turning this into a cumulative distribution, $a\%$ of subhierarchies had more than b classes in them, where:

a	1%	3%	5%	10%	20%	25%	50%	64%	100%
b	700	110	50	20	10	7	3	2	1

These numbers reflect the percentage of use cases one may expect in the real word that have a given number of case clauses in them.

For each non-leaf class A we created a function performing a type switch on every possible derived class D_i of it as well as itself. The function was then executed with every possible subobject $D_i \prec \sigma_j \succ A$ it can possibly be applied to, given the static type A of the subject. It was executed multiple but the same number of times on each subobject to ensure uniformity on one side (since we do not have the data about the actual probabilities of each subobject the benchmark hierarchies) as well as let the type switch infer the optimal parameters k and l of its cache indexing function H_{kl}. We then plotted the result of each of the 4396 experiments as a point in chart of Figure 7 relating the optimal probability of conflict p achieved by the type switch and the number of subobjects n that came through that type switch. To account for the fact that multiple experiments could have resulted in the same pair (n, p), we use a shadow of each point to reflect somewhat the number of experiments yielding it.

The curves on which the results of experiments line up correspond to the fact that under uniform distribution of n subobjects, only a finite number of different values representing the probability of conflict p are possible. In particular, all such values $p = \frac{m}{n}$, where $0 \leq m < n$. The number m reflects the number of subobjects an optimal cache indexing function H_{kl} could not allocate their own entry for. We

Figure 7. Probability of Conflict vs. Number of Subobjects in Hierarchy

showed in §3.8 that the probability of conflict under uniform distribution of n subobjects depends only on m. The points on the same curve (which becomes a line on a log-log plot) all share the same number m of "extra" vtbl-pointers that optimal cache indexing function could not allocate individual entries for.

While it is hard to see from the chart, 87.5% of all the points on the chart lay on the X-axis, which means that the optimal hash function for the corresponding type switches had no conflicts at all. In other words, only in 12.5% of cases the optimal H_{kl}^V for the set of vtbl-pointers V coming through a given type switch had non-zero probability of conflict. This is why the average probability of conflict for the entire set is only 1.17. Experiments laying on the first curve amount to 5.58% of subhierarchies and represent the cases in which optimal H_{kl}^V had only one "extra" vtbl-pointer. 2.63% experiments had H_{kl}^V with 2 conflicts, 0.87% with 3 etc. These numbers do not indicate that the hash function we used is better than other hash functions, they do indicate instead that the set of vtbl-pointers present in many applications is particularly suitable for such a hash function.

4.5 Refactoring an existing visitors based application

For this experiment we have reimplemented a visitor based C++ pretty printer for Pivot[39] using our pattern-matching library. Pivot's class hierarchy consists of 154 node kinds representing various entities in the C++ program. The original code had 8 visitor classes each handling 5, 7, 8, 10, 15, 17, 30 and 63 cases, which we turned into 8 match statements with corresponding numbers of case clauses. Most of the rewrite was performed by sed-like replaces that converted visit methods into respective case-clauses. In several cases we had to manually reorder case-clauses to avoid redundancy as visit-methods for base classes were typically coming before the same for derived, while for type switching we needed them to come after. Redundancy checking support provided by our library was invaluable in finding out all such cases.

Both pretty printers were executed on a set of header files from the C++ standard library and the produced output of both program was byte-to-byte the same. We timed execution of the pretty printing phase (not including loading and termination of the application or parsing of the input file) and observed that on small files (e.g. those from C run-time library and few small C++ files) visitors-based implementation was faster because the total number of nodes in AST and thus calls did not justify our set-up calls. In particular, visitor-based implementation of pretty printer was faster on files of 44–588 lines of code, with average 136 lines per those inputs, where visitors win. On these input files it is faster by 1.17%–21.42% with an average speed-up of 8.75%. Open type switch based implementation of pretty printer was faster on files of 144–9851 lines of code, with average 3497 lines per those input files, where open type switch wins. On these inputs it is faster by 0.18% – 32.99% with an average speed-up of 5.53%.

Figure 8 shows memory usage as well as cache hits and misses for the run of our pretty printer on 'queue' standard library header (it has the largest LOC after preprocessing in our test set).

Figure 8. Memory usage in real application

The bars represent the total amount of memory in bytes each of the 8 match statements (marked A-H) used. The info $[N/M]$ next to the letter indicates the actual number of different subobjects (i.e. vtbl-pointers) N that came through that match statement, and the number of case clauses M the match statement had (the library uses it as an estimate of N). N is also the number of cases the corresponding match statement had to be executed sequentially (instead of a direct jump).

The blue part of each bar corresponds to the memory used by cache, while the maroon – to the memory used by the hash table. Transparent parts of both colors indicate the allocated extra memory that is not holding any data. The black box within the blue part also indicates the amount

of entries in the cache that are allocated for only one vtbl pointer and thus never result in a cache miss. The non-transparent part without black box represents the percentage of vtbl-pointers that have to share their cache entry with at least one other vtbl-pointer and thus may result in collisions during access.

The actual number of hits and misses for each of the match statements is indicated on top of the corresponding column. The sum of them is the total amount of calls made. Hits indicate situation when we found entry in cache and didn't have to make roundtrip to the hash-table to get it. Misses indicate the number of cases during actual run we had to pick the entry from the hash table and update the cache with it. The number of misses is always larger then or equal to N.

The library always preallocates memory for at least 8 sub-objects to avoid unnecessery recomputations of optimal parameters k and l – this is the case with the last 3 match statements. In all other cases it allocate the amount of memory proportional to the smallest power of 2 that is greater of equal than $\max(M, N)$. The table does not have to be hash table and can be implemented with any other container i.e. sorted vector, map etc. that let us find quickly by a given vtbl-pointer the data associated with it. If we implement it with sorted vector, the red part will shrink to only the non-transparent part.

5. Related Work

Extensible Visitors with Default Cases [49, §4.2] attempt to solve the extensibility problem of visitors; however, the solution has problems of its own. The visitation interface hierarchy can easily be grown linearly, but independent extensions by different authorities require developer's intervention. On top of the double dispatch the solution will incur two additional virtual calls and a dynamic cast for each level of visitor extension. The solution is simpler with virtual inheritance, which adds even more indirections.

Löh and Hinze proposed to extend Haskell's type system with open data types and open functions [31]. The solution allows top-level data types and functions to be marked as open with concrete variants and overloads defined anywhere in the program. The semantics of open extension is given by transformation into a single module, which assumes a whole-program view and thus is not an open solution unfortunately. Besides, open data types are extensible but not hierarchical, which avoids the problems discussed here.

Polymorphic variants in OCaml [19] allow the addition of new variants as well as define subtyping on them. The subtyping, however, is not defined between the variants, but between combinations of them. This maintains disjointness between values from different variants and makes an important distinction between *extensible sum types* like polymorphic variants and *extensible hierarchical sum types* like classes. Our memoization device can be used to implement pattern matching on polymorphic variants.

Tom is a pattern-matching compiler that can be used together with Java, C or Eiffel to bring a common pattern matching and term rewriting syntax into the languages [34]. In comparison to our approach, Tom has much bigger goals: the combination of pattern matching, term rewriting and strategies turns Tom into a fully fledged tree-transformation language. Its type patterns and %match statement can be used as a type switch; however, Tom's handling of type switching is based on decision trees and an instanceof-like predicate, which are inefficient.

Pattern matching in Scala [35] also supports type switching through type patterns. The language supports extensible and hierarchical data types, but their handling in a type switching constructs varies. Sealed classes are handled with an efficient switch over all tags, while extensible classes are similarly approached with a combination of an InstanceOf operator and a decision tree [16].

6. Conclusions and Future Work

Type switching is an open alternative to visitor design pattern that overcomes the restrictions, inconveniences, and difficulties in teaching and using, typically associated with it. Our implementation of it comes close or outperforms the visitor design pattern, which is true even in a library setting using a production-quality compiler, where the performance base-line is already very high.

In the future we would like to provide a compiler implementation of our technique, which will enable a better surface syntax, improved diagnostics, increased performance and many other benefits hard to achieve in a library implementation.

Acknowledgments

We would like to thank Xavier Leroy and Luc Maranget for valuable feedback and suggestions for improvements on the initial idea, Gregory Berkolaiko for ideas related to minimization of conflicts, Jaakko Jarvi for assistance in comparison to other languages, Andrew Sutton, Peter Pirkelbauer and Abe Skolnik for helpful discussions and comments to numerous rewrites of this paper. We also benefitted greatly from insightful comments by anonymous reviewers on earlier revisions of this work. We would also like to thank Karel Driesen for letting us use his class hierarchies benchmark for this work.

References

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`, 2007.

[2] A. Appel, L. Cardelli, K. Fisher, C. Gunter, R. Harper, X. Leroy, M. Lillibridge, D. B. MacQueen, J. Mitchell, G. Morrisett, J. H. Reppy, J. G. Riecke, Z. Shao, and C. A.

Stone. Principles and a preliminary design for ML2000. March 1999.

[3] L. Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. URL `http://dl.acm.org/citation.cfm?id=5280.5303`.

[4] Blind Review. Technical report submitted as supplementary material. Technical Report XXXX, University, Nov. 2011.

[5] L. Cardelli. Compiling a functional language. In *Proc. of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802037. URL `http://doi.acm.org/10.1145/800055.802037`.

[6] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The modula3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 202–212, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75295. URL `http://doi.acm.org/10.1145/75277.75295`.

[7] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 271–287, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: 10.1145/165854.165905. URL `http://doi.acm.org/10.1145/165854.165905`.

[8] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI, March 2001. http://www.codesourcery.com/public/cxx-abi/.

[9] N. H. Cohen. Type-extension type test can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115297. URL `http://doi.acm.org/10.1145/115372.115297`.

[10] W. R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, 1991. Springer-Verlag. ISBN 3-540-53931-X.

[11] O.-J. Dahl. *SIMULA 67 common base language, (Norwegian Computing Center. Publication)*. 1968. ISBN B0007JZ9J6.

[12] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 83–100, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: 10.1145/236337.236344. URL `http://doi.acm.org/10.1145/236337.236344`.

[13] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. Jan. 1975.

[14] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):33:1–33:56, Oct. 2008. ISSN 0164-0925. doi: 10.1145/1391956.1391960. URL `http://doi.acm.org/10.1145/1391956.1391960`.

[15] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51459-1.

[16] B. Emir. *Object-oriented pattern matching*. PhD thesis, Lausanne, 2007.

[17] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.

[18] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5.

[19] J. Garrigue. Programming with polymorphic variants. In *ACM Workshop on ML*, Sept. 1998.

[20] M. Gibbs and B. Stroustrup. Fast dynamic casting. *Softw. Pract. Exper.*, 36:139–156, February 2006.

[21] N. Glew. Type dispatch for named hierarchical types. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 172–182, New York, NY, USA, 1999.

[22] C. Grothoff. Walkabout revisited: The runabout. In *ECOOP 2003 - Object-Oriented Programming*, pages 103–125. Springer-Verlag, 2003. URL `http://grothoff.org/christian/runabout/runabout.pdf`.

[23] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 130–141, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199475. URL `http://doi.acm.org/10.1145/199448.199475`.

[24] International Organization for Standardization. *ISO/IEC 14882:2011: Programming languages: C++*. Geneva, Switzerland, 3rd edition, 2011. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue`

[25] S. P. Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[26] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In *In Proc. European Conference on Object-Oriented Programming, ECOOP'97, Lecture Notes in Computer Science*, pages 128–145. Springer-Verlag, 1997.

[27] S. Krishnamurthi, M. Felleisen, and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Berlin / Heidelberg, 1998.

[28] F. Le Fessant and L. Maranget. Optimizing pattern matching. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0.

[29] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-266-7.

[30] B. Liskov, R. R. Atkinson, T. Bloom, E. B. Moss, R. Schaffert, and A. Snyder. Clu reference manual. Technical report, Cambridge, MA, USA, 1979.

[31] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM.

[32] Microsoft Research. Phoenix compiler and shared source common language infrastructure. `http://research.microsoft.com/phoenix/`, 2005.

[33] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262132559.

[34] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3.

[35] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. Mcdirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the scala programming language (second edition. Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, 2006.

[36] B. C. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 439–456, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3.

[37] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8585-9. URL `http://dl.acm.org/citation.cfm?id=645980.674267`.

[38] T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for c++ multiple inheritance. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 67–80, New York, NY, USA, 2011. ACM.

[39] G. D. Reis and B. Stroustrup. A principled, complete, and efficient representation of C++. In *Proc. Joint Conference of ASCM 2009 and MACIS 2009*, volume 22 of *COE Lecture Notes*, pages 407–421, December 2009.

[40] J. G. Rossie, Jr. and D. P. Friedman. An algebraic semantics of subobjects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 187–199, New York, NY, USA, 1995. ACM.

[41] L. Schubert, M. Papalaskaris, and J. Taugher. Determining type, part, color, and time relationships. *Computer*, 16:53–60, 1983. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.1983.1654198.

[42] D. A. Spuler. Compiler Code Generation for Multiway Branch Statements as a Static Search Problem. Technical Report Technical Report 94/03, James Cook University, Jan. 1994.

[43] Tom Duff. Duff's Device, Aug 1988. http://www.lysator.liu.se/c/duffs-device.html.

[44] J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 142–157, New York, NY, USA, 1997. ACM. ISBN 0-89791-908-4. doi: 10.1145/263698.263730. URL `http://doi.acm.org/10.1145/263698.263730`.

[45] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 13–24, New York, NY, USA, 2005. ACM. ISBN 1-58113-999-3. doi: 10.1145/1040294.1040296. URL `http://doi.acm.org/10.1145/1040294.1040296`.

[46] P. Wadler. The expression problem. Mail to the java-genericity mailing list, November 1998.

[47] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An operational semantics and type safety prooffor multiple inheritance in c++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 345–362, New York, NY, USA, 2006. ACM.

[48] N. Wirth. Type extensions. *ACM Trans. Program. Lang. Syst.*, 10(2):204–214, Apr. 1988. ISSN 0164-0925. doi: 10.1145/42190.46167. URL `http://doi.acm.org/10.1145/42190.46167`.

[49] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. ACM.

[50] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005.

[51] Y. Zibin and J. Y. Gil. Efficient subtyping tests with pq-encoding. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 96–107, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504290. URL `http://doi.acm.org/10.1145/504282.504290`.

Software Development for Infrastructure

Bjarne Stroustrup, *Texas A&M University*

Infrastructure software needs more stringent correctness, reliability, efficiency, and maintainability requirements than nonessential applications. This implies greater emphasis on up-front design, static structure enforced by a type system, compact data structures, simplified code structure, and improved tool support. Education for infrastructure and application developers should differ to reflect that emphasis.

Our lives are directly affected by software correctness and efficiency:

- A datacenter, as run by a major corporation such as Amazon, AT&T, Google, or IBM, uses about 15 MW per day (equivalent to 15,000 US homes) and the set-up cost is about US$500 million. In 2010, Google used 2.26 million MW to run its servers.[1]
- A smartphone battery lasts for less than a day if used a lot.
- We're surrounded by systems that, if they fail, can injure people or ruin them economically. Examples include automobile control systems, banking software, telecommunication software, and just about any industrial control software.
- We can't send a repairman to fix a software bug in a space probe, and sending one to reprogram a ship's engine on the high seas is impractical. Trying to fix a software error in a consumer device, such as a camera or a TV set, typically isn't economical.

The implication puts a new emphasis on the old challenge to software developers: deliver code that is both correct and efficient. The benefits achievable with better system design and implementation are huge. If we can double the efficiency of server software, we can make do with one server farm instead of two (or more). If we can double the efficiency of critical software in a smartphone, its battery life almost doubles. If we halve the bug count in safety-critical software, we can naively expect only half as many people to sustain injuries.

This view contrasts with the common opinion that human effort is the factor to optimize in system development and that computing power is essentially free. The view that efficiency and proper functioning are both paramount (and inseparable) has profound implications for system design. However, not all software is the same—it doesn't all "live" under the same conditions and with the same constraints, so we should adjust our software development techniques and tools to take that into account. Similarly, we should adjust our expectations of software developers. One size doesn't fit all when it comes to software, software development, or software developers.

I call software where failure can cause serious injury or serious economic disruption *infrastructure software*. Such software must be dependable and meet far more stringent reliability standards than "regular software." Because ordinary personal computers and smartphones are used as platforms for applications that also serve as infrastructure in my operational use of that word, the fundamental software of such platforms is itself infrastructure, as is the software used to deploy it. For example, if the software that updates the operating system on a cell phone malfunctions, crucial calls might not get through, and someone could be injured or bankrupted.

0018-9162/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

One of my inspirations for quality infrastructure software is the requirement AT&T placed on switches in its telecommunication system backbone: no more than two hours of downtime in 40 years (www.greatachievements.org/?id=3639). Hardware failure, loss of main power, or a truck colliding with the building that houses the switch isn't an excuse. To reach such goals, we need to become very serious about reliability, which is a vastly different mindset from "we must get something—anything—to market first."

DO MORE WITH LESS

Software reliability is improving. If it were not, we would have been in deep trouble already. Our civilization runs on software. If it were not for computerized systems, most of us would starve. Our dependence on computerized systems is increasing, the complexity of those systems is increasing, the amount of code in those systems is increasing, and the hardware requirements to run those systems are increasing. But our ability to comprehend them is not.

The increases in demands on hardware and software will continue: human expectation grows even faster than hardware performance.

Many of the improvements in system reliability over the years have been achieved through deep layering of software, each layer distrusting other software in the system, checking and rechecking information.[2] Furthermore, layers of software are used to monitor hardware and, if possible, compensate for hardware errors. Often, applications are interpreted or run in virtual machines that intentionally isolate them from hardware. These efforts have resulted in systems that are more reliable, but huge and less well understood.

Another contribution to the improved reliability has come from massive testing. The cost of maintaining and deploying software is increasing. The increases in demands on hardware and software will continue: human expectation grows even faster than hardware performance.

Further consumers of computing power are tools and languages aimed at making programming easier and less error-prone. Unfortunately, many such tools move decisions from design time to runtime, consuming memory and processor cycles. We compensate for lack of design-time knowledge by postponing decisions until runtime and adding runtime tests to catch any errors.

We compensate for our lack of understanding by increasing our requirements for computing power. For example, my word processing software is so "sophisticated" that I often experience delays using it on my dual-core 2.8-GHz computer. But processors are no longer getting faster. The number of transistors on a chip still increases according to Moore's law, but those transistors are used for more processors and more memory. Also, we depend more and more on energy-consuming server farms and on portable "gadgets" for which battery life is an issue. Software efficiency equates to energy conservation.

Reliability and energy efficiency require improvements in many areas. Here, I discuss implications for software. Basically, my conjecture is that we must structure our systems to be more comprehensible. For reliability and efficiency, we must compute less to get the results we need.

Doing more with less is an attractive proposition, but it isn't cost-free: it requires more work up front in the infrastructure software. We must look at high-reliability systems as our model, rather than the techniques used to produce "Internet-time Web applications." Building high-efficiency, high-reliability systems can be slow, costly, and demanding of developer skills. However, this expenditure of time, money, and talent to develop infrastructure hardware isn't just worthwhile, it's necessary. In the longer term, it might even imply savings in maintenance time and costs.

PROGRAMMING TECHNIQUES

I base the discussion here on very simple code examples, but most current infrastructure software doesn't systematically use the techniques I suggest. Code that did so would differ dramatically from what we see today and would be much better for it.

I won't try to show anything radically new, but I highlight what I hope to see deployed over the next 10 years. My examples are in C++, the programming language I know best.[3,4] Of the languages currently used for infrastructure programming, C++ most closely approximates my ideals, but I hope to see even better language and tool support over the next decade. There is clearly much room for improvement.

Compute less

On 23 September 1999, NASA lost its US$654 million *Mars Climate Orbiter* due to a navigation error. "The root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, 'Small Forces,' used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used."[5] The amount of work lost was roughly equivalent to the lifetime's work of 200 good engineers. In reality, the cost is even higher because we're deprived of the mission's scientific results until (and if) it can be repeated. The really galling aspect is that we were all taught how to avoid such errors in high school: "Always make sure the units are correct in your computations."

Why didn't the NASA engineers do that? They're indisputably experts in their field, so there must be good reasons.

No mainstream programming language supports units, but every general-purpose language allows a programmer to encode a value as a `{quantity,unit}` pair. We can encode enough of the ISO standard SI units (meters, kilograms, seconds, and so on) in an integer to deal with all of NASA's needs, but we don't because that would almost double the size of our data. Furthermore, checking the units in every computation would more than double the amount of computation needed.

Space probes tend to be both memory and compute limited, so the engineers—just as essentially everyone else in their situation has done—decided to keep track of the units themselves (in their heads, in the comments, and in the documentation). In this case, they lost. Compilers read neither documentation nor comments, and a magnitude crossed an interface without its unit and suddenly took on a whole new meaning (which was a factor of 4.45 wrong). One conclusion we can draw is that integers and floating-point numbers make for very general but essentially unsafe interfaces—a value can represent anything.

It isn't difficult to design a language that supports SI units as part of its type system. In such a language, all unit checking would be done at compile time and only unit-correct programs would get to execute:

```
Speed sp1 = 100m/9.8s; // fast for a human
Speed sp2 = 100m/9.8s2; // error: m/s2 is acceleration
Speed sp3 = 100/9.8s; // error: speed must be m/s
Acceleration acc = sp1/0.5s; // too fast for a human
```

General-purpose programming languages don't provide direct support for SI units. Some reasons are historical, but the deeper reason is that a language might support a variety of such notions (such as other unit systems, systems for naming dates, markers to help concurrency, and timing constraints). However, we can't build every useful notion into a language, so language designers, programming teachers, and practitioners have preferred the simplicity of doing nothing.

Could a tool or a specialized language supply SI units? In theory, yes, but in practice, specialized languages suffer from high development and maintenance costs and tend not to work well with other specialized tools and languages. Features work best when they're integrated into a general-purpose language and don't need separate tool chains.[6]

Interestingly, a sufficiently expressive language can achieve compile-time unit checking without language extensions. In C++, we can define `Speed` as a simple template that makes the unit (meters per second) part of the type and holds only the quantity as a runtime value. In the recent ISO C++ standard C++11, we can even define literals of those types so that the code fragment above is legal. Doing so isn't rocket science; we simply map the rules of the SI units into the general type system:

```
template<int M, int K, int S>
struct Unit {   // a unit in the MKS system
    enum { m=M, kg=K, s=S };
};

template<typename Unit> // magnitude with unit
struct Value {
    double val;     // the magnitude
    explicit Value(double d)
         : val(d) {} // construct a Value from a double
};

using Speed = Value<Unit<1,0,-1>>;        // m/s
using Acceleration = Value<Unit<1,0,-2>>; // m/s/s

using Second = Unit<0,0,1>;  // s
using Second2 = Unit<0,0,2>; // s*s

constexpr
Value<Second> operator"" s(long double d)
    // a f-p literal suffixed by 's'
{
   return Value<Second> (d);
}

constexpr
Value<Second2> operator"" s2(long double d)
      // a f-p literal  suffixed by 's2'
{
  return Value<Second2 > (d);
}
```

If you aren't familiar with modern C++, much of this code is cryptic. However, it's fundamentally simple and performs all checking and conversions at compile time. Obviously, handling the complete SI unit system takes more code—about three pages in all.

Why bother with the user-defined literals, such as `9.8s`, and `100m`? Many developers dismiss this as redundant and distracting "syntactic sugar." Although a library supporting the SI system has been available in C++ for a decade, very few people have used it. Most engineers and physicists simply refuse to write code using variants like this:

```
// a very explicit notation (quite verbose):
Speed sp1 = Value<1,0,0> (100)/ Value<0,0,1> (9.8);

// use a shorthand notation:
Speed sp1 = Value<M> (100)/ Value<S> (9.8);

// abbreviate further still:
Speed sp1 = Meters(100)/Seconds(9.8);

Speed sp1 = M(100)/S(9.8); // this is getting cryptic
```

Notation matters. SI units are important and should be supported, but so should a variety of other notions. The fundamental point here is that we can improve code quality without adding runtime costs. The use of a static type system improves code quality by reducing the number of errors and moves checking to compile time. In fact, we can move much simple computation to compile time.

Compile-time computation has been done in a variety of languages for decades. Before that (and sometimes still)

developers simply precomputed answers and added them to the source code or as data inputs, which is rather ad hoc. In C, compile-time computation implies a rat's nest of complicated and error-prone macros. Such computation is essentially untyped because most information must be encoded as integers. For infrastructure code, I suggest a more systematic and structured approach: type-rich programming at compile time.[7] The SI units example is an illustration.

Compile-time evaluation (and immutability in general) becomes more important as more systems become concurrent: You can't have a data race on a constant.

Access memory less

When I first wrote microcode to squeeze the last bit of efficiency out of a processor, a good rule of thumb was that the system could execute nine instructions while waiting for a memory read to complete. Today, that factor is 200 to 500, depending on the architecture. Memory has become relatively slower. In response, hardware architects have added systems of registers, pipelines, and caches to keep the instructions flowing. This has major implications for software: How do I organize my code and data to minimize memory usage, cache misses, and so on? My first-order answer is

We can improve code quality without adding runtime costs.

- don't store data unnecessarily,
- keep data compact, and
- access memory in a predictable manner.

This again has implications on software design. Consider a simple example: generate N random integers and insert them into a sequence so that each is in its proper position in the numerical order. For example, if the random numbers are 5, 1, 4, and 2, the sequence should grow like this:

```
5
1 5
1 4 5
1 2 4 5
```

Once the N elements are in order, we remove them one at a time by selecting a random position in the sequence and removing the element there. For example, if we choose positions 1, 2, 0, and 0 (using 0 as the origin), the sequence should shrink like this:

```
1 2 4 5
1 4 5
1 4
4
```

Now, for which N is it better to use a linked list than a vector (or an array) to represent the sequence? If we naively apply complexity theory, that answer will be something like, "Is this a trick question? A list, of course!" We can insert an element into and delete from a linked list without moving other elements. In a vector, every element after the position of an inserted or deleted element must be moved. Worse still, if you don't know the maximum number of elements in advance, it's occasionally necessary to copy the entire vector to make room for another element.

Depending on the machine architecture and the programming language, the answer will be that the vector is best for small to medium values of N. When I ran the experiment on my 8-Gbyte laptop, I found N to be much larger than 500,000. The red line in Figure 1 shows the time taken for the list, and the blue line the time taken by the vector.

This isn't a subtle difference. The x-axis is in 100,000 elements, and the y-axis in seconds. So for "small lists," a vector is a better representation of a list than a linked structure. This is also true for numbers too small to show on this graph.

Why? First, it takes 4 bytes to store a 4-byte integer in a vector, but it takes 12 bytes to store it in a doubly linked list (assuming 4-byte pointers as links). Saying "list" tripled the memory requirement. Actually, it's worse because many general-purpose list types store each element as a separate object in dynamic (heap, free store) memory, adding another word or two of memory overhead per element. The green line in Figure 1 is a list that I optimized by preallocating space for elements and where each element wasn't a separate object. This demonstrates that even though allocation overheads are significant, they don't explain the vector's fundamental advantage.

Not only are the list nodes large, they're scattered in memory, implying that when we traverse the list to find a position for insertion or deletion, we randomly access memory locations in the area that stored the list, causing cache misses. On the other hand, the hardware really likes the vector's sequential access of words in memory. In an attempt at fairness, I didn't use a binary search to speed up insertion into the vector. Nor did I use random access to find the deletion point in the vector version. This keeps the number of elements traversed the same for all versions. In fact, I used identical generic code for the vector and the lists.

Is this an academic curiosity? No. Infrastructure application developers tell me that compactness and predictable access patterns are essential for efficiency. Power consumption is roughly proportional to the number of memory accesses, so the red (list) and blue (vector) lines are first-order approximations to the drain on a smartphone battery or the number of server farms needed.

We should prefer sequential access of compact structures, not thoughtlessly use linked structures, and avoid general memory allocators that scatter logically related data. We must measure to avoid being blind-sided by unexpected phenomena. Our systems are too complex for us to guess about efficiency and use patterns.

You might say, "But I don't use sequences of 100,000 elements." Neither do I (most of the time), but using 10,000 lists of 100 elements has the same efficiency implications. To developers who write code for huge server farms, 100,000 items are barely noticeable. Consider what's needed to recognize a repeat visitor to a major website and retrieve that person's preferences in time to display a personalized welcome screen. Compactness goes hand in hand with efforts to subdivide larger datasets as well as with the design of algorithms to ensure concurrent execution.[8]

For a small system such as an embedded processor, the differences between compact and linked structures are significant even for small datasets. Even individual object layouts can produce efficiency effects. Consider a simple vector of two-dimensional points:

```
vector<Point> vp = {
   Point{1,2}, Point{3,4}, Point{5,6}, Point{7,8}
};
```

We can represent this in memory as a compact structure with a handle, as in Figure 2, where the blue box represents the overhead required to place memory in dynamic storage. This compact layout is found in a traditional systems programming language, such as C or C++. If necessary, it's possible—at the cost of some flexibility—to eliminate the handle and the dynamic storage overhead.

Alternatively, we can represent the vector as a tree structure, as in Figure 3. This layout is found in a language that doesn't emphasize compact representation, such as Java or Python. The fundamental reason for the difference is that user-defined abstractions (class objects) in such languages are allocated in dynamic memory and accessed through references. The compact representation is 11 words, out of which nine are required data (the eight coordinates plus the number of points). The tree representation is 21 words. In the compact representation, access to a coordinate point requires one indirection; in the tree representation, access requires three indirections.

For languages that are not systems programming languages, getting the compact layout involves avoiding the abstraction mechanisms that make such languages attractive for applications programming.

Practice type-rich programming

So far, I've focused primarily on efficiency, but efficiency is irrelevant if the code contains critical errors. Addressing the issue of correctness requires making progress on two fronts:

Figure 1. List versus vector timings.

Figure 2. Compact representation.

Figure 3. Linked representation.

- eliminate programming errors (make such errors less likely and more easily spotted), and
- make it easier to analyze a program for design errors (in general, make it easier to build tools that manipulate programs).

The distinction between a design error and a programming error isn't completely clear. My practical definition is that programming errors are those that can be caught by a good production compiler; they involve inconsistent use of values and don't require extensive computation to detect. Design errors involve plausible-looking code that just happens to do the wrong thing. Catching design errors requires tools that "understand" the semantics of higher-level operations. Even a compiler can be such a tool if semantic

information is encoded in the type system.

Consider

```
void increase_to(double speed);  // speed in m/s
```

This is an error waiting to happen. Maybe we understand the requirements for an argument, but they only appear in comments and other documentation. Is `increase_to(7.2)` correct? Does `7.2` represent a speed? If so, in what units? A better try would be

```
void increase_to(Speed s);
```

Given a reasonable definition of `Speed`, `increase_to(7.2)` is an error and `increase_to(72m/10s)` is likely to be correct. This isn't just an issue of units; I have a philosophical problem with parameters of near-universal types. For example, an integer can represent just about everything. Consider

```
// construct a rectangle:
Rectangle(int x, int y, int h, int w);
```

We want to practice type-rich programming, but we also want to minimize the size of the implementation by using only a few fundamental types.

What does `Rectangle (100,200,50,100)` mean? Are `h` and `w` the coordinates for the bottom right corner or a height and a width? To avoid errors, we should be more specific about `Rectangle`'s requirements on its arguments:

```
Rectangle(Point xy, Area hv);
```

I can now write the clearer and less error-prone

```
Rectangle(Point(100,200), Area(50,100));
```

I can also add a second version:

```
Rectangle(Point top_left, Point bottom_right);
```

and use either of the following to suit my needs and preferences:

```
Rectangle(Point(100,200), Area(50,100));
Rectangle(Point(100,200), Point(150,300));
```

To be relevant for infrastructure development, simple user-defined types (such as `Point` and `Speed`) may not impose overheads compared to built-in types.

Use libraries

Type-rich programming must reflect an overall design philosophy or the software descends into a mess of incompatible types, incompatible styles, and replication. For example, overuse of simple types (such as integers and character stings) to encode arbitrary information hides a program's structure from human readers and analysis tools, but a function specified to accept any `int` will perform its action on integers representing a variety of things. This generality makes functions accepting general types useful in many contexts. Generality makes it easier to design and implement libraries for noncritical uses. It avoids unnecessary replication of effort and code.

We want to practice type-rich programming, but we also want to minimize the size of the implementation by using only a few fundamental abstractions. Object-oriented programming resolves this dilemma by organizing related types into hierarchies, whereas generic programming tackles it by generalizing related algorithms and data structures through (explicit or implicit) parameterization. However, each style of programming handles only a subset of the desirable generalizations: only some relations are hierarchical and only some variation can be conveniently expressed through parameterization. The evolution of languages over the past few years bears this out. For example, Java and C# have added some support for generic programming to their object-oriented cores.

What kinds of libraries are suitable for infrastructure code? A library should encourage type-rich programming to ease human comprehension and tool use, compact data structures, and minimal computation. It should aid in writing efficient and maintainable software, and not encourage bloatware. In particular, it shouldn't "bundle" facilities so that using one library component forces the inclusion of other, potentially unused components. Although the zero-overhead principle—"what you don't use, you don't pay for"—is remarkably difficult to follow when designing tools and systems, it's extremely worthwhile as a goal. Composing solutions to problems out of separately developed library and application components should be easy and natural.

Many aspects of libraries are determined by the type systems of their implementation languages. A language choice determines the overheads of the basic operations and is a major factor in the style of libraries. Obviously, library design for infrastructure is a more suitable topic for a series of books than a section of an article, but short examples can illustrate the role of types.

I see a type system primarily as a way of imposing a definite structure—a technique for specifying interfaces so that a value is always manipulated according to its definition. Without a type system, all we have are bits, with their meaning assigned by any piece of code that accesses them. With a type system, every value has a type. Moreover, every operation has a type and can accept only operands of its required argument types. It isn't possible to realize this ideal for every line of code in every system because of the need to access hardware and communicate between separately developed and maintained systems. Further-

more, backward compatibility requirements for languages and protocols pose limitations. However, type safety is an unavoidable ideal. Encoding a program's static structure in the type system (ensuring that every object has a type and can hold only the values of its type) can be a major tool for eliminating errors.

Programmers can and do vigorously disagree about the meaning of "type" and its purpose. I tend to emphasize a few significant benefits:

- more specific interfaces (relying on named types), implying early error detection;
- opportunities for terse and general notation (such as + for addition for any arithmetic type or `draw()` for all shapes);
- opportunities for tool building that rely on high-level structure (test generators, profilers, race condition finders, and timing estimators); and
- improved optimization (for example, references to objects of unrelated types can't be aliases).

We can classify widely used languages by their support for types:

- *Languages that provide only a fixed set of types and no user-defined types (for example, C and Pascal).* Records (structs) provide representations for composite values and functions provide operations. Popular built-in types (such as integers, floating-point numbers, and strings) are overused to specify interfaces with no explicit high-level semantics. A trivial type system can catch only trivial errors.
- *Languages that provide user-defined types (classes) with compile-time checked interfaces (such as Simula, C++, and Java).* They also tend to support runtime polymorphism (class hierarchies) for added flexibility and extensibility. Very general interfaces (for example, `Object`) are often overused to specify interfaces with no explicit high-level semantics. Semantically meaningful operations, such as initialization and copy can be associated with user-defined types.
- *Languages that provide user-defined types (classes) with runtime type checking (such as Smalltalk, JavaScript, and Python).* An `Object` can hold values of any type. This implies overly general interfaces.

The demands of correctness and efficiency will push infrastructure developers toward a disciplined use of the second alternative: rich, extensible type systems with named, statically checked, and semantically meaningful interfaces.

Alternative one, writing code without serious use of user-defined types, leads to hard-to-comprehend, verbose results with few opportunities for higher-level analysis. However, this kind of code (usually written in C or low-level C++) is popular because it's easy to teach the language basics (the complexities disappear into the application code) and provide low-level analysis.

There's a widespread yet mistaken belief that only low-level, messy code can be efficient. I once gave a presentation of a C++ linear-algebra library that achieved astounding efficiency because it used a type system that allowed it to eliminate redundant temporaries and apply optimized operations by "knowing" (from the static type system) the fundamental properties of some matrices.[9] Afterward, I was repeatedly asked, "But how much faster would it run if it was rewritten in C?" Many developers equate "low level" with "fast" out of naiveté or from experience with complicated bloatware.

Relying heavily on runtime resolution or interpretation does not provide the maintainability and efficiency needed for infrastructure.

Alternative three, relying heavily on runtime resolution or interpretation, doesn't provide the maintainability and efficiency needed for infrastructure. Too many decisions are hidden in conditional statements and calls to overly general interfaces. Obviously, I'm not saying that JavaScript (or whatever) is never useful, but I do suggest that the JavaScript engine should be written in a language more suitable for systems programming (as it invariably is).

One of the advantages of dynamic typing is that it (typically) provides "duck typing" ("if it walks like a duck and quacks like a duck, it's a duck," or, in more technical terms, values have types, but objects do not—an object's behavior is determined by the value it holds). This can be used to provide more general and flexible libraries than interface-based class hierarchies. However, duck typing is suspect in infrastructure code; it relies on weakly specified, very general interfaces. This can result in unexpected semantics and need for runtime checking. It simplifies debugging but complicates systematic testing. Runtime typing carries heavy costs—often a factor of 10 to 50 (http://shootout.alioth.debian.org): objects are larger because they must carry type information, resolving types at runtime implies extra computation compared to less dynamic alternatives, and optimization is inhibited because interfaces must be able to handle objects of every type, if only to give an error message.

However, a statically typed language can provide much of the desired flexibility. In particular, duck typing, as provided by templates, is the backbone of generic programming in C++. It's also the practical basis of the ISO C++ standard library and most high-efficiency C++ (including

generic code optimized for particular hardware through parameters). For example,

```
template<typename Container, typename Predicate>
typename Container::iterator
find_if(Container& c, Predicate pred)
      // return an iterator to the first element in c
      // for which pred(element) is true
{
       auto p = c.begin();
       while(p!=c.end() && !pred(*p)) ++p;
       return p;
}

void user (vector<string>& v,
    list<Record*>& lst,
    Date threshold)
{
       auto p = find_if(v,
                 [](const string& s)
                         { return s<"Fulcrum"; }
          );
       // …
       auto q = find_if(lst,
                 [](const Record* p)
                         { return threshold<p->date; }
)
    // …
}
```

Errors that manifest as exceptions in a dynamically checked language become compile-time errors.

Here, the generic function `find_if()` is called with dramatically different arguments: a `vector` of `strings`, and a `list` of pointers to `Records`. The search criteria are presented as lambda expressions. Every modern C++ compiler can generate code for this that equals or outperforms elaborate hand-coded alternative implementations.[10] Note the way that code from different sources is used to compose the solution:

- the `list`, `vector`, and `string` from the ISO C++ standard library;
- the `find_if()` and `Record` from my application; and
- the lambda expressions written specifically in `user()`.

There's no need for "glue code," use of inheritance, wrapping of built-in types (such as `Record*`) into self-describing objects, or dynamic resolution. This style of generic code is often called STL-style.[11]

However, the interfaces are still underspecified—in particular, the definition of `find_if()` didn't say that its first argument had to be a `Container`. Errors that manifest as exceptions in a dynamically checked language become compile-time errors. Unfortunately, the reporting of those errors can be very obscure. Consequently, much effort in modern C++ is aimed at better specification of template arguments.[12,13] We also need to add semantic constraints[14]—for example, a generic function that uses a copy operation needs to know whether that copy is deep or shallow. Similarly, a generic function that uses a + needs to know whether that + is an arithmetic function or something else, such as a concatenation.

An explicit representation of system structure can simultaneously increase flexibility, improve error diagnostics, and improve efficiency. Correctness and efficiency are closely related properties of a well-specified system. It seems that extensive use of a rich type system leads to shorter code independently of whether the type system is static or dynamic. Obviously, we can also use types to gain the usual aspects of an object-oriented system—encapsulation, runtime polymorphism, and inheritance—but we can't go overboard and try to rely completely on static structure. That isn't necessary for noncritical parts of a system, and a complete absence of runtime checks would leave the system open to catastrophic errors caused by malfunctioning hardware. Dealing gracefully with hardware failure is a crucial characteristic of infrastructure systems.

Prefer highly structured code

It isn't enough to be disciplined in our specification of data structures and interfaces: we must also simplify our code logic. Complicated control structures are as dangerous to efficiency and correctness as are complicated data structures.

When choosing among alternatives in code, we often have three options:

- *Overloading.* Select among alternative functions based on the static type of arguments, for example, `f(x)`.
- *Virtual function call.* Select based on the dynamic type of a class object in a class hierarchy, for example, `x.f()`.
- *Selection statement.* Select on a value with an if-statement or a switch-statement, for example, `if (x.c) f1(x) else f2(x)`.

Using an if-statement is the least structured and most flexible option. However, we should choose the more structured and easier-to-optimize alternatives whenever possible. The hierarchy option can be most useful, but was seriously overused in the past couple of decades. Not every class naturally belongs in a hierarchy, not every class benefits from the coupling that a hierarchy introduces, and not every class is improved by the possibility of adding to it through derivation (subclassing). Of the alternatives, a call of an ordinary function is the easiest to understand—it's also often easy to inline. Consequently, overloading—statically selecting a function based on argument types—can lead to major efficiency advantages compared to indirect calls (as used to select among alternatives in a class hierarchy).

To become significantly more reliable, code must become more transparent. In particular, nested conditions

and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs.

Consider a real example (taken from Sean Parent): in many user interfaces, an item of a sequence can be moved to a different position in the sequence by dragging its screen representation over a screen representation of the sequence. The original solution (after cleanup and simplification) involved 25 lines of code with one loop, three tests, and 14 function calls. Its author considered it messy and an attempt to simplify the code was made. The result was

```
void drag_item_to(Vector& v,
                  Vector::iterator source,
                  Coordinate p)
{
   auto dest = find_if(v.begin(), v.end(), contains(p));
    if (source < dest)
        // from before insertion point:
        rotate(source, source+1, dest);
    else
        // from after insertion point:
        rotate(dest, source, source+1);
}
```

That is, find the insertion point using the standard library algorithm `find_if` and then move the element to the insertion point using the standard library algorithm `rotate`. Obvious once you see it!

The original solution was complicated enough to raise worries about correctness, plus it was completely special-purpose. What might happen to such hard-to-understand code during maintenance? The improved code is shorter, simpler, more general, and runs faster. It uses only well-documented standard library facilities known to experienced C++ developers.

But why move only one element? Some user interfaces allow selecting and moving a collection of elements. And why use application-specific notions such as `Vector` and `Coordinate`? Dealing with this more general problem turns out to be simpler still:

```
template <typename Iter, typename Predicate>
pair<Iter, Iter>
gather(Iter first, Iter last, Iter p, Predicate pred)
// move e for which pred(e) to the insertion point p
{
     return make_pair(
        // from before insertion point:
        stable_partition(first, p, !bind(pred, _1)),
        // from after insertion point:
        stable_partition(p, last, bind(pred, _1))
      );
}
```

The `Predicate` determines which elements are moved. Admittedly, `stable_partition` tends to be used only by specialists, but its meaning isn't obscure: `stable_partition(first,last,pred)` places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that don't satisfy it. Note the absence of explicit tests and loops.

Expressing code in terms of algorithms rather than hand-crafted, special-purpose code can lead to more readable code that's more likely to be correct, often more general, and often more efficient. Such code is also far more likely to be used again elsewhere. However, making algorithm-based code mainstream requires a culture change among systems programmers. Analyzing problems with the aim of coming up with general, elegant, and simple solutions isn't emphasized in the education of or selection of systems programmers. Too many programmers take pride in complicated solutions (invariably assumed to be efficient), and too many managers are more impressed with such code than with simpler alternatives.

Complicated control flows also complicate resource management. A resource is anything that has to be acquired and (explicitly or implicitly) released after use. Memory, locks, sockets, and thread handles are examples.[15] For efficiency and local reasoning, it's preferable to hold a resource for as short a time as possible and for that to be obvious.

Consider a slightly artificial simple example. The function `algo` updates all records from a `vector` that matches a predicate and returns a list of indices of the updated records:

```
vector<int>  algo(vector<Record>& vr, Predicate pred)
   // update vr[i] if pred(vr[i])
{
      vector<int> indices;   // updated records
      for (int i = 0; i<v.size(); ++i)
        if (pred(vr[i]))  {
           unique_lock lck(vr[i].mtx); // acquire mutex
          // update vr[i]
          indices.push_back(i); // record the update
        }
      return indices;
}
```

Here, I assume concurrency, so that some form of mutual exclusion is necessary. The `vector` and `unique_lock` come from the C++ 11 standard library; `Record` and `Predicate` are assumed to be defined by the application. Two resources are required here: the memory for elements of the `vector` of indices, and the `mutex` members (`mtx`). The lifetime of the `unique_lock` determines how long its `mutex` is held. Both `vector` and `unique_lock` hold onto their resource while it's in scope and release it when that scope is exited.

Note that there is no user code for handling error returns from `algo`: `vector` and `unique_lock` release their resources even if an exit from their scope is done by a return or by throwing an exception. Explicit resource management in the presence of errors can be a major source of complexity—for example, nested conditions or nested exception handlers—and obscure errors. Here, the resource management is made part of the semantics of the types used (`vector` and `unique_lock`) and is therefore implicit. Importantly, the resource management is still

local and predictable: it follows the lexical scope rules that every programmer understands.

Naturally, not all resource management can be scoped and not all code is best expressed as highly stylized algorithms. However, it's essential to keep simple things simple. Doing so leaves us with more time for the more complex cases. Generalizing all cases to the most complex level is inherently wasteful.

WHY WORRY ABOUT CODE?

Do we need traditional programmers and traditional programs? Can't we just express what we want in some high-level manner, such as pictures, diagrams, high-level specification languages, formalized English text, or mathematical equations, and use code-generation tools to provide compact data structures and fast code? That question reminds me of an observation by Alan Perlis: "When someone says, 'I want a programming language in which I need only say what I wish done,' give him a lollipop" (www.cs.yale.edu/quotes.html).

True, we've made progress—Modelica, for example (https://modelica.org)—but generative techniques work best for well-understood and limited application domains, especially for domains with a formal model, such as relational databases, systems of differential equations, and state machines. Such techniques have worked less well in the infrastructure area. Many tasks aren't mathematically well-defined, resource constraints can be draconian, and we must deal with hardware errors.

Many tasks aren't mathematically well-defined, resource constraints can be draconian, and we must deal with hardware errors.

I see no alternative to programmers and designers directly dealing with code for most of these tasks. Where something like model-driven development looks promising, the generated code should be well-structured and human readable—going directly to low-level code could easily lead to too much trust in nonstandard tools. Infrastructure code often "lives" for decades, which is longer than most tools are stable and supported.

I'm also concerned about the number of options for code generation in tools. How can we understand the meaning of a program when it depends on 500 option settings? How can we be sure we can replicate a result with next year's version of the tool? Even compilers for formally standardized language aren't immune to this problem.

Could we leave our source code conventionally messy? Alternatively, could we write code at a consistently high level isolated from hardware issues? That is, could we rely on "smart compilers" to generate compact data structures, minimize runtime evaluation, ensure inlining of operations passed as arguments, and catch type errors from source code in languages that don't explicitly deal with these concepts?

These have been interesting research topics for decades, and I fear that they will remain so for even more decades. Although I'm a big fan of better compilers and static code analysis, I can't recommend putting all of our eggs into those baskets. We need good programmers dealing with programming languages aimed at infrastructure problems.

I suspect that we can make progress on many fronts, but for the next 10 years or so, relying on well-structured, type-rich source code is our best bet by far.

THE FUTURE

Niels Bohr said, "It is hard to make predictions, especially about the future." But, of course, that's what I've done here. If easy-to-use processing power continues to grow exponentially, my view of the near future is probably wrong. If it turns out that most reliability and efficiency problems are best solved by a combination of lots of runtime decision-making, runtime checking, and a heavy reliance on metadata, then I have unintentionally written a history paper. But I don't think I have: correctness, efficiency, and comprehensibility are closely related. Getting them right requires essentially the same tools.

Low-level code, multilevel bloated code, and weakly structured code mar the status quo. Because there's a lot of waste, making progress is relatively easy: much of the research and experimentation for many improvements have already been done. Unfortunately, progress is only relatively easy; the amount of existing code and the number of programmers who are used to it seriously complicate any change.

Hardware improvements make the problems and costs resulting from isolating software from hardware far worse than they used to be. For a typical desktop machine,

- 3/4ths of the MIPS are in the GPU;
- from what's left, 7/8ths are in the vector units; and
- 7/8ths of that are in the "other" cores.

So a single-threaded, nonvectorized, non-GPU-utilizing application has access to roughly 0.4 percent of the compute power available on the device (taken from Russell Williams). Trends in hardware architecture will increase this effect over the next few years, and heavy use of software layers only adds to the problem.

I can't seriously address concurrency and physical distribution issues here, but we must either find ways of structuring infrastructure software to take advantage of heterogeneous, concurrent, vectorized hardware or face massive waste. Developers are already addressing this

problem with careful programming using architecture-specific concurrency primitives for the code closest to the hardware and using threads libraries for system-level concurrent tasks. Getting such code correct, efficient, maintainable, and portable to next year's hardware adds to the requirements for structure, compactness, and code formalization (algorithms).

Locality and compactness are key to comprehensibility and efficiency for nonuniform processors and memory. We need to be able to reason about code without knowledge of the complete system. Shared resources are poison to such reasoning, making local resource management essential.

We also need more and better tools for specifying requirements, analyzing code, verifying system properties, supporting testing, measuring efficiency, and so on. However, we shouldn't expect miracles: designing, implementing, and using such tools tends to be difficult.[16-20] My conjecture is that much of the complexity of such tools comes from the messiness of the code they're supposed to deal with. Whatever else we do, we must also clean up our code.

What should be done? There's a saying that "real software engineers write code." I wish that were true. Production of quality code should be elevated to a central role in software development. A software developer should be able to proudly display a portfolio of work (including code), much the way an artist or architect does. We should read and evaluate code as part of every project. We should distinguish between infrastructure code and application code. Often, the two areas need different languages, tools, and techniques. Sometimes, that's the case even when we use the same language for both infrastructure and applications. The role of static typing should be increased.

All of this has implications for education: you can't expect a person trained to deliver applications quickly in JavaScript to design and implement an infrastructure component (say, a JavaScript engine or a JVM) in C++ using the same skills. We need to specialize at least part of our computer science, software engineering, and IT curricula.[21] I strongly prefer general-purpose programming languages, but no single language is ideal for everything. We should master and use multiple languages, often as part of a single system.

Infrastructure developers should be highly skilled professionals, not assembly-line workers, and not just scholars. The mathematical basis of their work must be strengthened. We need developers with good analytical skills, some ability in algebraic reasoning (calculus alone isn't sufficient), and enough familiarity with statistics to understand systems through measurement. Obviously, algorithms, data structures, machine architecture, and operating systems must remain core subjects to provide a basis for emphasizing efficiency and reliability.

In research, we need a greater appreciation of incremental (engineering) improvements with a relevance to real-world systems. "That's just engineering, and we're computer scientists" is an attitude we can't afford. I suspect the era of transformative breakthroughs is over. We need to achieve a factor-of-two-or-three improvement in several areas, rather than trying to solve our problems with a single elusive two-orders-of-magnitude improvement from a silver bullet. C

References

1. R. Waters, "Server Farm Cost," *Financial Times*, 9 Sept. 2011; www.ft.com/cms/s/2/6e358ae0-da7a-11e0-bc99-00144feabdc0.html#axzz1YhypogYG.
2. N. Mitchell et al., "The Diary of a Datum: An Approach to Analyzing Runtime Complexity," *Proc. Library-Centric Software Design*, ACM, 2006; http://domino.watson.ibm.com/comm/research_projects.nsf/pages/sorbet.pubs.html/$FILE/diaryslides.pdf.
3. B. Stroustrup, *The C++ Programming Language* (special ed.), Addison-Wesley, 2000.
4. "Programming Languages—C++," Int'l Standard ISO/IEC 14882-2011; www-d0.fnal.gov/~dladams/cxx_standard.pdf.
5. "Mars Climate Orbiter Mishap Investigation Board Phase I Report," NASA, 10 Nov. 1999; ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.
6. B. Stroustrup, "A Rationale for Semantically Enhanced Library Languages," *Proc. Library-Centric Software Design* (LCSD 05), Rensselaer Polytechnic Inst., 2005; www.cs.rpi.edu/research/pdf/06-12.pdf.
7. G. Dos Reis and B. Stroustrup, "General Constant Expressions for System Programming Languages," *Proc. Symp. Applied Computing* (SAC 10), ACM, 2010; http://www2.research.att.com/~bs/sac10-constexpr.pdf.
8. A.A. Stepanov et al., "SIMD-Based Decoding of Posting Lists," *Proc. ACM Conf. Information and Knowledge Management* (CIKM 11), ACM, 2011; www.stepanovpapers.com/CIKM_2011.pdf.
9. J. Siek, A. Lumsdaine, and L.-Q. Lee, "Generic Programming for High-Performance Numerical Linear Algebra," *Proc. SIAM Workshop Interoperable Object-Oriented Scientific Computing*, SIAM, 1998, pp. 1-10; www.osl.iu.edu/publications/prints/1998/siek98:_siamoo.pdf.
10. B. Stroustrup, "Learning Standard C++ as a New Language," *C/C++ Users J.*, May 1999; http://www2.research.att.com/~bs/new_learning.pdf.
11. A. Stepanov and M. Lee, *The Standard Template Library*, tech. report 95-11(R.1), HP Labs, 1995.
12. G. Dos Reis and B. Stroustrup, "Specifying C++ Concepts," *Proc. Symp. Principles of Programming Languages* (POPL 06), ACM, 2006; http://www2.research.att.com/~bs/popl06.pdf.
13. D. Gregor et al., "Concepts: Linguistic Support for Generic Programming in C++," *Proc. Object-Oriented Programming,*

Systems, Languages & Applications (OOPSLA 06), ACM, 2006; http://www2.research.att.com/~bs/oopsla06.pdf.

14. A. Sutton and B. Stroustrup, "Design of Concept Libraries for C++," *Proc. Software Language Eng.* (SLE 11), 2011; http://www2.research.att.com/~bs/sle2011-concepts.pdf.
15. B. Stroustrup, "Exception Safety: Concepts and Techniques," LNCS 2022, Springer, 2001; http://www2.research.att.com/~bs/except.pdf.
16. A. Bessey et al., "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Comm. ACM*, Feb. 2010, pp. 66-75; http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext.
17. G. Dos Reis and B. Stroustrup, "A Principled, Complete, and Efficient Representation of C++," *J. Mathematics in Computer Science*, 2011; http://www2.research.att.com/~bs/macis09.pdf.
18. D. Parnas, "Really Rethinking Formal Methods," *Computer*, Jan. 2010, pp. 28-34; www.computer.org/portal/web/csdl/doi/10.1109/MC.2010.22.
19. T. Ramananandro, G. Dos Reis, and X. Leroy, "Formal Verification of Object Layout for C++ Multiple Inheritance," *Proc. Symp. Principles of Programming Languages* (POPL 11), ACM, 2011; http://gallium.inria.fr/~xleroy/publi/cpp-object-layout.pdf.
20. G.J. Holzmann, *The Spin Model Checker*, Addison-Wesley, 2003.
21. B. Stroustrup, "What Should We Teach Software Developers? Why?" *Comm. ACM*, Jan. 2010, pp. 40-42; http://cacm.acm.org/magazines/2010/1/55760-what-should-we-teach-new-software-developers-why/fulltext.

***Bjarne Stroustrup**, the College of Engineering Chair Professor in Computer Science at Texas A&M University, is the designer and original implementer of C++ and the author of several books, including* Programming—Principles and Practice Using C++ *and* The C++ Programming Language. *His research interests include distributed systems, programming techniques, software development tools, and programming languages. Stroustrup is actively involved in the ISO standardization of C++. He is a member of the US National Academy of Engineering and a fellow of IEEE and ACM. He retains a link to AT&T as an AT&T Fellow. Stroustrup received a PhD in computer science from Cambridge University, England. Contact him at bs@cs.tamu.edu; www.research.att.com/~bs.*

Selected CS articles and columns are available for free at http://ComputingNow.computer.org.

THERE NOW IS A **QUICK AND EASY** WAY TO FIND OUT ABOUT OUR **COLLECTION OF TRANSACTIONS**

Our new **"What's New in Transactions"** webpage provides an overview of our 14 peer-reviewed scholarly journals.

- Read free articles from each title,
- View and listen to new multimedia,
- Learn about recent special issues and special sections, and
- Discover open calls for papers.

It is ever changing like the research areas our journals cover, so make sure to come back frequently. Visit **http://www.computer.org/whats-new** today!

IEEE

IEEE computer society