Whenever **reserve()** may have moved the elements, any iterators into the **vector** may have been invalidated (§31.3.3).

Remember that a move operation should not throw. In the rare cases where the obvious implementation of a move might throw, we typically go out of our way to avoid that. A throw from a move operation is rare, unexpected, and damaging to normal reasoning about code. If at all possible avoid it. The standard-library **move_if_noexcept()** operations may be of help here (§35.5.1).

The explicit use of **move()** is needed because the compiler doesn't know that **elem[i]** is just about to be destroyed.

13.6.4.2 resize()

The **vector** member function **resize()** changes the number of elements. Given **reserve()**, the implementation **resize()** is fairly simple. If the number of elements increases, we must construct the new elements. Conversely, if the number of elements decrease, we must destroy the surplus elements:

```
template<class T, class A>
void vector<T,A>::resize(size_type newsize, const T& val)
{
    reserve(newsize);
    if (size()<newsize)
        uninitialized_fill(elem+size(),elem+newsize,val);    // construct new elements: [size():newsize)
    else
        destroy(elem.size(),elem+newsize);                   // destroy surplus elements: [newsize:size())
    vb.space = vb.last = vb.elem+newsize;
}
```

There is no standard **destroy()**, but that easily written:

```
template<typename In>
void destroy(In b, In e)
{
    for (; b!=e; ++b)       // destroy [b:e)
        b->~T();
}
```

13.6.4.3 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 (capacity()==size())                  // no more free space; relocate:
        reserve(sz?2*sz:8);                  // grow or start with 8
    vb.alloc.construct(&vb.elem[size()],val);    // add val at end
    ++vb.space;                              // increment size
}
```

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. However, reserve() may already have reallocated the existing elements.

This definition of push_back() contains two "magic numbers" (2 and 8). An industrial-strength implementation would not do that, but it would still have values determining the size of the initial allocation (here, 8) and the rate of growth (here, 2, indicating a doubling in size each time the vector would otherwise overflow). As it happens, these are not unreasonable or uncommon values. The assumption is that once we have seen one push_back() for a vector, we will almost certainly see many more. The factor two is larger than the mathematically optimal factor to minimize average memory use (1.618), so as to give better run-time performance for systems where memories are not tiny.

13.6.4.4 Final Thoughts

Note the absence of try-blocks in the vector implementation (except for the one hidden inside uninitialized_copy()). The changes in state were done by carefully ordering the operations so that if an exception is thrown, the vector remains unchanged or at least valid.

The approach of gaining exception safety through ordering and the RAII technique (§13.3) tends to be more elegant and more efficient than explicitly handling errors using try-blocks. 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 reserve(), safe_assign(), and push_back() examples, the opportunities for surprises are limited. It is relatively simple to look at such code and ask, "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-blocks increases this local control structure complexity and can therefore be a source of confusion and errors (§13.3). I conjecture that the effectiveness of the ordering approach and the RAII approach compared to more extensive use of try-blocks stems from the simplification of the local control flow. Simple, stylized code is easier to understand, easier to get right, and easier to generate good code for.

This 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. However, the standard does require the exception-safety guarantees as provided by the example.

13.7 Advice

[1] Develop an error-handling strategy early in a design; §13.1.
[2] Throw an exception to indicate that you cannot perform an assigned task; §13.1.1.
[3] Use exceptions for error handling; §13.1.4.2.
[4] Use purpose-designed user-defined types as exceptions (not built-in types); §13.1.1.

[5] If you for some reason cannot use exceptions, mimic them; §13.1.5.
[6] Use hierarchical error handling; §13.1.6.
[7] Keep the individual parts of error handling simple; §13.1.6.
[8] Don't try to catch every exception in every function; §13.1.6.
[9] Always provide the basic guarantee; §13.2, §13.6.
[10] Provide the strong guarantee unless there is a reason not to; §13.2, §13.6.
[11] Let a constructor establish an invariant, and throw if it cannot; §13.2.
[12] Release locally owned resources before throwing an exception; §13.2.
[13] Be sure that every resource acquired in a constructor is released when throwing an exception in that constructor; §13.3.
[14] Don't use exceptions where more local control structures will suffice; §13.1.4.
[15] Use the ''Resource Acquisition Is Initialization'' technique to manage resources; §13.3.
[16] Minimize the use of **try**-blocks; §13.3.
[17] Not every program needs to be exception-safe; §13.1.
[18] Use ''Resource Acquisition Is Initialization'' and exception handlers to maintain invariants; §13.5.2.2.
[19] Prefer proper resource handles to the less structured **finally**; §13.3.1.
[20] Design your error-handling strategy around invariants; §13.4.
[21] What can be checked at compile time is usually best checked at compile time (using **static_assert**); §13.4.
[22] Design your error-handling strategy to allow for different levels of checking/enforcement; §13.4.
[23] If your function may not throw, declare it **noexcept**; §13.5.1.1
[24] Don't use exception specification; §13.5.1.3.
[25] Catch exceptions that may be part of a hierarchy by reference; §13.5.2.
[26] Don't assume that every exception is derived from class **exception**; §13.5.2.2.
[27] Have **main()** catch and report all exceptions; §13.5.2.2, §13.5.2.4.
[28] Don't destroy information before you have its replacement ready; §13.6.
[29] Leave operands in valid states before throwing an exception from an assignment; §13.2.
[30] Never let an exception escape from a destructor; §13.2.
[31] Keep ordinary code and error-handling code separate; §13.1.1, §13.1.4.2.
[32] Beware of memory leaks caused by memory allocated by **new** not being released in case of an exception; §13.3.
[33] Assume that every exception that can be thrown by a function will be thrown; §13.2.
[34] A library shouldn't unilaterally terminate a program. Instead, throw an exception and let a caller decide; §13.4.
[35] A library shouldn't produce diagnostic output aimed at an end user. Instead, throw an exception and let a caller decide; §13.1.3.

This page intentionally left blank

14

Namespaces

The year is 787!
A.D.?
– Monty Python

- Composition Problems
- Namespaces
  Explicit Qualification; **using**-Declarations; **using**-Directives; Argument-Dependent Lookup; Namespaces Are Open
- Modularization and Interfaces
  Namespaces as Modules; Implementations; Interfaces and Implementations
- Composition Using Namespaces
  Convenience vs. Safety; Namespace Aliases; Namespace Composition; Composition and Selection; Namespaces and Overloading; Versioning; Nested Namespaces; Unnamed Namespaces; C Headers
- Advice

14.1 Composition Problems

Any realistic program consists of a number of separate parts. Functions (§2.2.1, Chapter 12) and classes (§3.2, Chapter 16) provide relatively fine-grained separation of concerns, whereas ‘‘libraries,’’ source files, and translation units (§2.4, Chapter 15) provide coarser grain. The logical ideal is *modularity*, that is, to keep separate things separate and to allow access to a ‘‘module’’ only through a well-specified interface. C++ does not provide a single language feature supporting the notion of a module; there is no module construct. Instead, modularity is expressed through combinations of other language facilities, such as functions, classes, and namespaces, and source code organization.

This chapter and the next deal with the coarse structure of a program and its physical representation as source files. That is, these two chapters are more concerned with programming in the

large than with the elegant expression of individual types, algorithms, and data structures.

Consider some of the problems that can arise when people fail to design for modularity. For example, a graphics library may provide different kinds of graphical **Shape**s and functions to help use them:

```
// Graph_lib:

class Shape { /* ... */ };
class Line : public Shape { /* ... */ };
class Poly_line: public Shape { /* ... */ };          // connected sequence of lines
class Text : public Shape { /* ... */ };              // text label

Shape operator+(const Shape&, const Shape&);          // compose

Graph_reader open(const char*);                       // open file of Shapes
```

Now someone comes along with another library, providing facilities for text manipulation:

```
// Text_lib:

class Glyph { /* ... */ };
class Word { /* ... */ };          // sequence of Glyphs
class Line { /* ... */ };          // sequence of Words
class Text { /* ... */ };          // sequence of Lines

File* open(const char*);                        // open text file

Word operator+(const Line&, const Line&);       // concatenate
```

For the moment, let us ignore the specific design issues for graphics and text manipulation and just consider the problems of using **Graph_lib** and **Text_lib** together in a program.

Assume (realistically enough) that the facilities of **Graph_lib** are defined in a header (§2.4.1), **Graph_lib.h**, and the facilities of **Text_lib** are defined in another header, **Text_lib.h**. Now, I can ''innocently'' **#include** both and try to use facilities from the two libraries:

```
#include "Graph_lib.h"
#include "Text_lib.h"
// ...
```

Just **#include**ing those headers causes a slurry of error messages: **Line**, **Text**, and **open()** are defined twice in ways that a compiler cannot disambiguate. Trying to use the libraries would give further error messages.

There are many techniques for dealing with such *name clashes*. For example, some such problems can be addressed by placing all the facilities of a library inside a few classes, by using supposedly uncommon names (e.g., **Text_box** rather than **Text**), or by systematically using a prefix for names from a library (e.g., **gl_shape** and **gl_line**). Each of these techniques (also known as ''workarounds'' and ''hacks'') works in some cases, but they are not general and can be inconvenient to use. For example, names tend to become long, and the use of many different names inhibits generic programming (§3.4).

14.2 Namespaces

The notion of a *namespace* is provided to directly represent the notion of a set of facilities that directly belong together, for example, the code of a library. The members of a namespace are in the same scope and can refer to each other without special notation, whereas access from outside the namespace requires explicit notation. In particular, we can avoid name clashes by separating sets of declarations (e.g., library interfaces) into namespaces. For example, we might call the graph library **Graph_lib**:

```
namespace Graph_lib {
        class Shape { /* ... */ };
        class Line : public Shape { /* ... */ };
        class Poly_line: public Shape { /* ... */ };          // connected sequence of lines
        class Text : public Shape { /* ... */ };              // text label

        Shape operator+(const Shape&, const Shape&);          // compose

        Graph_reader open(const char*);                       // open file of Shapes
}
```

Similarly, the obvious name for our text library is **Text_lib**:

```
namespace Text_lib {
        class Glyph { /* ... */ };
        class Word { /* ... */ };           // sequence of Glyphs
        class Line { /* ... */ };           // sequence of Words
        class Text { /* ... */ };           // sequence of Lines

        File* open(const char*);                        // open text file

        Word operator+(const Line&, const Line&);       // concatenate
}
```

As long as we manage to pick distinct namespace names, such as **Graph_lib** and **Text_lib** (§14.4.2), we can now compile the two sets of declarations together without name clashes.

A namespace should express some logical structure: the declarations within a namespace should together provide facilities that unite them in the eyes of their users and reflect a common set of design decisions. They should be seen as a logical unit, for example, ''the graphics library'' or ''the text manipulation library,'' similar to the way we consider the members of a class. In fact, the entities declared in a namespace are referred to as the members of the namespace.

A namespace is a (named) scope. You can access members defined earlier in a namespace from later declarations, but you cannot (without special effort) refer to members from outside the namespace. For example:

```
class Glyph { /* ... */ };
class Line { /* ... */ };

namespace Text_lib {
        class Glyph { /* ... */ };
        class Word { /* ... */ };       // sequence of Glyphs
```

```
        class Line { /* ... */ };// sequence of Words
        class Text { /* ... */ };// sequence of Lines

        File* open(const char*);                    // open text file

        Word operator+(const Line&, const Line&);   // concatenate
    }

    Glyph glyph(Line& ln, int i);    // ln[i]
```

Here, the **Word** and **Line** in the declaration of **Text_lib::operator+()** refer to **Text_lib::Word** and **Text_lib::Line**. That local name lookup is not affected by the global **Line**. Conversely, the **Glyph** and **Line** in the declaration of the global **glyph()** refer to the global **::Glyph** and **::Line**. That (nonlocal) lookup is not affected by **Text_lib**'s **Glyph** and **Line**.

To refer to members of a namespace, we can use its fully qualified name. For example, if we want a **glyph()** that uses definitions from **Text_lib**, we can write:

```
Text_lib::Glyph glyph(Text_lib::Line& ln, int i);    // ln[i]
```

Other ways of referring to members from outside their namespace are **using**-declarations (§14.2.2), **using**-directives (§14.2.3), and argument-dependent lookup (§14.2.4).

14.2.1 Explicit Qualification

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
}
```

For example:

```
namespace Parser {
    double expr(bool);      // declaration
    double term(bool);
    double prim(bool);
}

double val = Parser::expr();    // use

double Parser::expr(bool b)    // definition
{
    // ...
}
```

We cannot declare a new member of a namespace outside a namespace definition using the qualifier syntax (§iso.7.3.1.2). The idea is to catch errors such as misspellings and type mismatches, and also to make it reasonably easy to find all names in a namespace declaration. For example:

```
void Parser::logical(bool);     // error: no logical() in Parser
double Parser::trem(bool);      // error: no trem() in Parser (misspelling)
double Parser::prim(int);       // error: Parser::prim() takes a bool argument (wrong type)
```

A namespace is a scope. The usual scope rules hold for namespaces. 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. The global scope is a namespace and can be explicitly referred to using **::**. For example:

```
int f();            // global function

int g()
{
    int f;          // local variable; hides the global function
    f();            // error: we can't call an int
    ::f();          // OK: call the global function
}
```

Classes are namespaces (§16.2).

14.2.2 using-Declarations

When a name is frequently used outside its namespace, it can be a bother to repeatedly qualify it with its namespace name. Consider:

```
#include<string>
#include<vector>
#include<sstream>

std::vector<std::string> split(const std::string& s)
    // split s into its whitespace-separated substrings
{
    std::vector<std::string> res;
    std::istringstream iss(s);
    for (std::string buf; iss>>buf;)
        res.push_back(buf);
    return res;
}
```

The repeated qualification **std** is tedious and distracting. In particular, we repeat **std::string** four times in this small example. To alleviate that we can use a **using**-declaration to say that in this code **string** means **std::string**:

```
using std::string;  // use "string" to mean "std::string"

std::vector<string> split(const string& s)
    // split s into its whitespace-separated substrings
{
    std::vector<string> res;
    std::istringstream iss(s);
```

```
        for (string buf; iss>>buf;)
            res.push_back(buf);
        return res;
    }
```

A **using**-declaration introduces a synonym into a scope. It is usually a good idea to keep local synonyms as local as possible to avoid confusion.

When used for an overloaded name, a **using**-declaration applies to all the overloaded versions. For example:

```
    namespace N {
        void f(int);
        void f(string);
    };

    void g()
    {
        using N::f;
        f(789);          // N::f(int)
        f("Bruce");      // N::f(string)
    }
```

For the use of **using**-declarations within class hierarchies, see §20.3.5.

14.2.3 using-Directives

In the **split()** example (§14.2.2), we still had three uses of **std::** left after introducing a synonym for **std::string**. Often, we like to use every name from a namespace without qualification. That can be achieved by providing a **using**-declaration for each name from the namespace, but that's tedious and requires extra work each time a new name is added to or removed from the namespace. Alternatively, we can use a **using**-directive to request that every name from a namespace be accessible in our scope without qualification. For example:

```
    using namespace std;    // make every name from std accessible

    vector<string> split(const string& s)
        // split s into its whitespace-separated substrings
    {
        vector<string> res;
        istringstream iss(s);
        for (string buf; iss>>buf;)
            res.push_back(buf);
        return res;
    }
```

A **using**-directive makes names from a namespace available almost as if they had been declared outside their namespace (see also §14.4). Using a **using**-directive to make names from a frequently used and well-known library available without qualification is a popular technique for simplifying code. This is the technique used to access standard-library facilities throughout this book. The standard-library facilities are defined in namespace **std**.

Within a function, a **using**-directive can be safely used as a notational convenience, but care should be taken with global **using**-directives because overuse can lead to exactly the name clashes that namespaces were introduced to avoid. For example:

```
namespace Graph_lib {
        class Shape { /* ... */ };
        class Line : Shape { /* ... */ };
        class Poly_line: Shape { /* ... */ };    // connected sequence of lines
        class Text : Shape { /* ... */ };        // text label

        Shape operator+(const Shape&, const Shape&);   // compose

        Graph_reader open(const char*);   // open file of Shapes
}

namespace Text_lib {
        class Glyph { /* ... */ };
        class Word { /* ... */ };       // sequence of Glyphs
        class Line { /* ... */ };       // sequence of Words
        class Text { /* ... */ };       // sequence of Lines

        File* open(const char*);  // open text file

        Word operator+(const Line&, const Line&);   // concatenate
}

using namespace Graph_lib;
using namespace Text_lib;

Glyph gl;                 // Text_lib::Glyph
vector<Shape*> vs;        // Graph_lib::Shape
```

So far, so good. In particular, we can use names that do not clash, such as **Glyph** and **Shape**. However, name clashes now occur as soon as we use one of the names that clash – exactly as if we had not used namespaces. For example:

```
Text txt;                                  // error: ambiguous
File* fp = open("my_precious_data");       // error: ambiguous
```

Consequently, we must be careful with **using**-directives in the global scope. In particular, don't place a **using**-directive in the global scope in a header file except in very specialized circumstances (e.g., to aid transition) because you never know where a header might be **#include**d.

14.2.4 Argument-Dependent Lookup

A function taking an argument of user-defined type **X** is more often than not defined in the same namespace as **X**. 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 (called *argument-dependent lookup* or simply ADL) saves the programmer a lot of typing compared to using explicit qualification, yet it doesn't pollute the namespace the way a **using**-directive (§14.2.3) can. It is especially useful for operator operands (§18.2.5) and template arguments (§26.3.5), 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 (§12.3) 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 §18.2.5.

When a class member invokes a named function, other members of the same class and its base classes are preferred over functions potentially found based on the argument types (operators follow a different rule; §18.2.1, §18.2.5). For example:

```
namespace N {
    struct S { int i };
    void f(S);
    void g(S);
    void h(int);
}
```

```
struct Base {
    void f(N::S);
};

struct D : Base {
    void mf();

    void g(N::S x)
    {
        f(x);       // call Base::f()
        mf(x);      // call D::mf()
        h(1);       // error: no h(int) available
    }
};
```

In the standard, the rules for argument-dependent lookup are phrased in terms of *associated namespaces* (§iso.3.4.2). Basically:

- If an argument is a class member, the associated namespaces are the class itself (including its base classes) and the class's enclosing namespaces.
- If an argument is a member of a namespace, the associated namespaces are the enclosing namespaces.
- If an argument is a built-in type, there are no associated namespaces.

Argument-dependent lookup can save a lot of tedious and distracting typing, but occasionally it can give surprising results. For example, the search for a declaration of a function **f()** does not have a preference for functions in a **namespace** in which **f()** is called (the way it does for functions in a **class** in which **f()** is called):

```
namespace N {
    template<class T>
        void f(T, int);    // N::f()
    class X { };
}

namespace N2 {
    N::X x;

    void f(N::X, unsigned);

    void g()
    {
        f(x,1);      // calls N::f(X,int)
    }
}
```

It may seem obvious to choose **N2::f()**, but that is not done. Overload resolution is applied and the best match is found: **N::f()** is the best match for **f(x,1)** because **1** is an **int** rather than an **unsigned**. Conversely, examples have been seen where a function in the caller's namespace is chosen but the programmer expected a better function from a known namespace to be used (e.g., a standard-library function from **std**). This can be most confusing. See also §26.3.6.

14.2.5 Namespaces Are Open

A namespace is open; that is, you can add names to it from several separate 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()
}
```

That way, the members of a namespace need not be placed contiguously in a single file. This can be important when converting older programs to use namespaces. For example, consider a header file written without the use of namespaces:

```
// my header:

    void mf();      // my function
    void yf();      // your function
    int mg();       // my function
    // ...
```

Here, we have (unwisely) just added the declarations needed without concerns of modularity. This can be rewritten without reordering the declarations:

```
// my header:

    namespace Mine {
        void mf();      // my function
        // ...
    }

    void yf();          // your function (not yet put into a namespace)

    namespace Mine {
        int mg();       // my function
        // ...
    }
```

When writing new code, I prefer to use many smaller namespaces (see §14.4) 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.

Another reason to define the members of a namespace in several separate namespace declarations is that sometimes we want to distinguish parts of a namespace used as an interface from parts used to support easy implementation; §14.3 provides an example.

A namespace alias (§14.4.2) cannot be used to re-open a namespace.

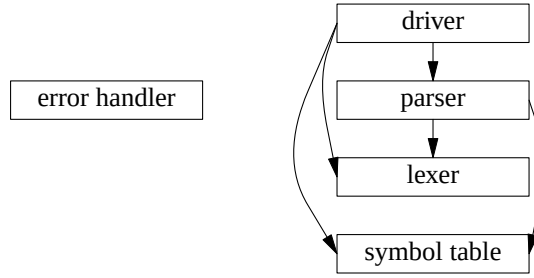
14.3 Modularization and Interfaces

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 §10.2. It can be viewed as composed of five parts:

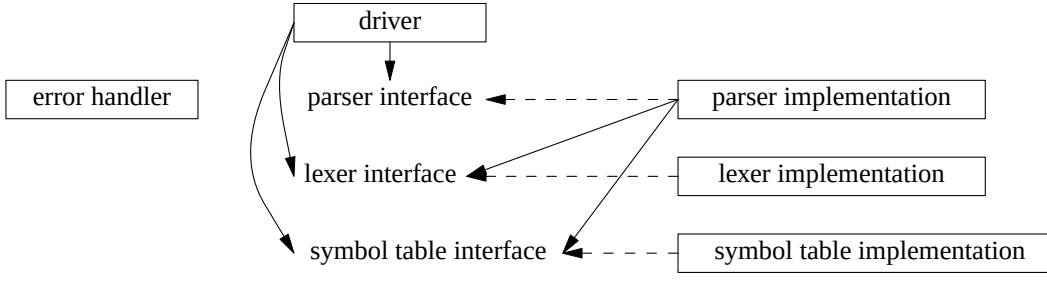
[1] The parser, doing syntax analysis: **expr()**, **term()**, and **prim()**
[2] The lexer, composing tokens out of characters: **Kind**, **Token**, **Token_stream**, and **ts**
[3] The symbol table, holding (string,value) pairs: **table**
[4] The driver: **main()** and **calculate()**
[5] The error handler: **error()** and **number_of_errors**

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:



A dashed line 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 subsections show how the logical structure of the desk calculator program can be made clear, and §15.3 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, §15.1) to the extent done here. Making the structure of the calculator explicit is simply an illustration of techniques useful for larger programs without drowning in code. In real programs, each ''module'' represented by a separate namespace will often have hundreds of functions, classes, templates, etc.

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 (§2.4.3.1, Chapter 13).

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 tasks (§5.3, Chapter 41) or 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.

14.3.1 Namespaces as Modules

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. So we can use namespaces to express the logical structure of our calculator. For example, the declarations of the parser from the desk calculator (§10.2.1) 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 §10.2.1.

The input part of the desk calculator could also be placed in its own namespace:

```
namespace Lexer {
        enum class Kind : char { /* ... */ };
        class Token { /* ... */ };
        class Token_stream { /* ... */ };

        Token_stream ts;
}
```

The symbol table is extremely simple:

```
namespace Table {
    map<string,double> table;
}
```

The driver cannot be completely put into a namespace because the language rules require `main()` to be a global function:

```
namespace Driver {
    void calculate() { /* ... */ }
}

int main() { /* ... */ }
```

The error handler is also trivial:

```
namespace Error {
    int no_of_errors;
    double error(const string& s) { /* ... */ }
}
```

This use of namespaces makes explicit what the lexer and the parser provide to a user. 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 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, 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 from 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.

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 compiler to recognize it as special (§2.2.1, §15.4).

14.3.2 Implementations

What will the code look like once it has been modularized? That depends on how we decide to access code in other namespaces. We can always access names from ''our own'' namespace exactly as we did before we introduced namespaces. However, for names in other namespaces, we have to choose among explicit qualification, **using**-declarations, and **using**-directives.

Parser::prim() provides a good test case for the use of namespaces in an implementation because it uses each of the other namespaces (except **Driver**). If we use explicit qualification, we get:

```
double Parser::prim(bool get)        // handle primaries
{
        if (get) Lexer::ts.get();

        switch (Lexer::ts.current().kind) {
        case Lexer::Kind::number:            // floating-point constant
        {       double v = Lexer::ts.current().number_value;
                Lexer::ts.get();
                return v;
        }
        case Lexer::Kind::name:
        {       double& v = Table::table[Lexer::ts.current().string_value];
                if (Lexer::ts.get().kind == Lexer::Kind::assign) v = expr(true);  // '=' seen: assignment
                return v;
        }
        case Lexer::Kind::minus:        // unary minus
                return –prim(true);
        case Lexer::Kind::lp:
        {       double e = expr(true);
                if (Lexer::ts.current().kind != Lexer::Kind::rp) return Error::error(" ')' expected");
                Lexer::ts.get();        // eat ')'
                return e;
        }
        default:
                return Error::error("primary expected");
        }
}
```

I count 14 occurrences of **Lexer::**, and (despite theories to the contrary) I don't think the more explicit use of modularity has improved readability. I didn't use **Parser::** because that would be redundant within namespace **Parser**.

If we use **using**-declarations, we get:

```
using Lexer::ts;        // saves eight occurrences of "Lexer::"
using Lexer::Kind;      // saves six occurrences of "Lexer::"
using Error::error;     // saves two occurrences of "Error::"
using Table::table;     // saves one occurrence of "Table::"
```

```
double prim(bool get)     // handle primaries
{
    if (get) ts.get();

    switch (ts.current().kind) {
    case Kind::number:        // floating-point constant
    {   double v = ts.current().number_value;
        ts.get();
        return v;
    }
    case Kind::name:
    {   double& v = table[ts.current().string_value];
        if (ts.get().kind == Kind::assign) v = expr(true);      // '=' seen: assignment
        return v;
    }
    case Kind::minus:         // unary minus
        return –prim(true);
    case Kind::lp:
    {   double e = expr(true);
        if (ts.current().kind != Kind::rp) return error("')' expected");
        ts.get();         // eat ')'
        return e;
    }
    default:
        return error("primary expected");
    }
}
```

My guess is that the **using**-declarations for **Lexer::** were worth it, but that the value of the others was marginal.

If we use **using**-directives, we get:

```
using namespace Lexer;      // saves fourteen occurrences of "Lexer::"
using namespace Error;      // saves two occurrences of "Error::"
using namespace Table;      // saves one occurrence of "Table::"

double prim(bool get)               // handle primaries
{
    // as before
}
```

The **using**-declarations for **Error** and **Table** don't buy much notationally, and it can be argued that they obscure the origins of the formerly qualified names.

So, the tradeoff among explicit qualification, **using**-declarations, and **using**-directives must be made on a case-by-case basis. The rules of thumb are:

[1] If some qualification is really common for several names, use a **using**-directive for that namespace.

[2] If some qualification is common for a particular name from a namespace, use a **using**-declaration for that name.

[3] If a qualification for a name is uncommon, use explicit qualification to make it clear from where the name comes.
[4] Don't use explicit qualification for names in the same namespace as the user.

14.3.3 Interfaces and Implementations

It should be clear that the namespace definition we used for **Parser** is not the ideal interface for **Parser** to present to its users. Instead, that **Parser** declares 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:

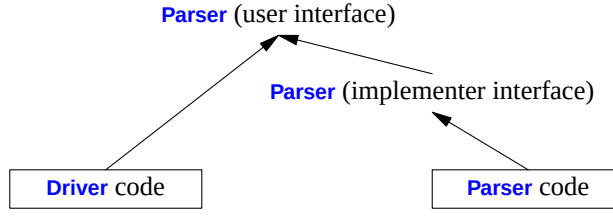
```
namespace Parser {// user interface
    double expr(bool);
}
```

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 the user interface.

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 {              // implementer interface
    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;      // use all facilities offered by lexer
    using Error::error;
    using Table::table;
}
```

or graphically:

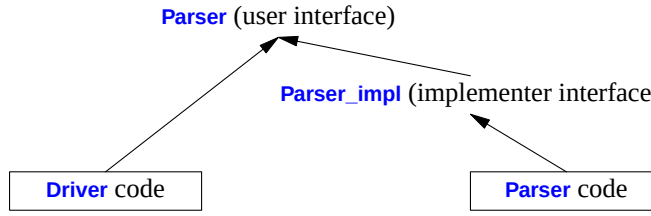


The arrows represent "relies on the interface provided by" relations.

We could give the user's interface and the implementer's interface different names, but (because namespaces are open; §14.2.5) we don't have to. The lack of separate names need not lead to confusion because the physical layout of the program (see §15.3.2) naturally provides separate (file) names. Had we decided to use a separate implementation namespace, the design would not have looked different to users:

```
namespace Parser {// user interface
        double expr(bool);
}

namespace Parser_impl {                 // implementer interface
        using namespace Parser;

        double prim(bool);
        double term(bool);
        double expr(bool);

        using namespace Lexer; // use all facilities offered by Lexer
        using Error::error;
        using Table::table;
}
```

or graphically:



For larger programs, I lean toward introducing **_impl** interfaces.

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, **Driver** using **Parser**) be insulated from such changes.

14.4 Composition Using Namespaces

In larger programs, we tend to use many namespaces. This section examines technical aspects of composing code out of namespaces.

14.4.1 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's k or the 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 overloading of the name is 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**-directives, it is a significant advantage that clashes of unused names are not considered errors.

14.4.2 Namespace Aliases

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.

14.4.3 Namespace Composition

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**-directives (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(char c)          // error: no fill() declared in My_lib
{
        // ...
}

void His_string::fill(char c)      // OK: fill() declared in His_string
{
        // ...
}

void My_lib::my_fct(String& v)// OK: String is My_lib::String, meaning His_string::String
{
        // ...
}
```

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.

Together with the **#include** mechanism (§15.2.2), the composition techniques presented here and in the following subsections provide strong support for this.

14.4.4 Composition and Selection

Combining composition (by **using**-directives) with selection (by **using**-declarations) 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 §14.4.1). 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** is providing a **List**.

Usually, I prefer to leave a name unchanged when including it into a new namespace. Then, 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
```

```
        using Her_string = Her_lib::String;                    // rename
        template<class T>
            using His_vec = His_lib::Vector<T>;        // rename

        template<class T>
            class List { /* ... */ };          // additional stuff
        // ...
}
```

There is no general language mechanism for renaming, but for types and templates, we can introduce aliases with **using** (§3.4.5, §6.5).

14.4.5 Namespaces and Overloading

Function overloading (§12.3) 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**-directives in the header files. However, it is usually best to avoid **using**-directives in header files, because putting them there greatly increases the chances of name clashes.

This overloading rule also provides a mechanism for extending libraries. For example, people often wonder why they have to explicitly mention a sequence to manipulate a container using a standard-library algorithm. For example:

```
sort(v.begin(),v.end());
```

Why not write:

```
sort(v);
```

The reason is the need for generality (§32.2), but manipulating a container is by far the most common case. We can accommodate that case like this:

```
#include<algorithm>

namespace Estd {
    using namespace std;
    template<class C>
        void sort(C& c) { std::sort(c.begin(),c.end()); }
    template<class C, class P>
        void sort(C& c, P p) { std::sort(c.begin(),c.end(),p); }
}
```

Estd (my ''extended **std**'') provides the frequently wanted container versions of **sort()**. Those are of course implemented using **std::sort()** from **<algorithm>**. We can use it like this:

```
using namespace Estd;

template<class T>
void print(const vector<T>& v)
{
    for (auto& x : v)
        cout << v << ' ';
    cout << '\n';
}
```

```
void f()
{
        std::vector<int> v {7, 3, 9, 4, 0, 1};

        sort(v);
        print(v);
        sort(v,[](int x, int y) { return x>y; });
        print(v);
        sort(v.begin(),v.end());
        print(v);
        sort(v.begin(),v.end(),[](int x, int y) { return x>y; });
        print(v);
}
```

The namespace lookup rules and the overloading rules for templates ensure that we find and invoke the correct variants of `sort()` and get the expected output:

```
0 1 3 4 7 9
9 7 4 3 1 0
0 1 3 4 7 9
9 7 4 3 1 0
```

If we removed the `using namespace std;` from `Estd`, this example would still work because `std`'s `sort()`s would be found by argument-dependent lookup (§14.2.4). However, we would then not find the standard `sort()`s for our own containers defined outside `std`.

14.4.6 Versioning

The toughest test for many kinds of interfaces is to cope with a sequence of new releases (versions). Consider a widely used interface, say, an ISO C++ standard header. After some time, a new version is defined, say, the C++11 version of the C++98 header. Functions may have been added, classes renamed, proprietary extensions (that should never have been there) removed, types changed, templates modified. To make life ''interesting'' for the implementer, hundreds of millions of lines of code are ''out there'' using the old header, and the implementer of the new version cannot ever see or modify them. Needless to say, breaking such code will cause howls of outrage, as will the absence of a new and better version. The namespace facilities described so far can be used to handle this problem with very minor exceptions, but when large amounts of code are involved, ''very minor'' still means a lot of code. Consequently, there is a way of selecting between two versions that simply and obviously guarantees that a user sees exactly one particular version. This is called an *inline namespace*:

```
namespace Popular {

        inline namespace V3_2 { // V3_2 provides the default meaning of Popular
                double f(double);
                int f(int);
                template<class T>
                        class C { /* ... */ };
        }
```

```
        namespace V3_0 {
            // ...
        }
        namespace V2_4_2 {
            double f(double);
            template<class T>
                class C { /* ... */ };
        }
}
```

Here, **Popular** contains three subnamespaces, each defining a version. The **inline** specifies that **V3_2** is the default meaning of **Popular**. So we can write:

```
using namespace Popular;

void f()
{
    f(1);           // Popular::V3_2::f(int)
    V3_0::f(1);         // Popular::V3_0::f(double)
    V2_4_2::f(1);   // Popular::V2_4_2::f(double)
}

template<class T>
Popular::C<T*> { /* ... */ };
```

This **inline namespace** solution is intrusive; that is, to change which version (subnamespace) is the default requires modification of the header source code. Also, naively using this way of handling versioning would involve a lot of replication (of common code in the different versions). However, that replication can be minimized using **#include** tricks. For example:

```
// file V3_common:
    // ... lots of declarations ...

// file V3_2:

    namespace V3_2 {  // V3_2 provides the default meaning of Popular
        double f(double);
        int f(int);
        template<class T>
            class C { /* ... */ };
        #include "V3_common"
    }

// file V3_0.h:

    namespace V3_0 {
        #include "V3_common"
    }
```

```
// file Popular.h:

    namespace Popular {
        inline
        #include "V3_2.h"
        #include "V3_0.h"
        #include "V2_4_2.h"
    }
```

I do not recommend such intricate use of header files unless it is really necessary. The example above repeatedly violates the rules against including into a nonlocal scope and against having a syntactic construct span file boundaries (the use of **inline**); see §15.2.2. Sadly, I have seen worse.

In most cases, we can achieve versioning by less intrusive means. The only example I can think of that is completely impossible to do by other means is the specialization of a template explicitly using the namespace name (e.g., **Popular::C<T*>**). However, in many important cases "in most cases" isn't good enough. Also, a solution based on a combination of other techniques is less obviously completely right.

14.4.7 Nested 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
}
```

For examples of nested namespaces in the standard library, see **chrono** (§35.2) and **rel_ops** (§35.5.3).

14.4.8 Unnamed Namespaces

It is sometimes 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 outside the unnamed namespace. 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.

14.4.9 C Headers

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 (§10.2) is an example of this.

One way to provide the standard C I/O facilities in a namespace would be to place the declarations from the C header **stdio.h** in a namespace **std**:

```
// cstdio:

    namespace std {
        int printf(const char* ... );
        // ...
    }
```

Given this **<cstdio>**, we could provide backward compatibility by adding a **using**-directive:

```
// stdio.h:

    #include<cstdio>
    using namespace std;
```

This **<stdio.h>** makes the **Hello, world!** program compile. Unfortunately, the **using**-directive makes every name from namespace **std** accessible in the global namespace. For example:

```
    #include<vector>    // carefully avoids polluting the global namespace
    vector v1;          // error: no "vector" in global scope
    #include<stdio.h>   // contains a "using namespace std;"
    vector v2;          // oops: this now works
```

So the standard requires that **<stdio.h>** place only names from **<cstdio>** in the global scope. This can be done by providing a **using**-declaration for each declaration in **<cstdio>**:

```
// stdio.h:

    #include<cstdio>
    using std::printf;
    // ...
```

Another advantage is that the **using**-declaration for **printf()** prevents a user from (accidentally or deliberately) defining a nonstandard **printf()** in the global scope. I consider nonlocal **using**-directives primarily a transition tool. I also use them for essential foundation libraries, such as the ISO C++ standard library (**std**). Most code referring to names from other namespaces can be expressed more clearly with explicit qualification and **using**-declarations.

The relationship between namespaces and linkage is described in §15.2.5.

14.5 Advice

[1] Use namespaces to express logical structure; §14.3.1.
[2] Place every nonlocal name, except **main()**, in some namespace; §14.3.1.
[3] Design a namespace so that you can conveniently use it without accidentally gaining access to unrelated namespaces; §14.3.3.
[4] Avoid very short names for namespaces; §14.4.2.
[5] If necessary, use namespace aliases to abbreviate long namespace names; §14.4.2.
[6] Avoid placing heavy notational burdens on users of your namespaces; §14.2.2, §14.2.3.
[7] Use separate namespaces for interfaces and implementations; §14.3.3.
[8] Use the **Namespace::member** notation when defining namespace members; §14.4.
[9] Use **inline** namespaces to support versioning; §14.4.6.
[10] Use **using**-directives for transition, for foundational libraries (such as **std**), or within a local scope; §14.4.9.
[11] Don't put a **using**-directive in a header file; §14.2.3.

This page intentionally left blank

15

Source Files and Programs

Form must follow function.
– Le Corbusier

- Separate Compilation
- Linkage
  File-Local Names; Header Files; The One-Definition Rule; Standard-Library Headers; Linkage to Non-C++ Code; Linkage and Pointers to Functions
- Using Header Files
  Single-Header Organization; Multiple-Header Organization; Include Guards
- Programs
  Initialization of Nonlocal Variables; Initialization and Concurrency; Program Termination
- Advice

15.1 Separate Compilation

Any realistic program consists of many logically separate components (e.g., namespaces; Chapter 14). To better manage these components, we can represent the program as a set of (source code) files where each file contains one or more logical components. Our task is to devise a physical structure (set of files) for the program that represents the logical components in a consistent, comprehensible, and flexible manner. In particular, we aim for a clean separation of interfaces (e.g., function declarations) and implementations (e.g., function definitions). 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 enforce that logical structure. Where the unit of compilation is a file, all of the 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 (§12.6) is done and `#include` directives bring in headers (§2.4.1, §15.2.2). 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 has 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 running 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 structures of a program need not be identical. For example, it can be helpful to use several source files to store the functions from a single namespace, to store a collection of namespace definitions in a single file, or to scatter the definition of a namespace over several files (§14.3.3).

Here, we will first consider some technicalities relating to linking and then discuss two ways of breaking the desk calculator (§10.2, §14.3.1) into files.

15.2 Linkage

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.cpp:
    int x = 1;
    int f() { /* do something */ }
```

```
// file2.cpp:
    extern int x;
    int f();
    void g() { x = f(); }
```

The **x** and **f()** used by **g()** in **file2.cpp** are the ones defined in **file1.cpp**. The keyword **extern** indicates that the declaration of **x** in **file2.cpp** is (just) a declaration and not a definition (§6.3). 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.cpp:
    int x = 1;
    int b = 1;
    extern int c;

// file2.cpp:
    int x;                  // means "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. Many, however, are detectable by the linker. For example, all implementations I know of correctly diagnose the double definition of **x**. However, the inconsistent declarations of **b** are uncaught on popular implementations, and the missing definition of **c** is typically only caught if **c** is used.

Note that a variable defined without an initializer in the global or a namespace scope is initialized by default (§6.3.5.1). This is *not* the case for non-**static** local variables or objects created on the free store (§11.2).

Outside a class body, an entity must be declared before it is used (§6.3.4). For example:

```
// file1.cpp:
    int g() { return f()+7; }     // error: f() not (yet) declared
    int f() { return x; }         // error: x not (yet) declared
    int x;
```

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*. For example:

```
static int x1 = 1;          // internal linkage: not accessible from other translation units
const char x2 = 'a';        // internal linkage: not accessible from other translation units
```

When used in namespace scope (including the global scope; §14.2.1), the keyword **static** (somewhat illogically) means ''not accessible from other source files'' (i.e., internal linkage). If you wanted **x1** to be accessible from other source files (''have external linkage''), you should remove the **static**. The keyword **const** implies default internal linkage, so if you wanted **x2** to have external linkage, you need to precede its definitions with **extern**:

```
int x1 = 1;                     // external linkage: accessible from other translation units
extern const char x2 = 'a';     // external linkage: accessible from other translation units
```

Names that a linker does not see, such as the names of local variables, are said to have *no linkage*.

An **inline** function (§12.1.3, §16.2.8) must be defined identically in every translation unit in which it is used (§15.2.3). Consequently, the following example isn't just bad taste; it is illegal:

```
// file1.cpp:
    inline int f(int i) { return i; }

// file2.cpp:
    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.cpp:
    extern inline int g(int i);
    int h(int i) { return g(i); }  // error: g() undefined in this translation unit

// file2.cpp:
    extern inline int g(int i) { return i+1; }
    // ...
```

We keep **inline** function definitions consistent by using header files(§15.2.2). For example:

```
// h.h:
    inline int next(int i) { return i+1; }

// file1.cpp:
    #include "h.h"
    int h(int i) { return next(i); }      // fine

// file2.cpp:
    #include "h.h"
    // ...
```

By default, **const** objects (§7.5), **constexpr** objects (§10.4), type aliases (§6.5), and anything declared **static** (§6.3.4) in a namespace scope have internal linkage. Consequently, this example is legal (although potentially confusing):

```
// file1.cpp:
    using T = int;
    const int x = 7;
    constexpr T c2 = x+1;

// file2.cpp:
    using T = double;
    const int x = 8;
    constexpr T c2 = x+9;
```

To ensure consistency, place aliases, **const**s, **constexpr**s, and **inline**s in header files (§15.2.2).

A **const** can be given external linkage by an explicit declaration:

```
// file1.cpp:
    extern const int a = 77;

// file2.cpp:
    extern const int a;

    void g()
    {
        cout << a << '\n';
    }
```

Here, **g()** will print **77**.

The techniques for managing template definitions are described in §23.7.

15.2.1 File-Local Names

Global variables are in general best avoided because they cause maintenance problems. In particular, it is hard to know where in a program they are used, and they can be a source of data races in multi-threaded programs (§41.2.4), leading to very obscure bugs.

Placing variables in a namespace helps a bit, but such variables are still subject to data races.

If you must use global variables, at least restrict their use to a single source file. This restriction can be achieved in one of two ways:

[1] Place declarations in an unnamed namespace.

[2] Declare an entity **static**.

An unnamed namespace (§14.4.8) 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.cpp:
    namespace {
        class X { /* ... */ };
        void f();
        int i;
        // ...
    }

// file2.cpp:
    class X { /* ... */ };
    void f();
    int i;
    // ...
```

The function **f()** in **file1.cpp** is not the same function as the **f()** in **file2.cpp**. 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.

The keyword **static** (confusingly) means ‘‘use internal linkage’’ (§44.2.3). That’s an unfortunate leftover from the earliest days of C.

15.2.2 Header Files

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. Consider:

```
#include "to_be_included"
```

The `#include`-directive replaces the line in which the `#include` appears with the contents of the file `to_be_included`. The content of `to_be_included` 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 seems extravagant to recompile a source file each time it is included somewhere, but the text can be a reasonably dense encoding for program interface information, and the compiler need only analyze details actually used (e.g., template bodies are often not completely analyzed until instantiation time; §26.3). Furthermore, most modern C++ implementations provide some form of (implicit or explicit) 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 { /* ... */ }`
`inline` namespaces	`inline 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) { /* ... */ }`
`constexpr` function definitions	`constexpr int fac(int n) { return (n<2) ? 1 : fac(n–1); }`
Data declarations	`extern int a;`
`const` definitions	`const float pi = 3.141593;`
`constexpr` definitions	`constexpr float pi2 = pi*pi;`
Enumerations	`enum class Light { red, yellow, green };`
Name declarations	`class Matrix;`
Type aliases	`using value_type = long;`

Compile-time assertions	`static_assert(4<=sizeof(int),"small ints");`
Include directives	`#include<algorithm>`
Macro definitions	`#define VERSION 12.03`
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 { /* ... */ }`
using-directives	`using namespace Foo;`

Including a header containing such definitions will lead to errors or (in the case of the **using**-directive) to confusion. Header files are conventionally suffixed by **.h**, and files containing function or data definitions are suffixed by **.cpp**. They are therefore often referred to as ''.h files'' and ''.cpp files,'' respectively. Other conventions, such as **.c**, **.C**, **.cxx**, **.cc**, **.hh**, and **hpp** 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 recommendations are:

- **#include** only as headers (don't **#include** ''ordinary source code containing variable definitions and non-**inline** functions'').
- **#include** only complete declarations and definitions.
- **#include** only in the global scope, in linkage specification blocks, and in namespace definitions when converting old code (§15.2.4).
- Place all **#include**s before other code to minimize unintended dependencies.
- Avoid macro magic.
- Minimize the use of names (especially aliases) not local to a header in a header.

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.

15.2.3 The One-Definition Rule

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!), and a definition of a class may be textually included in two source files by **#include** directives (§15.2.2).

Worse, a ''file'' isn't a concept that is part of the C++ language definition; 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:

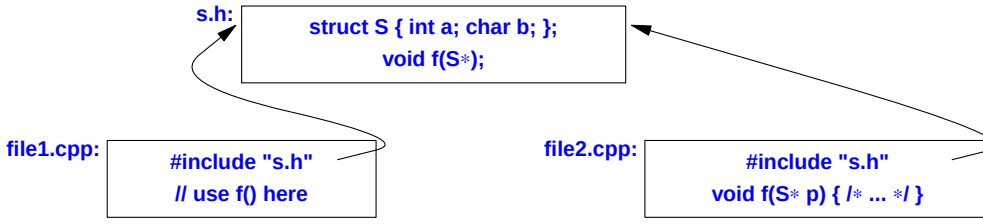
```
// file1.cpp:
    struct S { int a; char b; };
    void f(S*);

// file2.cpp:
    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.cpp` will naturally assume that the definition of `S` in `file2.cpp` 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:

```
// s.h:
    struct S { int a; char b; };
    void f(S*);

// file1.cpp:
    #include "s.h"
    // use f() here

// file2.cpp:
    #include "s.h"
    void f(S* p) { /* ... */ }
```

or graphically:



Here are examples of the three ways of violating the ODR:

```
// file1.cpp:
    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.cpp:
    struct S2 { int a; char b; };

// file2.cpp:
    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.cpp:
    typedef int X;
    struct S3 { X a; char b; };

// file2.cpp:
    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 **#include**ing them doesn't protect against this last form of ODR violation. Local type aliases and macros can change the meaning of **#include**d declarations:

```
// s.h:
    struct S { Point a; char b; };

// file1.cpp:
    #define Point int
    #include "s.h"
    // ...

// file2.cpp:
    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. This applies even to function template definitions and to class templates containing member function definitions.

15.2.4 Standard-Library Headers

The facilities of the standard library are presented through a set of standard headers (§4.1.2, §30.2). 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>** is usually stored as a text file called **map.h** in some 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 (§40.3) built in and treat **#include<cmath>** as a switch that makes the standard math functions available without actually 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++ compilers only (§15.2.5)
namespace std {             // the standard library is defined in namespace std (§4.1.2)
extern "C" {                // stdio functions have C linkage (§15.2.5)
#endif
    /* ... */
    int printf(const char*, ...);
    /* ... */
#ifdef __cplusplus
}
}
// ...
using std::printf;      // make printf 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. The macro **__cplusplus** is defined by the C++ compiler (§12.6.2) and can be used to distinguish C++ code from code intended for a C compiler.

15.2.5 Linkage to Non-C++ Code

Typically, a C++ program contains parts written in other languages (e.g., C or Fortran). Similarly, it is common for C++ code fragments to be used as parts of programs written mainly in some other language (e.g., Python or Matlab). 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 (system-specific) 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 (§12.6.1) 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 (§12.6.2) 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 (§6.3.4, §6.4.2) 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
extern "C" { int g4; }      // 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() (§43.3).

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.

15.2.6 Linkage and Pointers to Functions

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 declarations marked *error* as a language extension. However, even for compatible C and C++ implementations, **std::function** (§33.5.3) or lambdas with any form of capture (§11.4.3) cannot cross the language barrier.

15.3 Using Header Files

To illustrate the use of headers, I present a few alternative ways of expressing the physical structure of the calculator program (§10.2, §14.3.1).

15.3.1 Single-Header Organization

The simplest solution to the problem of partitioning a program into several files is to put the definitions in a suitable number of **.cpp** files and to declare the types, functions, classes, etc., needed for them to cooperate in a single **.h** file that each **.cpp** file **#include**s. That's the initial organization I would use for a simple program for my own use; if something more elaborate turned out to be needed, I would reorganize later.

For the calculator program, we might use five **.cpp** files – **lexer.cpp**, **parser.cpp**, **table.cpp**, **error.cpp**, and **main.cpp** – to hold function and data definitions. The header **dc.h** holds the declarations of every name used in more than one **.cpp** file:

```
// dc.h:

#include <map>
#include<string>
#include<iostream>

namespace Parser {
        double expr(bool);
        double term(bool);
        double prim(bool);
}

namespace Lexer {
        enum class Kind : char {
                name, number, end,
                plus='+', minus='−', mul='*', div='/', print=';', assign='=', lp='(', rp=')'
        };

        struct Token {
                Kind kind;
                string string_value;
                double number_value;
        };

        class Token_stream {
        public:
                Token(istream& s) : ip{&s}, owns{false}, ct{Kind::end} { }
                Token(istream* p) : ip{p}, owns{true}, ct{Kind::end} { }

                ~Token() { close(); }

                Token get();            // read and return next token
                Token& current();       // most recently read token

                void set_input(istream& s) { close(); ip = &s; owns=false; }
                void set_input(istream* p) { close(); ip = p; owns = true; }
        private:
                void close() { if (owns) delete ip; }

                istream* ip;            // pointer to an input stream
                bool owns;                      // does the Token_stream own the istream?
                Token ct {Kind::end};   // current_token
        };

        extern Token_stream ts;
}
```

```
namespace Table {
        extern map<string,double> table;
}

namespace Error {
        extern int no_of_errors;
        double error(const string& s);
}

namespace Driver {
        void calculate();
}
```

The keyword **extern** is used for every variable declaration to ensure that multiple definitions do not occur as we **#include dc.h** in the various **.cpp** files. The corresponding definitions are found in the appropriate **.cpp** files.

I added standard-library headers as needed for the declarations in **dc.h**, but I did not add declarations (such as **using**-declarations) needed only for the convenience of an individual **.cpp** file.

Leaving out the actual code, **lexer.cpp** will look something like this:

```
// lexer.cpp:

#include "dc.h"
#include <cctype>
#include <iostream>       // redundant: in dc.h

Lexer::Token_stream ts;

Lexer::Token Lexer::Token_stream::get() { /* ... */ }
Lexer::Token& Lexer::Token_stream::current() { /* ... */ }
```

I used explicit qualification, **Lexer::**, for the definitions rather that simply enclosing them all in

```
namespace Lexer { /* ... */ }
```

That avoids the possibility of accidentally adding new members to **Lexer**. On the other hand, had I wanted to add members to **Lexer** that were not part of its interface, I would have had to reopen the namespace (§14.2.5).

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.cpp** the compiler will be presented with:

```
namespace Lexer {  // from dc.h
        // ...
        class Token_stream {
        public:
                Token get();
                // ...
        };
}
```

```
// ...

Lexer::Token Lexer::Token_stream::get() { /* ... */ }
```

This ensures that the compiler will detect any inconsistencies in the types specified for a name. For example, had **get()** been declared to return a **Token**, but defined to return an **int**, the compilation of **lexer.cpp** 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 **.cpp** files will fail to compile.

File **parser.cpp** will look like this:

```
// parser.cpp:

#include "dc.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

File **table.cpp** will look like this:

```
// table.cpp:

#include "dc.h"

std::map<std::string,double> Table::table;
```

The symbol table is a standard-library **map**.

File **error.cpp** becomes:

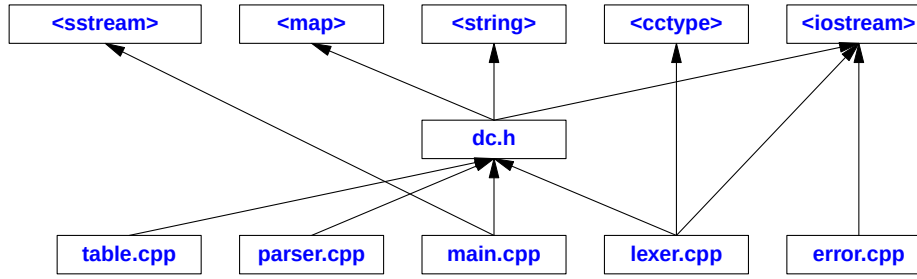
```
// error.cpp:

#include "dg.h"
// any more #includes or declarations

int Error::no_of_errors;
double Error::error(const string& s) { /* ... */ }
```

Finally, file **main.cpp** will look like this:

```
// main.cpp:

#include "dc.h"
#include <sstream>
#include <iostream>      // redundant: in dc.h

void Driver::calculate() { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

To be recognized as *the* **main()** of the program, **main()** must be a global function (§2.2.1, §15.4), so no namespace is used here.

The physical structure of the system can be presented like this:



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. Similarly, for a small program, separating out **error.cpp** and **table.cpp** from **main.cpp** would often be excessive.

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.

15.3.2 Multiple-Header Organization

An alternative physical organization lets each logical module have its own header defining the facilities it provides. Each **.cpp** file then has a corresponding **.h** file specifying what it provides (its interface). Each **.cpp** 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 **.cpp** 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 **expr()**, **prim()**, and **term()**, implementing the parser is presented by **parser_impl.h**:

```
// parser_impl.h:

#include "parser.h"
#include "error.h"
#include "lexer.h"

using Error::error;
using namespace Lexer;

namespace Parser {              // interface for implementers
        double prim(bool get);
        double term(bool get);
        double expr(bool get);
}
```

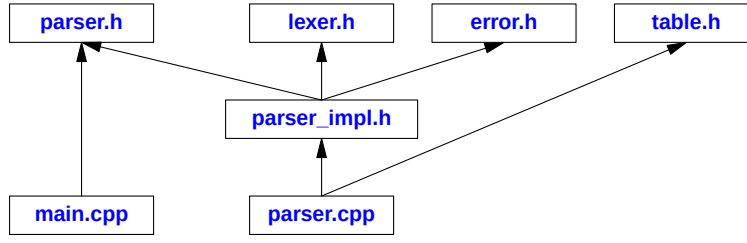
The distinction between the user interface and the interface for implementers would be even clearer had we used a **Parser_impl** namespace (§14.3.3).

The user's interface in header **parser.h** is **#include**d to give the compiler a chance to check consistency (§15.3.1).

The functions implementing the parser are stored in **parser.cpp** together with **#include** directives for the headers that the **Parser** functions need:

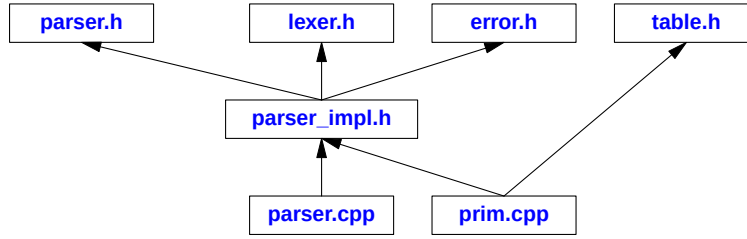
```
// parser.cpp:

#include "parser_impl.h"
#include "table.h"

using Table::table;

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 §14.3.1. To simplify this structure, we could have **#include**d **table.h** in **parser_impl.h** rather than in **parser.cpp**. 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, **prim()**,

so if we were really keen on minimizing dependencies we could place prim() in its own .cpp 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 1000 as the result of a simple dependency analysis leading to a better use of headers.

15.3.2.1 Other Calculator Modules

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 the implementation of a logical module consists of many functions that need a shared context (in addition to what is provided to users).

The error handler provides its interface in error.h:

```
// error.h:

#include<string>

namespace Error {
        int Error::number_of_errors;
        double Error::error(const std::string&);
}
```

The implementation is found in **error.cpp**:

```
// error.cpp:

#include "error.h"

int Error::number_of_errors;
double Error::error(const std::string&) { /* ... */ }
```

The lexer provides a rather large and messy interface:

```
// lexer.h:

#include<string>
#include<iostream>

namespace Lexer {

        enum class Kind : char {/* ... */ };

        class Token { /* ... */ };
        class Token_stream { /* ... */ };

        extern Token_stream is;
}
```

In addition to **lexer.h**, the implementation of the lexer depends on **error.h** and on the character-classification functions in **<cctype>** (§36.2):

```
// lexer.cpp:

#include "lexer.h"
#include "error.h"
#include <iostream>        // redundant: in lexer.h
#include <cctype>

Lexer::Token_stream is;  // defaults to "read from cin"

Lexer::Token Lexer::Token_stream::get() { /* ... */ };
Lexer::Token& Lexer::Token_stream::current() { /* ... */ };
```

We could have factored out the **#include** directive 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>

namespace Table {
        extern std::map<std::string,double> table;
}
```

Because we assume that every header may be #included in several .cpp files, we must separate the declaration of table from its definition:

```
// table.cpp:

#include "table.h"

std::map<std::string,double> Table::table;
```

I just stuck the driver into main.cpp:

```
// main.cpp:

#include "parser.h"
#include "lexer.h"    // to be able to set ts
#include "error.h"
#include "table.h"    // to be able to predefine names
#include <sstream> // to be able to put main()'s arguments into a string stream

namespace Driver {
        void calculate() { /* ... */ }
}

int main(int argc, char* argv[]) { /* ... */ }
```

For a larger system, it is usually worthwhile to separate out the driver and minimize what is done in main(). That way main() calls 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 for the driver to be called from a variety of functions.

15.3.2.2 Use of Headers

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/IDE does not make it convenient to look at several files simultaneously, then using many headers becomes less attractive.

A word of caution: a few 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-size 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 begin to be counted in the thousands. At that scale, the basic techniques discussed here still apply, but their management becomes a Herculean task. Tools, such as dependency analysers, can be of great help, but there is little they can do for compiler and linker performance if the program is an unstructured mess. 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. 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.

15.3.3 Include 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 (§15.2.3).

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 for 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.

15.4 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 (§6.3, §15.2.3). A program must contain exactly one function called `main()` (§2.2.1). The main computation performed by the program starts with the invocation of the global function `main()` and ends with a return from `main()`. The return type of `main()` is `int`, and the following two versions of `main()` are supported by all implementations:

```
int main() { /* ... */ }
int main(int argc, char* argv[]) { /* ... */ }
```

A program can only provide one of those two alternatives. In addition, an implementation can allow other versions of `main()`. The `argc`, `argv` version is used to transmit arguments from the program's environment; see §10.2.7.

The `int` returned by `main()` is passed to whatever system invoked `main()` as the result of the program. A nonzero return value from `main()` indicates an error.

This simple story must be elaborated on for programs that contain global variables (§15.4.1) or that throw an uncaught exception (§13.5.2.5).

15.4.1 Initialization of Nonlocal Variables

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 definition order. If such a variable has no explicit initializer, it is by default initialized to the default for its type (§17.3.3). 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 (§13.5.2.5). 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 (§7.7). For example:

```
void f()
{
    cout << ++use_count();   // read and increment
    // ...
}
```

Like other uses of **static**, this technique is not thread-safe. The initialization of a local **static** is thread-safe (§42.3.3). In this case, the initialization is even with a constant expression (§10.4), so that it is done at link time and not subject to data races (§42.3.3). However, the **++** can lead to a data race.

The initialization of nonlocal (statically allocated) 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 (§10.4) 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.

15.4.2 Initialization and Concurrency

Consider:

```
int x = 3;
int y = sqrt(++x);
```

What could be the values of x and y? The obvious answer is ''3 and 2!'' Why? The initialization of a statically allocated object with a constant expression is done at link time, so x becomes 3. However, y's initializer is not a constant expression (sqrt() is no constexpr), so y is not initialized until run time. However, the order of initialization of statically allocated objects in a single translation unit is well defined: they are initialized in definition order (§15.4.1). So, y becomes 2.

The flaw in this argument is that if multiple threads are used (§5.3.1, §42.2), each will do the run-time initialization. No mutual exclusion is implicitly provided to prevent a data race. Then, sqrt(++x) in one thread may happen before or after the other thread manages to increment x. So, the value of y may be sqrt(4) or sqrt(5).

To avoid such problems, we should (as usual):

- Minimize the use of statically allocated objects and keep their initialization as simple as possible.
- Avoid dependencies on dynamically initialized objects in other translation units (§15.4.1).

In addition, to avoid data races in initialization, try these techniques in order:

[1] Initialize using constant expressions (note that built-in types without initializers are initialized to zero and that standard containers and strings are initialized to empty by link-time initialization).
[2] Initialize using expressions without side effects.
[3] Initialize in a known single-threaded ''startup phase'' of computation.
[4] Use some form of mutual exclusion (§5.3.4, §42.3).

15.4.3 Program Termination

A program can terminate in several ways:

[1] By returning from main()
[2] By calling exit()
[3] By calling abort()
[4] By throwing an uncaught exception
[5] By violating noexcept
[6] By calling quick_exit()

In addition, there are a variety of ill-behaved and implementation-dependent ways of making a program crash (e.g., dividing a double by zero).

If a program is terminated using the standard-library function exit(), the destructors for constructed static objects are called (§15.4.1, §16.2.12). 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()** (§2.2.1), **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 (§13.5.1). 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. For example, **main()** may catch every exception (§13.5.2.2).

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 (§15.4.1, §16.2.12). An argument to **atexit()** cannot take arguments or return a result, and there is an implementation-defined limit to the number of atexit functions. A nonzero value returned by **atexit()** indicates that the limit is reached. These limitations make **atexit()** less useful than it appears at first glance. Basically, **atexit()** is a C workaround for the lack of destructors.

The destructor of a constructed statically allocated object (§6.4.2) created before a call of **atexit(f)** will be invoked after **f** is invoked. The destructor of such an object created after a call of **atexit(f)** will be invoked before **f** is invoked.

The **quick_exit()** function is like **exit()** except that it does not invoke any destructors. You register functions to be invoked by **quick_exit()** using **at_quick_exit()**.

The **exit()**, **abort()**, **quick_exit()**, **atexit()**, and **at_quick_exit()** functions are declared in **<cstdlib>**.

15.5 Advice

[1] Use header files to represent interfaces and to emphasize logical structure; §15.1, §15.3.2.
[2] **#include** a header in the source file that implements its functions; §15.3.1.
[3] Don't define global entities with the same name and similar-but-different meanings in different translation units; §15.2.
[4] Avoid non-inline function definitions in headers; §15.2.2.
[5] Use **#include** only at global scope and in namespaces; §15.2.2.
[6] **#include** only complete declarations; §15.2.2.
[7] Use include guards; §15.3.3.

[8] `#include` C headers in namespaces to avoid global names; §14.4.9, §15.2.4.
[9] Make headers self-contained; §15.2.3.
[10] Distinguish between users' interfaces and implementers' interfaces; §15.3.2.
[11] Distinguish between average users' interfaces and expert users' interfaces; §15.3.2.
[12] Avoid nonlocal objects that require run-time initialization in code intended for use as part of non-C++ programs; §15.4.1.

This page intentionally left blank

Part III

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

‘‘... 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...’’

— Niccolò Machiavelli (‘‘The Prince’’ §vi)

16

Classes

Those types are not "abstract";
they are as real as **int** *and* **float**.
– Doug McIlroy

- Introduction
- Class Basics
  Member Functions; Default Copying; Access Control; **class** and **struct**; Constructors; **explicit** Constructors; In-Class Initializers; In-Class Function Definitions; Mutability; Self-Reference; Member Access; **static** Members; Member Types
- Concrete Classes
  Member Functions; Helper Functions; Overloaded Operators; The Significance of Concrete Classes
- Advice

16.1 Introduction

C++ classes are a tool for creating new types that can be used as conveniently as the built-in types. In addition, derived classes (§3.2.4, Chapter 20) and templates (§3.4, Chapter 23) allow the programmer to express (hierachical and parametric) relationships among classes and to take advantage of such relationships.

A type is a concrete representation of a concept (an idea, a notion, etc.). 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 video game, 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, easier to reason about, and easier to modify than a program that does not. A well-chosen set of user-defined types

also 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 be found only through exhaustive testing.

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 its 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:

§16.2 Class Basics introduces the basic facilities for defining a class and its members.

§16.3 Concrete Classes discusses the design of elegant and efficient concrete classes.

The following chapters go into greater detail and presents abstract classes and class hierarchies:

Chapter 17 *Construction, Cleanup, Copy, and Move* presents the variety of ways to control initialization of objects of a class, how to copy and move objects, and how to provide ''cleanup actions'' to be performed when an object is destroyed (e.g., goes out of scope).

Chapter 18 *Operator Overloading* explains how to define unary and binary operators (such as `+`, `*`, and `!`) for user-defined types and how to use them.

Chapter 19 *Special Operators* considers how to define and use operators (such as `[]`, `()`, `->`, `new`) that are ''special'' in that they are commonly used in ways that differ from arithmetic and logical operators. In particular, this chapter shows how to define a string class.

Chapter 20 *Derived Classes* introduces the basic language features supporting object-oriented programming. Base and derived classes, virtual functions, and access control are covered.

Chapter 21 *Class Hierarchies* focuses on the use of base and derived classes to effectively organize code around the notion of class hierarchies. Most of this chapter is devoted to discussion of programming techniques, but technical aspects of multiple inheritance (classes with more than one base class) are also covered.

Chapter 22 *Run-Time Type Information* describes the techniques for explicitly navigating class hierarchies. In particular, the type conversion operations `dynamic_cast` and `static_cast` are presented, as is the operation for determining the type of an object given one of its base classes (`typeid`).

16.2 Class Basics

Here is a very brief summary of classes:

- A class is a user-defined type.
- A class consists of a set of members. The most common kinds of members are data members and member functions.
- Member functions can define the meaning of initialization (creation), copy, move, and cleanup (destruction).

- Members are accessed using `.` (dot) for objects and `->` (arrow) for pointers.
- Operators, such as `+`, `!`, and `[]`, can be defined for a class.
- A class is a namespace containing its members.
- The `public` members provide the class's interface and the `private` members provide implementation details.
- A `struct` is a `class` where members are by default `public`.

For example:

```
class X {
private:                    // the representation (implementation) is private
        int m;
public:                     // the user interface is public
        X(int i =0) :m{i} { }  // a constructor (initialize the data member m)

        int mf(int i)       // a member function
        {
                int old = m;
                m = i;      // set a new value
                return old; // return the old value
        }
};

X var {7}; // a variable of type X, initialized to 7

int user(X var, X* ptr)
{
        int x = var.mf(7);      // access using . (dot)
        int y = ptr->mf(9);     // access using -> (arrow)
        int z = var.m;          // error: cannot access private member
}
```

The following sections expand on this and give rationale. The style is tutorial: a gradual development of ideas, with details postponed until later.

16.2.1 Member Functions

Consider implementing the concept of a date using a `struct` (§2.3.1, §8.2) 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, `Date`, 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; §16.2.4) 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 (§8.2). 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 ''knows'' for which object it was invoked. But see §16.2.12 for the notion of a **static** member.

16.2.2 Default Copying

By default, objects can be copied. In particular, a class object can be initialized with a copy of an object of its class. For example:

```
Date d1 = my_birthday;   // initialization by copy
Date d2 {my_birthday};   // 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 (§3.3, §17.5).

Similarly, class objects can by default be copied by assignment. For example:

```
void f(Date& d)
{
    d = my_birthday;
}
```

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 (§3.3, §17.5).

16.2.3 Access Control

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 (§16.2.4); member functions can be defined and used exactly as before. For example:

```
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
}
```

The `init()` function is now essential because making the data private forces us to provide a way of initializing members. For example:

```
Date dx;
dx.m = 3;               // error: m is private
dx.init(25,3,2011);     // OK
```

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, 2016) 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 definitions of the member functions in order to learn to use a class. A more subtle, but most significant, advantage is that focusing on the design of a good interface simply leads to better code because thoughts and time otherwise devoted to debugging are expended on concerns related to proper use.

The protection of private data relies on restriction of the use of the class member names. It can therefore be circumvented by address manipulation (§7.4.1) and explicit type conversion (§11.5). But this, of course, is cheating. C++ protects against accident rather than deliberate circumvention (fraud). Only hardware can offer perfect protection against malicious use of a general-purpose language, and even that is hard to do in realistic systems.

16.2.4 **class** and **struct**

The construct

```
class X { ... };
```

is called a *class definition*; it defines a type called **X**. 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 (§15.2.3).

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: /* ... */ };
```

These two definitions of **S** are interchangeable, though it is usually wise to stick to one style. Which style you use depends on circumstances and taste. I tend to use **struct** for classes that I think of as ‘‘just simple data structures.’’ If I think of a class as ‘‘a proper type with an invariant,’’ I use **class**. Constructors and access functions can be quite useful even for **struct**s, but as a shorthand rather than guarantors of invariants (§2.4.3.2, §13.4).

By default, members of a **class** are private:

```
class Date1 {
        int d, m, y;            // private by default
public:
        Date1(int dd, int mm, int yy);
        void add_year(int n);   // add n years
};
```

However, we can also use the access specifier **private:** to say that the members following are private, just as **public:** says that the members following are public:

```
struct Date2 {
private:
        int d, m, y;
public:
        Date2(int dd, int mm, int yy);
        void add_year(int n);       // add n years
};
```

Except for the different name, **Date1** and **Date2** are equivalent.

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, though, and might affect the object layout (§20.5). So does having more than one private section. However, allowing many access specifiers in a class is useful for machine-generated code.

16.2.5 Constructors

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 {
        int d, m, y;
public:
        Date(int dd, int mm, int yy);          // constructor
        // ...
};
```

When a class has a constructor, all objects of that class will be initialized by a constructor call. 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: third argument missing
```

Since a constructor defines initialization for a class, we can use the **{}**-initializer notation:

```
Date today = Date {23,6,1983};
Date xmas {25,12,1990};         // abbreviated form
Date release1_0 {10,12};        // error: third argument missing
```

I recommend the **{}** notation over the **()** notation for initialization because it is explicit about what is being done (initialization), avoids some potential mistakes, and can be used consistently (§2.2.2, §6.3.5). There are cases where **()** notation must be used (§4.4.1, §17.3.2.1), but they are rare.

By providing several constructors, we can provide a variety of ways of initializing objects of a type. 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 ordinary functions (§12.3). As long as the constructors differ sufficiently in their argument types, the compiler can select the correct one for a use:

```
Date today {4};                 // 4, today.m, today.y
Date july4 {"July 4, 1983"};
Date guy {5,11};                // 5, November, today.y
Date now;                       // default initialized as today
Date start {};                  // 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 (§12.2.5). For **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**).

Alternatively, we could use the default values directly as default arguments:

```
class Date {
        int d, m, y;
public:
        Date(int dd =today.d, int mm =today.m, int yy =today.y);
        // ...
};

Date::Date(int dd, int mm, int yy)
{
        // check that the Date is valid
}
```

However, I chose to use **0** to avoid building actual values into **Date**'s interface. That way, we have the option to later improve the implementation of the default.

Note that by guaranteeing proper initialization of objects, the constructors greatly simplify the implementation of member functions. Given constructors, other member functions no longer have to deal with the possibility of uninitialized data (§16.3.1).

16.2.6 explicit Constructors

By default, a constructor invoked by a single argument acts as an implicit conversion from its argument type to its type. For example:

```
complex<double> d {1};        // d=={1,0} (§5.6.2)
```

Such implicit conversions can be extremely useful. Complex numbers are an example: if we leave out the imaginary part, we get a complex number on the real axis. That's exactly what mathematics requires. However, in many cases, such conversions can be a significant source of confusion and errors. Consider **Date**:

```
void my_fct(Date d);

void f()
{
    Date d {15};    // plausible: x becomes {15,today.m,today.y}
    // ...
    my_fct(15);     // obscure
    d = 15;         // obscure
    // ...
}
```

At best, this is obscure. There is no clear logical connection between the number **15** and a **Date** independently of the intricacies of our code.

Fortunately, we can specify that a constructor is not used as an *implicit* conversion. A constructor declared with the keyword **explicit** can only be used for initialization and explicit conversions. For example:

```
class Date {
    int d, m, y;
public:
    explicit Date(int dd =0, int mm =0, int yy =0);
    // ...
};

Date d1 {15};           // OK: considered explicit
Date d2 = Date{15};     // OK: explicit
Date d3 = {15};         // error: = initialization does not do implicit conversions
Date d4 = 15;           // error: = initialization does not do implicit conversions

void f()
{
    my_fct(15);         // error: argument passing does not do implicit conversions
    my_fct({15});       // error: argument passing does not do implicit conversions
    my_fct(Date{15});   // OK: explicit
    // ...
}
```

An initialization with an **=** is considered a *copy initialization*. In principle, a copy of the initializer is placed into the initialized object. However, such a copy may be optimized away (elided), and a move operation (§3.3.2, §17.5.2) may be used if the initializer is an rvalue (§6.4.1). Leaving out the **=** makes the initialization explicit. Explicit initialization is known as *direct initialization*.

By default, declare a constructor that can be called with a single argument **explicit**. You need a good reason not to do so (as for **complex**). If you define an implicit constructor, it is best to document your reason or a maintainer may suspect that you were forgetful (or ignorant).

If a constructor is declared **explicit** and defined outside the class, that **explicit** cannot be repeated:

```
class Date {
        int d, m, y;
public:
        explicit Date(int dd);
        // ...
};

Date::Date(int dd) { /* ... */ }                 // OK
explicit Date::Date(int dd) { /* ... */ }        // error
```

Most examples where **explicit** is important involve a single constructor argument. However, **explicit** can also be useful for constructors with zero or more than one argument. For example:

```
struct X {
        explicit X();
        explicit X(int,int);
};

X x1 = {};              // error: implicit
X x2 = {1,2};           // error: implicit

X x3 {};                // OK: explicit
X x4 {1,2};             // OK: explicit

int f(X);

int i1 = f({});         // error: implicit
int i2 = f({1,2});      // error: implicit

int i3 = f(X{});        // OK: explicit
int i4 = f(X{1,2});     // OK: explicit
```

The distinction between direct and copy initialization is maintained for list initialization (§17.3.4.3).

16.2.7 In-Class Initializers

When we use several constructors, member initialization can become repetitive. 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
        // ...
};
```

We can deal with that by introducing default arguments to reduce the number of constructors (§16.2.5). Alternatively, we can add initializers to data members:

```
class Date {
        int d {today.d};
        int m {today.m};
        int y {today.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
        // ...
```

Now, each constructor has the **d**, **m**, and **y** initialized unless it does it itself. For example:

```
Date::Date(int dd)
        :d{dd}
{
        // check that the Date is valid
}
```

This is equivalent to:

```
Date::Date(int dd)
        :d{dd}, m{today.m}, y{today.y}
{
        // check that the Date is valid
}
```

16.2.8 In-Class Function Definitions

A member function defined within the class definition – rather than simply declared there – is taken to be an inline (§12.1.5) member function. That is, in-class definition of member functions is for small, rarely modified, 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, the member function's meaning must be the same wherever it is **#include**d (§15.2.3).

A member can refer to another member of its class independently of where that member is defined (§6.3.4). Consider:

```
class Date {
public:
        void add_month(int n) { m+=n; }     // increment the Date's m
        // ...
private:
        int d, m, y;
};
```

That is, function and data member declarations are order independent. I could equivalently have written:

```
class Date {
public:
        void add_month(int n) { m+=n; }      // increment the Date's m
        // ...
private:
        int d, m, y;
};

inline void Date::add_month(int n)  // add n months
{
        m+=n;      // increment the Date's m
}
```

This latter style is often used to keep class definitions simple and easy to read. It also provides a textual separation of a class's interface and implementation.

Obviously, I simplified the definition of `Date::add_month`; just adding `n` and hoping to hit a good date is too naive (§16.3.1).

16.2.9 Mutability

We can define a named object as a constant or as a variable. In other words, a name can refer to an object that holds an *immutable* or a *mutable* value. Since the precise terminology can be a bit clumsy, we end up referring to some variables as being constant or briefer still to `const` variables. However odd that may sound to a native English speaker, the concept is useful and deeply embedded in the C++ type system. Systematic use of immutable objects leads to more comprehensible code, to more errors being found early, and sometimes to improved performance. In particular, immutability is a most useful property in a multi-threaded program (§5.3, Chapter 41).

To be useful beyond the definition of simple constants of built-in types, we must be able to define functions that operate on `const` objects of user-defined types. For freestanding functions that means functions that take `const T&` arguments. For classes it means that we must be able to define member functions that work on `const` objects.

16.2.9.1 Constant Member Functions

The `Date` as defined so far provides member functions for giving a `Date` a value. 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;

        void add_year(int n);       // add n years
        // ...
};
```

The **const** after the (empty) argument list in the function declarations 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:

```
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:

```
int Date::year()    // error: const missing in member function type
{
    return y;
}
```

In other words, **const** is part of the type of **Date::day()**, **Date::month()**, 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 a const Date
}
```

16.2.9.2 Physical and Logical Constness

Occasionally, a member function is logically **const**, but it still needs to change the value of a member. That is, to a user, the function appears not to change the state of its object, but 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. 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 {
public:
    // ...
    string string_rep() const;      // string representation
private:
    bool cache_valid;
    string cache;
    void compute_cache_value(); // fill cache
    // ...
};
```

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** and **cache_valid** members must change occasionally for the design to make sense.

Such problems could be solved through brute force using a cast, for example, a **const_cast** (§11.5.2). However, there are also reasonably elegant solutions that do not involve messing with type rules.

16.2.9.3 mutable

We can define a member of a class to be **mutable**, meaning that it can be modified even in a **const** object:

```
class Date {
public:
        // ...
        string string_rep() const;                  // string representation
private:
        mutable bool cache_valid;
        mutable string cache;
        void compute_cache_value() const;           // fill (mutable) cache
        // ...
};
```

Now we can define **string_rep()** in the obvious way:

```
string Date::string_rep() const
{
        if (!cache_valid) {
                compute_cache_value();
                cache_valid = true;
        }
        return cache;
}
```

We can now use **string_rep()** for both **const** and non-**const** objects. For example:

```
void f(Date d, const Date cd)
{
        string s1 = d.string_rep();
        string s2 = cd.string_rep();            // OK!
        // ...
}
```

16.2.9.4 Mutability through Indirection

Declaring a member **mutable** is most appropriate when only a small part of a representation of a small object is allowed to change. More complicated cases are often better handled by placing the changing data in a separate object and accessing it indirectly. If that technique is used, the string-with-cache example becomes:

```
struct cache {
        bool valid;
        string rep;
};

class Date {
public:
        // ...
        string string_rep() const;                  // string representation
private:
        cache* c;                                   // initialize in constructor
        void compute_cache_value() const;           // fill what cache refers to
        // ...
};

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.

Note that **const** does not apply (transitively) to objects accessed through pointers or references. The human reader may consider such an object as ''a kind of subobject,'' but the compiler does not know such pointers or references to be any different from any others. That is, a member pointer does not have any special semantics that distinguish it from other pointers.

16.2.10 Self-Reference

The state update functions **add_year()**, **add_month()**, and **add_day()** (§16.2.3) 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 (non-**static**) member function knows for which object it was invoked and can explicitly 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.

In a non-**static** 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***. However, **this** is considered an rvalue, so it is not possible to take the address of **this** or to assign to **this**. In a **const** member function of class **X**, the type of **this** is **const X*** to prevent modification of the object itself (see also §7.5).

Most uses of **this** are implicit. In particular, every reference to a non-**static** 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. For example:

```
struct Link {
        Link* pre;
        Link* suc;
        int data;

        Link* insert(int x)    // insert x before this
        {
                return pre = new Link{pre,this,x};
        }
```

```
        void remove()  // remove and destroy this
        {
                if (pre) pre->suc = suc;
                if (suc) suc->pre = pre;
                delete this;
        }

        // ...
};
```

Explicit use of **this** is required for access to members of base classes from a derived class that is a template (§26.3.7).

16.2.11 Member Access

A member of a class **X** can be accessed by applying the **.** (dot) operator to an object of class **X** or by applying the **->** (arrow) operator to a pointer to an object of class **X**. For example:

```
struct X {
        void f();
        int m;
};

void user(X x, X* px)
{
        m = 1;          // error: there is no m in scope
        x.m = 1;        // OK
        x->m = 1;       // error: x is not a pointer
        px->m = 1;      // OK
        px.m = 1;       // error: px is a pointer
}
```

Obviously, there is a bit of redundancy here: the compiler knows whether a name refers to an **X** or to an **X***, so a single operator would have been sufficient. However, a programmer might be confused, so from the first days of C the rule has been to use separate operators.

From inside a class no operator is needed. For example:

```
void X::f()
{
        m = 1;          // OK: "this->m = 1;" (§16.2.10)
}
```

That is, an unqualified member name acts as if it had been prefixed by **this->**. Note that a member function can refer to the name of a member before it has been declared:

```
struct X {
        int f() { return m; }   // fine: return this X's m
        int m;
};
```

If we want to refer to a member in general, rather than to a member of a particular object, we qualify by the class name followed by **::**. For example:

```
struct S {
        int m;
        int f();
        static int sm;
};

int X::f() { return m; }            // X's f
int X::sm {7};                      // X's static member sm (§16.2.12)
int (S::*) pmf() {&S::f};           // X's member f
```

That last construct (a pointer to member) is fairly rare and esoteric; see §20.6. I mention it here just to emphasize the generality of the rule for **::**.

16.2.12 [static] Members

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 (§6.4.2). 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 dd, int mm, int yy); // set default_date to Date(dd,mm,yy)
};
```

We can now define the **Date** constructor to use **default_date** 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 ...
}
```

Using **set_default()**, 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);    // call Date's static member set_default()
}
```

If used, a **static** member – a function or data member – must be defined somewhere. The keyword **static** is not repeated in the definition of a **static** member. For example:

```
Date Date::default_date {16,12,1770};          // definition of Date::default_date

void Date::set_default(int d, int m, int y)    // definition of Date::set_default
{
    default_date = {d,m,y};                    // assign new value to default_date
}
```

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{};

void f(Date);

void g()
{
    f(Date{});
}
```

Consequently, we don't need a separate function for reading the default date. Furthermore, where the target type is unambiguously a **Date**, plain **{}** is sufficient. For example:

```
void f1(Date);

void f2(Date);
void f2(int);

void g()
{
    f1({});          // OK: equivalent to f1(Date{})
    f2({}):          // error: ambiguous: f2(int) or f2(Date)?
    f2(Date{});      // OK
```

In multi-threaded code, **static** data members require some kind of locking or access discipline to avoid race conditions (§5.3.4, §41.2.4). Since multi-threading is now very common, it is unfortunate that use of **static** data members was quite popular in older code. Older code tends to use **static** members in ways that imply race conditions.

16.2.13 Member Types

Types and type aliases can be members of a class. For example:

```
template<typename T>
class Tree {
        using value_type = T;                   // member alias
        enum Policy { rb, splay, treeps };      // member enum
        class Node {                            // member class
                Node* right;
                Node* left;
                value_type value;
        public:
                void f(Tree*);
        };
        Node* top;
public:
        void g(const T&);
        // ...
};
```

A *member class* (often called a *nested class*) can refer to types and **static** members of its enclosing class. It can only refer to non-**static** members when it is given an object of the enclosing class to refer to. To avoid getting into the intricacies of binary trees, I use purely technical "**f()** and **g()**"-style examples.

A nested class has access to members of its enclosing class, even to **private** members (just as a member function has), but has no notion of a current object of the enclosing class. For example:

```
template<typename T>
void Tree::Node::f(Tree* p)
{
        top = right;                    // error: no object of type Tree specified
        p->top = right;                 // OK
        value_type v = left->value;     // OK: value_type is not associated with an object
}
```

A class does not have any special access rights to the members of its nested class. For example:

```
template<typename T>
void Tree::g(Tree::Node* p)
{
        value_type val = right->value;          // error: no object of type Tree::Node
        value_type v = p->right->value;         // error: Node::right is private
        p->f(this);                             // OK
}
```

Member classes are more a notational convenience than a feature of fundamental importance. On the other hand, member aliases are important as the basis of generic programming techniques relying on associated types (§28.2.4, §33.1.3). Member **enum**s are often an alternative to **enum class**es when it comes to avoiding polluting an enclosing scope with the names of enumerators (§8.4.1).

16.3 Concrete Classes

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, currency values, 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++ directly supports a few of these abstractions as built-in types. However, most are not, and cannot be, directly supported by the language 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 (§20.4) and classes in class hierarchies (§20.3, §21.2).

A class is called *concrete* (or *a concrete class*) if its representation is part of its definition. This distinguishes it from abstract classes (§3.2.2, §20.4) which provide an interface to a variety of implementations. Having the representation available allows us:

- To place objects on the stack, in statically allocated memory, and in other objects
- To copy and move objects (§3.3, §17.5)
- To refer directly to named objects (as opposed to accessing through pointers and references)

This makes concrete classes simple to reason about and easy for the compiler to generate optimal code for. Thus, we prefer concrete classes for small, frequently used, and performance-critical types, such as complex numbers (§5.6.2), smart pointers (§5.2.1), and containers (§4.4).

It was an early explicit aim of C++ to support the definition and efficient use of such user-defined 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:

```
namespace Chrono {

    enum class Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Date {
    public:          // public interface:
        class Bad_date { };  // exception class

        explicit Date(int dd ={}, Month mm ={}, int yy ={});          // {} means "pick a default"
```

```
    // nonmodifying 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[], in max) const;      // C-style string representation

    // (modifying) 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:
        bool is_valid();                            // check if this Date represents a date
        int d, m, y;                                // representation
    };

    bool is_date(int d, Month m, int y);            // true for valid date
    bool is_leapyear(int y);                        // true if y is a leap year

    bool operator==(const Date& a, const Date& b);
    bool operator!=(const Date& a, const Date& b);

    const Date& default_date();                     // the default date

    ostream& operator<<(ostream& os, const Date& d);    // print d to os
    istream& operator>>(istream& is, Date& d);          // read Date from is into d
} // Chrono
```

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 (§16.2.5).

[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 modify **Date**s without actually having to know the details of the representation or fiddle with the intricacies of the semantics.

[4] Implicitly defined operations that allow **Date**s to be freely copied (§16.2.2).

[5] A class, **Bad_date**, to be used for reporting errors as exceptions.

[6] A set of useful helper functions. The helper functions are not members and have no direct access to the representation of a **Date**, but they are identified as related by the use of the namespace **Chrono**.

I defined a **Month** type to cope with the problem of remembering the month/day order, for example, to avoid confusion about whether the 7th of June is written **{6,7}** (American style) or **{7,6}** (European style).

I considered introducing separate types **Day** and **Year** to cope with possible confusion of **Date{1995,Month::jul,27}** and **Date{27,Month::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. Dealing 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.

To save the user from having to explicitly mention year and month even when they are implied by context, I added a mechanism for providing a default. Note that for **Month** the **{}** gives the (default) value **0** just as for integers even though it is not a valid **Month** (§8.4). However, in this case, that's exactly what we want: an otherwise illegal value to represent "pick the default." Providing a default (e.g., a default value for **Date** objects) is a tricky design problem. For some types, there is a conventional default (e.g., **0** for integers); for others, no default makes sense; and finally, there are some types (such as **Date**) where the question of whether to provide a default is nontrivial. In such cases, it is best – at least initially – not to provide a default value. I provide one for **Date** primarily to be able to discuss how to do so.

I omitted the cache technique from §16.2.9 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 {16,Month::dec,d.year()};

    if (d.day()==29 && d.month()==Month::feb) {
        // ...
    }

    if (midnight()) d.add_day(1);

    cout << "day after:" << d+1 << '\n';

    Date dd;  // initialized to the default date
    cin>>dd;
    if (dd==d) cout << "Hurray!\n";
}
```

This assumes that the addition operator, **+**, has been declared for **Date**s. I do that in §16.3.3.

Note the use of explicit qualification of **dec** and **feb** by **Month**. I used an **enum class** (§8.4.1) specifically to be able to use short names for the months, yet also ensure that their use would not be obscure or ambiguous.

Why is it worthwhile to define a specific type for something as simple as a date? After all, we could just define a simple data structure:

```
struct Date {
    int day, month, year;
};
```

Each programmer could then 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. 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.

To simplify, I decided to eliminate the notion of changing the default date. Doing so eliminates some opportunities for confusion and the likelihood of race conditions in a multi-threaded program (§5.3.1). I seriously considered eliminating the notion of a default date altogether. That would have forced users to consistently explicitly initialize their Dates. However, that can be inconvenient and surprising, and more importantly common interfaces used for generic code require default construction (§17.3.3). That means that I, as the designer of Date, have to pick the default date. I chose January 1, 1970, because that is the starting point for the C and C++ standard-library time routines (§35.2, §43.6). Obviously, eliminating set_default_date() caused some loss of generality of Date. However, design – including class design – is about making decisions, rather than just deciding to postpone them or to leave all options open for users.

To preserve an opportunity for future refinement, I declared default_date() as a helper function:

```
const Date& Chrono::default_date();
```

That doesn't say anything about how the default date is actually set.

16.3.1 Member Functions

Naturally, an implementation for each member function must be provided somewhere. For example:

```
Date::Date(int dd, Month mm, int yy)
    :d{dd}, m{mm}, y{yy}
{
    if (y == 0) y = default_date().year();
    if (m == Month{}) m = default_date().month();
    if (d == 0) d = default_date().day();

    if (!is_valid()) throw Bad_date();
}
```

The constructor checks that the data supplied denotes a valid Date. If not, say, for {30,Month::feb,1994}, it throws an exception (§2.4.3.1, Chapter 13), which indicates that something went wrong. 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 §2.4.3.2, §13.4).

I'm using the value Month{} – which doesn't represent a month and has the integer value 0 – 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 `Month{}`, meaning `0`, can be used because it is within the range guaranteed for the enumeration `Month` (§8.4).

I use the member initializer syntax (§17.4) to initialize the members. After that, I check for `0` and modify the values as needed. This clearly does not provide optimal performance in the (hopefully rare) case of an error, but the use of member initializers leaves the structure of the code obvious. This makes the style less error-prone and easier to maintain than alternatives. Had I aimed at optimal performance, I would have used three separate constructors rather than a single constructor with default arguments.

I considered making the validation function `is_valid()` public. However, I found the resulting user code more complicated and less robust than code relying on catching the exception:

```
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 §4.4.2
    }
}
```

However, checking that a `{d,m,y}` set of values is a valid date is not a computation that depends on the representation of a `Date`, so I implemented `is_valid()` in terms of a helper function:

```
bool Date::is_valid()
{
    return is_date(d,m,y);
}
```

Why have both `is_valid()` and `is_date()`? In this simple example, we could manage with just one, but I can imagine systems where `is_date()` (as here) checks that a `(d,m,y)`-tuple represents a valid date and where `is_valid()` does an additional check on whether that date can be reasonably represented. For example, `is_valid()` might reject dates from before the modern calendar became commonly used.

As is common for such simple concrete types, the definitions of `Date`'s 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;                    // number of whole years
        int mm = static_cast<int>(m)+n%12;     // number of months ahead
        if (12 < mm) {                         // note: dec is represented by 12
            ++delta_y;
            mm -= 12;
        }

        // ... handle the cases where the month mm doesn't have day d ...

        y += delta_y;
        m = static_cast<Month>(mm);
        return *this;
    }

    // ... handle negative n ...

    return *this;
}
```

I wouldn't call the code for **add_month()** pretty. In fact, if I added all the details, it might even approach the complexity of relatively simple real-world code. This points to a problem: adding a month is conceptually simple, so why is our code getting complicated? In this case, the reason is that the **d,m,y** representation isn't as convenient for the computer as it is for us. A better representation (for many purposes) would be simply a number of days since a defined "day zero" (e.g., January 1, 1970). That would make computation on **Date**s simple at the expense of complexity in providing output fit for humans.

Note that assignment and copy initialization are provided by default (§16.2.2). Also, **Date** doesn't need a destructor because a **Date** owns no resources and requires no cleanup when it goes out of scope (§3.2.1.2).

16.3.2 Helper Functions

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 is_leapyear(int y);
bool is_date(int d, Month m, int y);

const Date& default_date();
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**? In early C++, as in C, their declarations were simply placed in the same file as the declaration of class **Date**. Users who needed **Date**s would make them all available by including the file that defined the interface (§15.2.2). For example:

```
#include "Date.h"
```

In addition (or alternatively), we can make the association explicit by enclosing the class and its helper functions in a namespace (§14.3.1):

```
namespace Chrono {          // facilities for dealing with time

    class Date { /* ... */};

    int diff(Date a, Date b);
    bool is_leapyear(int y);
    bool is_date(int d, Month m, int y);
    const Date& default_date();
    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 overelaboration that leads to inconvenience.

Naturally, the helper function must be defined somewhere:

```
bool Chrono::is_date(int d, Month m, int y)
{
    int ndays;

    switch (m) {
    case Month::feb:
        ndays = 28+is_leapyear(y);
        break;
    case Month::apr: case Month::jun: case Month::sep: case Month::nov:
        ndays = 30;
        break;
    case Month::jan: case Month::mar: case Month::may: case Month::jul:
    case Month::aug: case Month::oct: case Month::dec:
        ndays = 31;
        break;
    default:
        return false;
    }

    return 1<=d && d<=ndays;
}
```

I'm deliberately being a bit paranoid here. A **Month** shouldn't be outside the **jan** to **dec** range, but it is possible (someone might have been sloppy with a cast), so I check.

The troublesome **default_date** finally becomes:

```
const Date& Chrono::default_date()
{
        static Date d {1,Month::jan,1970};
        return d;
}
```

16.3.3 Overloaded Operators

It is often useful to add functions to enable conventional notation. For example, **operator==()** 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) { return d.add_day(1); }          // increase Date by one day
Date& operator--(Date& d) { return d.add_day(-1); }         // decrease Date by one day

Date& operator+=(Date& d, int n) { return d.add_day(n); }   // add n days
Date& operator-=(Date& d, int n) { return d.add_day(-n); }  // subtract n days

Date operator+(Date d, int n) { return d+=n; }              // add n days
Date operator-(Date d, int n) { return d+=n; }              // subtract n days

ostream& operator<<(ostream&, Date d);                      // output d
istream& operator>>(istream&, Date& d);                     // read into d
```

These operators are defined in **Chrono** together with **Date** to avoid overload problems and to benefit from argument-dependent lookup (§14.2.4).

For **Date**, these operators can be seen as mere conveniences. However, for many types – such as complex numbers (§18.3), vectors (§4.4.1), and function-like objects (§3.4.3, §19.2.2) – 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 18.

For **Date**, I was tempted to provide **+=** and **-=** as member functions instead of **add_day()**. Had I done so, I would have followed a common idiom (§3.2.1.1).

Note that assignment and copy initialization are provided by default (§16.3, §17.3.3).

16.3.4 The Significance of Concrete Classes

I call simple user-defined types, such as **Date**, *concrete types* to distinguish them from abstract classes (§3.2.2) and class hierarchies (§20.4), and also to emphasize their similarity to built-in types such as **int** and **char**. Concrete classes are used just like built-in types. Concrete types 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 called object-oriented programming (§3.2.4, Chapter 21).

The intent of a concrete type is to do a single, relatively simple 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 run-time polymorphic behavior (see §3.2.3, §20.3.2).

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);
        // ...
};
```

Alternatively, the derived class mechanism discussed in Chapter 20 can be used to define new types from a concrete class by describing the desired differences. The definition of **Vec** from **vector** (§4.4.1.2) is an example of this. However, derivation from a concrete class should be done with care and only rarely because of the lack of virtual functions and run-time type information (§17.5.1.4, Chapter 22).

With a reasonably good compiler, a concrete class such as **Date** incurs no hidden overhead in time or space. In particular, no indirection through pointers is necessary for access to objects of concrete classes, and no ''housekeeping'' data is stored in objects of concrete classes. 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 an 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. In particular, they can be used to make interfaces more specific and less error-prone. For example:

```
Month do_something(Date d);
```

This is far less likely to be misunderstood or misused than:

```
int do_something(int d);
```

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 represented as

simple aggregates of built-in types. Alternatively, 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.

16.4 Advice

[1] Represent concepts as classes; §16.1.
[2] Separate the interface of a class from its implementation; §16.1.
[3] Use public data (**struct**s) only when it really is just data and no invariant is meaningful for the data members; §16.2.4.
[4] Define a constructor to handle initialization of objects; §16.2.5.
[5] By default declare single-argument constructors **explicit**; §16.2.6.
[6] Declare a member function that does not modify the state of its object **const**; §16.2.9.
[7] A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures; §16.3.
[8] Make a function a member only if it needs direct access to the representation of a class; §16.3.2.
[9] Use a namespace to make the association between a class and its helper functions explicit; §16.3.2.
[10] Make a member function that doesn’t modify the value of its object a **const** member function; §16.2.9.1.
[11] 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; §16.2.12.

This page intentionally left blank

17

Construction, Cleanup, Copy, and Move

Ignorance more frequently begets confidence
than does knowledge.
– Charles Darwin

- Introduction
- Constructors and Destructors
  Constructors and Invariants; Destructors and Resources; Base and Member Destructors; Calling Constructors and Destructors; **virtual** Destructors
- Class Object Initialization
  Initialization Without Constructors; Initialization Using Constructors; Default Constructors; Initializer-List Constructors
- Member and Base Initialization
  Member Initialization; Base Initializers; Delegating Constructors; In-Class Initializers; **static** Member Initialization
- Copy and Move
  Copy; Move
- Generating Default Operations
  Explicit Defaults; Default Operations; Using Default Operations; **delete**d Functions
- Advice

17.1 Introduction

This chapter focuses on technical aspects of an object's ‘‘life cycle’’: How do we create an object, how do we copy it, how do we move it around, and how do we clean up after it when it goes away? What are proper definitions of ‘‘copy’’ and ‘‘move’’? For example:

```
string ident(string arg)        // string passed by value (copied into arg)
{
    return arg;                 // return string (move the value of arg out of ident() to a caller)
}

int main ()
{
    string s1 {"Adams"};        // initialize string (construct in s1).
    s1 = indet(s1);             // copy s1 into ident()
                                // move the result of ident(s1) into s1;
                                // s1's value is "Adams".
    string s2 {"Pratchett"};    // initialize string (construct in s2)
    s1 = s2;                    // copy the value of s2 into s1
                                // both s1 and s2 have the value "Pratchett".
}
```

Clearly, after the call of **ident()**, the value of **s1** ought to be **"Adams"**. We copy the value of **s1** into the argument **arg**, then we move the value of **arg** out of the function call and (back) into **s1**. Next, we construct **s2** with the value **"Prachett"** and copy it into **s1**. Finally, at the exit from **main()** we destroy the variables **s1** and **s2**. The difference between *move* and *copy* is that after a copy two objects must have the same value, whereas after a move the source of the move is not required to have its original value. Moves can be used when the source object will not be used again. They are particularly useful for implementing the notion of moving a resource (§3.2.1.2, §5.2).

Several functions are used here:

- A constructor initializing a **string** with a string literal (used for **s1** and **s2**)
- A copy constructor copying a **string** (into the function argument **arg**)
- A move constructor moving the value of a **string** (from **arg** out of **ident()** into a temporary variable holding the result of **ident(s1)**)
- A move assignment moving the value of a **string** (from the temporary variable holding the result of **ident(s1)** into **s1**)
- A copy assignment copying a **string** (from **s2** into **s1**)
- A destructor releasing the resources owned by **s1**, **s2**, and the temporary variable holding the result of **ident(s1)**

An optimizer can eliminate some of this work. For example, in this simple example the temporary variable is typically eliminated. However, in principle, these operations are executed.

Constructors, copy and move assignment operations, and destructors directly support a view of lifetime and resource management. An object is considered an object of its type after its constructor completes, and it remains an object of its type until its destructor starts executing. The interaction between object lifetime and errors is explored further in §13.2 and §13.3. In particular, this chapter doesn't discuss the issue of half-constructed and half-destroyed objects.

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 {
        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 (§6.4.2) and to implement explicit type conversion (§11.5).

Except for the ''ordinary constructor,'' these special member functions can be generated by the compiler; see §17.6.

This chapter is full of rules and technicalities. Those are necessary for a full understanding, but most people just learn the general rules from examples.

17.2 Constructors and Destructors

We can specify how an object of a class is to be initialized by defining a constructor (§16.2.5, §17.3). To complement constructors, we can define a destructor to ensure ''cleanup'' at the point of destruction of an object (e.g., when it goes out of scope). Some of the most effective techniques for resource management in C++ rely on constructor/destructor pairs. So do other techniques relying on a pair of actions, such as do/undo, start/stop, before/after, etc. For example:

```
struct Tracer {
        string mess;
        Tracer(const string& s) :mess{s} { clog << mess; }
        ~Tracer() {clog << "~" << mess; }
};

void f(const vector<int>& v)
{
        Tracer tr {"in f()\n"};
        for (auto x : v) {
                Tracer tr {string{"v loop "}+to<string>(x)+'\n'}; // §25.2.5.1
                // ...
        }
}
```

We could try a call:

```
f({2,3,5});
```

This would print to the logging stream:

```
in_f()
v loop 2
~v loop 2
v loop 3
~v loop 3
v loop 5
~v loop 5
~in_f()
```

17.2.1 Constructors and Invariants

A member with the same name as its class is called a *constructor*. For example:

```
class Vector {
public:
    Vector(int s);
    // ...
};
```

A constructor declaration specifies an argument list (exactly as for a function) but has no return type. The name of a class cannot be used for an ordinary member function, data member, member type, etc., within the class. For example:

```
struct S {
    S();                    // fine
    void S(int);            // error: no type can be specified for a constructor
    int S;                  // error: the class name must denote a constructor
    enum S { foo, bar };    // error: the class name must denote a constructor
};
```

A constructor's job is to initialize an object of its class. Often, that initialization must establish a *class invariant*, that is, something that must hold whenever a member function is called (from outside the class). Consider:

```
class Vector {
public:
    Vector(int s);
    // ...
private:
    double* elem;  // elem points to an array of sz doubles
    int sz;        // sz is non-negative
};
```

Here (as is often the case), the invariant is stated as comments: ''`elem` points to an array of `sz` doubles'' and ''`sz` is non-negative.'' The constructor must make that true. For example:

```
Vector::Vector(int s)
{
    if (s<0) throw Bad_size{s};
    sz = s;
    elem = new double[s];
}
```

This constructor tries to establish the invariant and if it cannot, it throws an exception. If the constructor cannot establish the invariant, no object is created and the constructor must ensure that no resources are leaked (§5.2, §13.3). A resource is anything we need to acquire and eventually (explicitly or implicitly) give back (release) once we are finished with it. Examples of resources are memory (§3.2.1.2), locks (§5.3.4), file handles (§13.3), and thread handles (§5.3.1).

Why would you define an invariant?

- To focus the design effort for the class (§2.4.3.2)
- To clarify the behavior of the class (e.g., under error conditions; §13.2)
- To simplify the definition of member functions (§2.4.3.2, §16.3.1)
- To clarify the class's management of resources (§13.3)
- To simplify the documentation of the class

On average, the effort to define an invariant ends up saving work.

17.2.2 Destructors and Resources

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 (§5.2, §13.3). 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 a function is called a *destructor*. The name of a destructor is `~` followed by the class name, for example `~Vector()`. One meaning of `~` is ''complement'' (§11.1.2), and a destructor for a class complements its constructors. A destructor does not take an argument, and a class can have only one destructor. Destructors are called implicitly when an automatic variable goes out of scope, an object on the free store is deleted, etc. Only in very rare circumstances does the user need to call a destructor explicitly (§17.2.4).

Destructors typically clean up and release resources. For example:

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { };          // constructor: acquire memory
    ~Vector() { delete[] elem; }                            // destructor: release memory
    // ...
private:
    double* elem;  // elem points to an array of sz doubles
    int sz;        // sz is non-negative
};
```

For example:

```
Vector* f(int s)
{
    Vector v1(s);
    // ...
    return new Vector(s+s);
}

void g(int ss)
{
    Vector* p = f(ss);
    // ...
    delete p;
}
```

Here, the **Vector v1** is destroyed upon exit from **f()**. Also, the **Vector** created on the free store by **f()** using **new** is destroyed by the call of **delete**. In both cases, **Vector**'s destructor is invoked to free (deallocate) the memory allocated by the constructor.

What if the constructor failed to acquire enough memory? For example, **s*sizeof(double)** or **(s+s)*sizeof(double)** may be larger than the amount of available memory (measured in bytes). In that case, an exception **std::bad_alloc** (§11.2.3) is thrown by **new** and the exception-handling mechanism invokes the appropriate destructors so that all memory that has been acquired (and only that) is freed (§13.5.1).

This style of constructor/destructor-based resource management is called *Resource Acquisition Is Initialization* or simply *RAII* (§5.2, §13.3).

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 **vector** and **unordered_map**, use variants of this technique for providing storage for their elements.

A type that has no destructor declared, such as a built-in type, is considered to have a destructor that does nothing.

A programmer who declares a destructor for a class must also decide if objects of that class can be copied or moved (§17.6).

17.2.3 Base and Member Destructors

Constructors and destructors interact correctly with class hierarchies (§3.2.4, Chapter 20). A constructor builds a class object "from the bottom up":

[1] first, the constructor invokes its base class constructors,
[2] then, it invokes the member constructors, and
[3] finally, it executes its own body.

A destructor "tears down" an object in the reverse order:

[1] first, the destructor executes its own body,
[2] then, it invokes its member destructors, and
[3] finally, it invokes its base class destructors.

In particular, a **virtual** base is constructed before any base that might use it and destroyed after all such bases (§21.3.5.1). This ordering ensures that a base or a member is not used before it has been initialized or used after it has been destroyed. The programmer can defeat this simple and

essential rule, but only through deliberate circumvention involving passing pointers to uninitialized variables as arguments. Doing so violates language rules and the results are usually disastrous.

Constructors execute member and base constructors in declaration order (not the order of initializers): if two constructors used a different order, the destructor could not (without serious overhead) guarantee to destroy in the reverse order of construction. See also §17.4.

If a class is used so that a default constructor is needed, and if the class does not have other constructors, the compiler will try to generate a default constructor. For example:

```
struct S1 {
        string s;
};

S1 x;       // OK: x.s is initialized to ""
```

Similarly, memberwise initialization can be used if initializers are needed. For example:

```
struct X { X(int); };

struct S2 {
        X x;
};

S2 x1;      // error:
S2 x2 {1}; // OK: x2.x is initialized with 1
```

See also §17.3.1.

17.2.4 Calling Constructors and Destructors

A destructor is invoked implicitly upon exit from a scope or by **delete**. It is typically not only unnecessary to explicitly call a destructor; doing so would lead to nasty errors. However, there are rare (but important) cases where a destructor must be called explicitly. Consider a container that (like **std::vector**) maintains a pool of memory into which it can grow and shrink (e.g., using **push_back()** and **pop_back()**). When we add an element, the container must invoke its constructor for a specific address:

```
void C::push_back(const X& a)
{
        // ...
        new(p) X{a};    // copy construct an X with the value a in address p
        // ...
}
```

This use of a constructor is known as ''placement **new**'' (§11.2.4).

Conversely, when we remove an element, the container needs to invoke its destructor:

```
void C::pop_back()
{
        // ...
        p->~X();   // destroy the X in address p
}
```

The `p->~X()` notation invokes `X`'s destructor for `*p`. That notation should never be used for an object that is destroyed in the normal way (by its object going out of scope or being `delete`d).

For a more complete example of explicit management of objects in a memory area, see §13.6.1.

If declared for a class `X`, a destructor will be implicitly invoked whenever an `X` goes out of scope or is `delete`d. This implies that we can prevent destruction of an `X` by declaring its destructor `=delete` (§17.6.4) or `private`.

Of the two alternatives, using `private` is the more flexible. For example, we can create a class for which objects can be explicitly destroyed, but not implicitly:

```
class Nonlocal {
public:
        // ...
        void destroy() { this->~Nonlocal(); }       // explicit destruction
private:
        // ...
        ~Nonlocal();                                // don't destroy implicitly
};

void user()
{
        Nonlocal x;                 // error: cannot destroy a Nonlocal
        X* p = new Nonlocal;        // OK
        // ...
        delete p;                   // error: cannot destroy a Nonlocal
        p.destroy();                // OK
}
```

17.2.5 `virtual` Destructors

A destructor can be declared to be `virtual`, and usually should be for a class with a virtual function. For example:

```
class Shape {
public:
        // ...
        virtual void draw() = 0;
        virtual ~Shape();
};

class Circle {
public:
        // ...
        void draw();
        ~Circle();          // overrides ~Shape()
        // ...
};
```

The reason we need a `virtual` destructor is that an object usually manipulated through the interface provided by a base class is often also `delete`d through that interface:

```
void user(Shape* p)
{
    p->draw();      // invoke the appropriate draw()
    // ...
    delete p;       // invoke the appropriate destructor
};
```

Had `Shape`'s destructor not been `virtual` that `delete` would have failed to invoke the appropriate derived class destructor (e.g., `~Circle()`). That failure would cause the resources owned by the deleted object (if any) to be leaked.

17.3 Class Object Initialization

This section discusses how to initialize objects of a class with and without constructors. It also shows how to define constructors to accept arbitrarily sized homogeneous initializer lists (such as `{1,2,3}` and `{1,2,3,4,5,6}`).

17.3.1 Initialization Without Constructors

We cannot define a constructor for a built-in type, yet we can initialize it with a value of suitable type. For example:

```
int a {1};
char* p {nullptr};
```

Similarly, we can initialize objects of a class for which we have not defined a constructor using

- memberwise initialization,
- copy initialization, or
- default initialization (without an initializer or with an empty initializer list).

For example:

```
struct Work {
    string author;
    string name;
    int year;
};

Work s9 { "Beethoven",
          "Symphony No. 9 in D minor, Op. 125; Choral",
          1824
        };                          // memberwise initialization

Work currently_playing { s9 };      // copy initialization
Work none {};                       // default initialization
```

The three members of `currently_playing` are copies of those of `s9`.

The default initialization of using `{}` is defined as initialization of each member by `{}`. So, `none` is initialized to `{{},{},{}}`, which is `{"","",0}` (§17.3.3).

Where no constructor requiring arguments is declared, it is also possible to leave out the initializer completely. For example:

```
Work alpha;

void f()
{
    Work beta;
    // ...
}
```

For this, the rules are not as clean as we might like. For statically allocated objects (§6.4.2), the rules are exactly as if you had used **{}**, so the value of **alpha** is **{"","",0}**. However, for local variables and free-store objects, the default initialization is done only for members of class type, and members of built-in type are left uninitialized, so the value of **beta** is **{"","",unknown}**.

The reason for this complication is to improve performance in rare critical cases. For example:

```
struct Buf {
    int count;
    char buf[16*1024];
};
```

You can use a **Buf** as a local variable without initializing it before using it as a target for an input operation. Most local variable initializations are not performance critical, and uninitialized local variables are a major source of errors. If you want guaranteed initialization or simply dislike surprises, supply an initializer, such as **{}**. For example:

```
Buf buf0;       // statically allocated, so initialized by default

void f()
{
    Buf buf1;           // leave elements uninitialized
    Buf buf2 {};        // I really want to zero out those elements

    int* p1 = new int;  // *p1 is uninitialized
    int* p2 = new int{};// *p2 == 0
    int* p3 = new int{7};// *p3 == 7
    // ...
}
```

Naturally, memberwise initialization works only if we can access the members. For example:

```
template<class T>
class Checked_pointer { // control access to T* member
public:
    T& operator*();     // check for nullptr and return value
// ...
};

Checked_pointer<int> p {new int{7}};    // error: can't access p.p
```

If a class has a private non-**static** data member, it needs a constructor to initialize it.

17.3.2 Initialization Using Constructors

Where memberwise copy is not sufficient or desirable, a constructor can be defined to initialize an object. In particular, a constructor is often used to establish an invariant for its class and to acquire resources necessary to do that (§17.2.1).

If a constructor is declared for a class, some constructor will be used for every object. It is an error to try to create an object without a proper initializer as required by the constructors. For example:

```
struct X {
    X(int);
};

X x0;          // error: no initializer
X x1 {};       // error: empty initializer
X x2 {2};      // OK
X x3 {"two"};  // error: wrong initializer type
X x4 {1,2};    // error: wrong number of initializers
X x5 {x4};     // OK: a copy constructor is implicitly defined (§17.6)
```

Note that the default constructor (§17.3.3) disappears when you define a constructor requiring arguments; after all, `X(int)` states that an `int` is required to construct an `X`. However, the copy constructor does not disappear (§17.3.3); the assumption is that an object can be copied (once properly constructed). Where the latter might cause problems (§3.3.1), you can specifically disallow copying (§17.6.4).

I used the `{}` notation to make explicit the fact that I am initializing. I am not (just) assigning a value, calling a function, or declaring a function. The `{}` notation for initialization can be used to provide arguments to a constructor wherever an object can be constructed. For example:

```
struct Y : X {
    X m {0};                    // provide default initializer for member m
    Y(int a) :X{a}, m{a} { };   // initialize base and member (§17.4)
    Y() : X{0} { };             // initialize base and member
};

X g {1};    // initialize global variable

void f(int a)
{
    X def {};                   // error: no default value for X
    Y de2 {};                   // OK: use default constructor
    X* p {nullptr};
    X var {2};                  // initialize local variable
    p = new X{4};               // initialize object on free store
    X a[] {1,2,3};              // initialize array elements
    vector<X> v {1,2,3,4};      // initialize vector elements
}
```

For this reason, {} initialization is sometimes referred to as *universal* initialization: the notation can be used everywhere. In addition, {} initialization is *uniform*: wherever you initialize an object of type X with a value v using the {v} notation, the same value of type X (X{v}) is created.

The = and () notations for initialization (§6.3.5) are not universal. For example:

```
struct Y : X {
        X m;
        Y(int a) : X(a), m=a { };     // syntax error: can't use = for member initialization
};

X g(1);     // initialize global variable

void f(int a)
{
        X def();                      // function returning an X (surprise!?)
        X* p {nullptr};
        X var = 2;                    // initialize local variable
        p = new X=4;                  // syntax error: can't use = for new
        X a[](1,2,3);                 // error: can't use () for array initialization
        vector<X> v(1,2,3,4);         // error: can't use () for list elements
}
```

The = and () notations for initialization are not uniform either, but fortunately the examples of that are obscure. If you insist on using = or () initialization, you have to remember where they are allowed and what they mean.

The usual overload resolution rules (§12.3) apply for constructors. For example:

```
struct S {
        S(const char*);
        S(double*);
};

S s1 {"Napier"};              // S::S(const char*)
S s2 {new double{1.0}};       // S::S(double*);
S s3 {nullptr};               // ambiguous: S::S(const char*) or S::S(double*)?
```

Note that the {}-initializer notation does not allow narrowing (§2.2.2). That is another reason to prefer the {} style over () or =.

17.3.2.1 Initialization by Constructors

Using the () notation, you can request to use a constructor in an initialization. That is, you can ensure that for a class, you will get initialization by constructor and not get the memberwise initialization or initializer-list initialization (§17.3.4) that the {} notation also offers. For example:

```
struct S1 {
        int a,b;                              // no constructor
};
```

```
struct S2 {
        int a,b;
        S2(int a = 0, int b = 0) : a(aa), b(bb) {}          // constructor
};

S1 x11(1,2);    // error: no constructor
S1 x12 {1,2};   // OK: memberwise initialization

S1 x13(1);      // error: no constructor
S1 x14 {1};     // OK: x14.b becomes 0

S2 x21(1,2);    // OK: use constructor
S2 x22 {1,2};   // OK: use constructor

S2 x23(1);      // OK: use constructor and one default argument
S2 x24 {1};     // OK: use constructor and one default argument
```

The uniform use of **{}** initialization only became possible in C++11, so older C++ code uses **()** and **=** initialization. Consequently, the **()** and **=** may be more familiar to you. However, I don't know any logical reason to prefer the **()** notation except in the rare case where you need to distinguish between initialization with a list of elements and a list of constructor arguments. For example:

```
vector<int> v1 {77};        // one element with the value 77
vector<int> v2(77);         // 77 elements with the default value 0
```

This problem – and the need to choose – can occur when a type with an initializer-list constructor (§17.3.4), typically a container, also has an ''ordinary constructor'' accepting arguments of the element type. In particular, we occasionally must use **()**initialization for **vector**s of integers and floating-point numbers but never need to for **vector**s of strings of pointers:

```
vector<string> v1 {77};         // 77 elements with the default value ""
                                // (vector<string>(std::initializer_list<string>) doesn't accept {77})
vector<string> v2(77);          // 77 elements with the default value ""

vector<string> v3 {"Booh!"};    // one element with the value "Booh!"
vector<string> v4("Booh!");     // error: no constructor takes a string argument

vector<int*> v5 {100,0};        // 100 int*s initialized to nullptr (100 is not an int*)

vector<int*> v6 {0,0};          // 2 int*s initialized to nullptr
vector<int*> v7(0,0);           // empty vector (v7.size()==0)
vector<int*> v8;                // empty vector (v7.size()==0)
```

The **v6** and **v7** examples are only of interest to language lawyers and testers.

17.3.3 Default Constructors

A constructor that can be invoked without an argument is called a *default constructor*. Default constructors are very common. For example:

```
class Vector {
public:
    Vector();  // default constructor: no elements
    // ...
};
```

A default constructor is used if no arguments are specified or if an empty initializer list is provided:

```
Vector v1;      // OK
Vector v2 {};   // OK
```

A default argument (§12.2.5) can make a constructor that takes arguments into a default constructor. For example:

```
class String {
public:
    String(const char* p = "");      // default constructor: empty string
    // ...
};

String s1;      // OK
String s2 {};   // OK
```

The standard-library **vector** and **string** have such default constructors (§36.3.2, §31.3.2).

The built-in types are considered to have default and copy constructors. However, for a built-in type the default constructor is not invoked for uninitialized non-**static** variables (§17.3). The default value of a built-in type is **0** for integers, **0.0** for floating-point types, and **nullptr** for pointers. For example:

```
void f()
{
    int a0;             // uninitialized
    int a1();           // function declaration (intended?)

    int a {};           // a becomes 0
    double d {};        // d becomes 0.0
    char* p {};         // p becomes nullptr

    int* p1 = new int;  // uninitialized int
    int* p2 = new int{}; // the int is initialized to 0
}
```

Constructors for built-in types are most often used for template arguments. For example:

```
template<class T>
struct Handle {
    T* p;
    Handle(T* pp = new T{}) :p{pp} { }
    // ...
};

Handle<int> px;     // will generate int{}
```

The generated **int** will be initialized to **0**.

References and **const**s must be initialized (§7.7, §7.5). Therefore, a class containing such members cannot be default constructed unless the programmer supplies in-class member initializers (§17.4.4) or defines a default constructor that initializes them (§17.4.1). For example:

```
int glob {9};

struct X {
        const int a1 {7};      // OK
        const int a2;          // error: requires a user-defined constructor
        const int& r {9};      // OK
        int& r1 {glob};        // OK
        int& r2;               // error: requires a user-defined constructor
};

X x;        // error: no default constructor for X
```

An array, a standard-library **vector**, and similar containers can be declared to allocate a number of default-initialized elements. In such cases, a default constructor is obviously required for a class used as the element type of a **vector** or array. For example:

```
struct S1 { S1(); };                    // has default constructor
struct S2 { S2(string); };              // no default constructor

S1 a1[10];                              // OK: 10 default elements
S2 a2[10];                              // error: cannot initialize elements
S2 a3[] { "alpha", "beta" };            // OK: two elements: S2{"alpha"}, S2{"beta"}

vector<S1> v1(10);                      // OK: 10 default elements
vector<S2> v2(10);                      // error: cannot initialize elements
vector<S2> v3 { "alpha", "beta" };      // OK: two elements: S2{"alpha"}, S2{"beta"}

vector<S2> v2(10,"");                   // OK: 10 elements each initialized to S2{""}
vector<S2> v4;                          // OK: no elements
```

When should a class have a default constructor? A simple-minded technical answer is ‘‘when you use it as the element type for an array, etc.’’ However, a better question is ‘‘For what types does it make sense to have a default value?’’ or even ‘‘Does this type have a ‘special’ value we can ‘naturally’ use as a default?’’ String has the empty string, **""**, containers have the empty set, **{}**, and numeric values have zero. The trouble with deciding on a default **Date** (§16.3) arose because there is no ‘‘natural’’ default date (the Big Bang is too far in the past and not precisely associated with our everyday dates). It is a good idea not to be too clever when inventing default values. For example, the problem with containers of elements without default values is often best solved by not allocating elements until you have proper values for them (e.g., using **push_back()**).

17.3.4 Initializer-List Constructors

A constructor that takes a single argument of type **std::initializer_list** is called an *initializer-list constructor*. An initializer-list constructor is used to construct objects using a **{}**-list as its initializer

value. Standard-library containers (e.g., **vector** and **map**) have initializer-list constructors, assignments, etc. (§31.3.2, §31.4.3). Consider:

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = {
        {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};

map<vector<string>,vector<int>> years = {
        { {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000} },
        { {"Martin", "Richards"} {1982, 2003, 2007} },
        { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

The mechanism for accepting a **{}**-list is a function (often a constructor) taking an argument of type **std::initializer_list<T>**. For example:

```
void f(initializer_list<int>);

f({1,2});
f({23,345,4567,56789});
f({});      // the empty list

f{1,2};     // error: function call () missing

years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

The initializer list can be of arbitrary length but must be homogeneous. That is, all elements must be of the template argument type, **T**, or implicitly convertible to **T**.

17.3.4.1 initializer_list Constructor Disambiguation

When you have several constructors for a class, the usual overload resolution rules (§12.3) are used to select the right one for a given set of arguments. For selecting a constructor, default and initializer lists take precedence. Consider:

```
struct X {
        X(initializer_list<int>);
        X();
        X(int);
};

X x0 {};    // empty list: default constructor or initializer-list constructor? (the default constructor)
X x1 {1};   // one integer: an int argument or a list of one element? (the  initializer-list constructor)
```

The rules are:

- If either a default constructor or an initializer-list constructor could be invoked, prefer the default constructor.
- If both an initializer-list constructor and an ‘‘ordinary constructor’’ could be invoked, prefer the initializer-list constructor.

The first rule, ''prefer the default constructor,'' is basically common sense: pick the simplest constructor when you can. Furthermore, if you define an initializer-list constructor to do something with an empty list that differs from what the default constructor does, you probably have a design error on your hands.

The second rule, ''prefer the initializer-list constructor,'' is necessary to avoid different resolutions based on different numbers of elements. Consider **std::vector** (§31.4):

```
vector<int> v1 {1};        // one element
vector<int> v2 {1,2};      // two elements
vector<int> v3 {1,2,3};    // three elements

vector<string> vs1 {"one"};
vector<string> vs2 {"one", "two"};
vector<string> vs3 {"one", "two", "three"};
```

In every case, the initializer-list constructor is used. If we really want to invoke the constructor taking one or two integer arguments, we must use the **()** notation:

```
vector<int> v1(1);    // one element with the default value (0)
vector<int> v2(1,2);  // one element with the value 2
```

17.3.4.2 Use of initializer_lists

A function with an **initializer_list<T>** argument can access it as a sequence using the member functions **begin()**, **end()**, and **size()**. For example:

```
void f(initializer_list<int> args)
{
    for (int i = 0; i!=args.size(); ++i)
        cout << args.begin()[i] << "\n";
}
```

Unfortunately, **initializer_list** doesn't provide subscripting.

An **initializer_list<T>** is passed by value. That is required by the overload resolution rules (§12.3) and does not impose overhead because an **initializer_list<T>** object is just a small handle (typically two words) to an array of **T**s.

That loop could equivalently have been written:

```
void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p)
        cout << *p << "\n";
}
```

or:

```
void f(initializer_list<int> args)
{
    for (auto x : args)
        cout << x << "\n";
}
```

To explicitly use an initializer_list you must #include the header file in which it is defined: <initializer_list>. However, since vector, map, etc., use initializer_lists, their headers (<vector>, <map>, etc.) already #include <initializer_list>, so you rarely have to do so directly.

The elements of an initializer_list are immutable. Don't even think about trying to modify their values. For example:

```
int f(std::initializer_list<int> x, int val)
{
    *x.begin() = val;          // error: attempt to change the value of an initializer-list element
    return *x.begin();    // OK
}

void g()
{
    for (int i=0; i!=10; ++i)
        cout << f({1,2,3},i) << '\n';
}
```

Had the assignment in f() succeeded, it would have appeared that the value of 1 (in {1,2,3}) could change. That would have done serious damage to some of our most fundamental concepts. Because initializer_list elements are immutable, we cannot apply a move constructor (§3.3.2, §17.5.2) to them.

A container might implement an initializer-list constructor like this:

```
template<class E>
class Vector {
public:
    Vector(std::initializer_list<E> s); // initializer-list constructor
    // ...
private:
    int sz;
    E* elem;
};

template<class E>
Vector::Vector(std::initializer_list<E> s)
    :sz{s.size()}                                  // set vector size
{
    reserve(sz);                                   // get the right amount of space
    uninitialized_copy(s.begin(), s.end(), elem);      // initialize elements in elem[0:s.size())
}
```

The initializer lists are part of the universal and uniform initialization design (§17.3).

17.3.4.3 Direct and Copy Initialization

The distinction between direct initialization and copy initialization (§16.2.6) is maintained for {} initialization. For a container, this implies that the distinction is applied to both the container and its elements:

- The container's initializer-list constructor can be **explicit** or not.
- The constructor of the element type of the initializer list can be **explicit** or not.

For a **vector<vector<double>>**, we can see the direct initialization vs. copy initialization distinction applied to elements. For example:

```
vector<vector<double>> vs = {
        {10,11,12,13,14},           // OK: vector of five elements
        {10},                       // OK: vector of one element
        10,                         // error: vector<double>(int) is explicit

        vector<double>{10,11,12,13}, // OK: vector of five elements
        vector<double>{10},         // OK: vector of one element with value 10.0
        vector<double>(10),         // OK:  vector of 10 elements with value 0.0
};
```

A container can have some constructors explicit and some not. The standard-library **vector** is an example of that. For example, **std::vector<int>(int)** is **explicit**, but **std::vector<int>(initialize_list<int>)** is not:

```
vector<double> v1(7);    // OK: v1 has 7 elements; note: uses () rather than {}
vector<double> v2 = 9;   // error: no conversion from int to vector

void f(const vector<double>&);
void g()
{
        v1 = 9;          // error: no conversion from int to vector
        f(9);            // error: no conversion from int to vector
}
```

By replacing **()** with **{}** we get:

```
vector<double> v1 {7};       // OK: v1 has one element (with the value 7)
vector<double> v2 = {9};     // OK: v2 has one element (with the value 9)

void f(const vector<double>&);
void g()
{
        v1 = {9};            // OK: v1 now has one element (with the value 9)
        f({9});              // OK: f is called with the list {9}
}
```

Obviously, the results are dramatically different.

This example was carefully crafted to give an example of the most confusing cases. Note that the apparent ambiguities (in the eyes of the human reader but not the compiler) do not emerge for longer lists. For example:

```
vector<double> v1 {7,8,9};       // OK: v1 has three elements with values {7,8,9}
vector<double> v2 = {9,8,7};     // OK: v2 has three elements with values {9,8,7}
```

```
void f(const vector<double>&);
void g()
{
        v1 = {9,10,11};         // OK: v1 now has three elements with values {9,10,11}
        f({9,8,7,6,5,4});       // OK: f is called with the list {9,8,7,6,5,4}
}
```

Similarly, the potential ambiguities do not occur for lists of elements of nonintegral types:

```
vector<string> v1 { "Anya"};          // OK: v1 has one element (with the value "Anya")
vector<string> v2 = {"Courtney"};     // OK: v2 has one element (with the value "Courtney")

void f(const vector<string>&);
void g()
{
        v1 = {"Gavin"};         // OK: v1 now has one element (with the value "Gavin")
        f({"Norah"});           // OK: f is called with the list {"Norah"}
}
```

17.4 Member and Base Initialization

Constructors can establish invariants and acquire resources. Generally, they do that by initializing class members and base classes.

17.4.1 Member Initialization

Consider a class that might be used to hold information for a small organization:

```
class Club {
        string name;
        vector<string> members;
        vector<string> 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 initializer list starts with a colon, and the individual member initializers are separated by commas.

The members' constructors are called before the body of the containing class's own constructor is executed (§17.2.3). The constructors are called in the order in which the members are declared in the class rather than the order in which the members appear in the initializer list. To avoid confusion, it is best to specify the initializers in the member declaration order. Hope for a compiler warning if you don't get the order right. The member destructors are called in the reverse order of construction after the body of the class's own destructor has been executed.

If a member constructor needs no arguments, the member need not be mentioned in the member initializer list. For example:

```
Club::Club(const string& n, Date fd)
        : name{n}, founded{fd}
{
        // ...
}
```

This constructor is equivalent to the previous version. In each case, **Club::officers** and **Club::members** are initialized to a **vector** with no elements.

It is usually a good idea to be explicit about initializing members. Note that an ''implicitly initialized'' member of a built-in type is left uninitialized (§17.3.1).

A constructor can initialize members and bases of its class, but not members or bases of its members or bases. For example:

```
struct B { B(int); /* ... */};
struct BB : B { /* ... */ };
struct BBB : BB {
        BBB(int i) : B(i) { };  // error: trying to initialize base's base
        // ...
};
```

17.4.1.1 Member Initialization and Assignment

Member initializers are essential for types for which the meaning of initialization differs from that of assignment. For example:

```
class X {
        const int i;
        Club cl;
        Club& rc;
        // ...
        X(int ii, const string& n, Date d, Club& c) : i{ii}, cl{n,d}, rc{c} { }
};
```

A reference member or a **const** member must be initialized (§7.5, §7.7, §17.3.3). However, for most types 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 to make it explicit that initialization is being done. Often, there also is an efficiency advantage to using the initializer syntax (compared to using an assignment). 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.

17.4.2 Base Initializers

Bases of a derived class are initialized in the same way non-data members are. That is, if a base requires an initializer, it must be provided as a base initializer in a constructor. If we want to, we can explicitly specify default construction. For example:

```
class B1 { B1(); };    // has default constructor
class B2 { B2(int); } // no default constructor

struct D1 : B1, B2 {
        D1(int i) :B1{}, B2{i} {}
};

struct D2 : B1, B2 {
        D2(int i) :B2{i} {}          // B1{} is used implicitly
};

struct D1 : B1, B2 {
        D1(int i) { }                // error: B2 requires an int initializer
};
```

As with members, the order of initialization is the declaration order, and it is recommended to specify base initializers in that order. Bases are initialized before members and destroyed after members (§17.2.3).

17.4.3 Delegating Constructors

If you want two constructors to do the same action, you can repeat yourself or define ''an **init()** function'' to perform the common action. Both ''solutions'' are common (because older versions of C++ didn't offer anything better). For example:

```
class X {
        int a;
        validate(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
public:
        X(int x) { validate(x); }
        X() { validate(42); }
        X(string s) { int x = to<int>(s); validate(x); }     // §25.2.5.1
        // ...
};
```

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. The alternative is to define one constructor in terms of another:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42} { }
        X(string s) :X{to<int>(s)} { }          // §25.2.5.1
        // ...
};
```

That is, a member-style initializer using the class's own name (its constructor name) calls another constructor as part of the construction. Such a constructor is called a *delegating constructor* (and occasionally a *forwarding constructor*).

You cannot both delegate and explicitly initialize a member. For example:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42}, a{56} { }          // error
        // ...
};
```

Delegating by calling another constructor in a constructor's member and base initializer list is very different from explicitly calling a constructor in the body of a constructor. Consider:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() { X{42}; }     // likely error
        // ...
};
```

The `X{42}` simply creates a new unnamed object (a temporary) and does nothing with it. Such use is more often than not a bug. Hope for a compiler warning.

An object is not considered constructed until its constructor completes (§6.4.2). When using a delegating constructor, the object is not considered constructed until the delegating constructor completes – just completing the delegated-to constructor is not sufficient. A destructor will not be

called for an object unless its original constructor completed.

If all you need is to set a member to a default value (that doesn't depend on a constructor argument), a member initializer (§17.4.4) may be simpler.

17.4.4 In-Class Initializers

We can specify an initializer for a non-**static** data member in the class declaration. For example:

```
class A {
public:
        int a {7};
        int b = 77;
};
```

For pretty obscure technical reasons related to parsing and name lookup, the **{}** and **=** initializer notations can be used for in-class member initializers, but the **()** notation cannot.

By default, a constructor will use such an in-class initializer, so that example is equivalent to:

```
class A {
public:
        int a;
        int b;
        A() : a{7}, b{77} {}
};
```

Such use of in-class initializers can save a bit of typing, but the real benefits come in more complicated classes with multiple constructors. Often, several constructors use the same initializer for a member. For example:

```
class A {
public:
        A() :a{7}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
        A(int a_val) :a{a_val}, b{5}, algorithm{"MD5"}, state{"Constructor run"} {}
        A(D d) :a{7}, b{g(d)}, algorithm{"MD5"}, state{"Constructor run"} {}
        // ...
private:
        int a, b;
        HashFunction algorithm;      // cryptographic hash to be applied to all As
        string state;                // string indicating state in object life cycle
};
```

The fact that **algorithm** and **state** have the same value in all constructors is lost in the mess of code and can easily become a maintenance problem. To make the common values explicit, we can factor out the unique initializer for data members:

```
class A {
public:
        A() :a{7}, b{5} {}
        A(int a_val) :a{a_val}, b{5} {}
        A(D d) :a{7}, b{g(d)} {}
        // ...
```

```
private:
    int a, b;
    HashFunction algorithm {"MD5"};     // cryptographic hash to be applied to all As
    string state {"Constructor run"};   // string indicating state in object life cycle
};
```

If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it ''overrides'' the default). So we can simplify further:

```
class A {
public:
    A() {}
    A(int a_val) :a{a_val} {}
    A(D d) :b{g(d)} {}
    // ...
private:
    int a {7};                          // the meaning of 7 for a is ...
    int b {5};                          // the meaning of 5 for b is ...
    HashFunction algorithm {"MD5"};     // Cryptographic hash to be applied to all As
    string state {"Constructor run"};   // String indicating state in object lifecycle
};
```

As shown, default in-class initializers provide an opportunity for documentation of common cases.

An in-class member initializer can use names that are in scope at the point of their use in the member declaration. Consider the following headache-inducing technical example:

```
int count = 0;
int count2 = 0;

int f(int i) { return i+count; }

struct S {
    int m1 {count2};    // that is, ::count2
    int m2 {f(m1)};     // that is, this->m1+::count; that is, ::count2+::count
    S() { ++count2; }   // very odd constructor
};

int main()
{
    S s1;       // {0,0}
    ++count;
    S s2;       // {1,2}
}
```

Member initialization is done in declaration order (§17.2.3), so first **m1** is initialized to the value of a global variable **count2**. The value of the global variable is obtained at the point where the constructor for a new **S** object is run, so it can (and in this example does) change. Next, **m2** is initialized by a call to the global **f()**.

It is a bad idea to hide subtle dependencies on global data in member initializers.

17.4.5 static Member Initialization

A **static** class member is statically allocated rather than part of each object of the class. Generally, the **static** member declaration acts as a declaration for a definition outside the class. For example:

```
class Node {
    // ...
    static int node_count;          // declaration
};

int Node::node_count = 0;           // definition
```

However, for a few simple special cases, it is possible to initialize a **static** member in the class declaration. The **static** member must be a **const** of an integral or enumeration type, or a **constexpr** of a literal type (§10.4.3), and the initializer must be a *constant-expression*. For example:

```
class Curious {
public:
    static const int c1 = 7;            // OK
    static int c2 = 11;                 // error: not const
    const int c3 = 13;                  // OK, but not static (§17.4.4)
    static const int c4 = sqrt(9);      // error: in-class initializer not constant
    static const float c5 = 7.0;        // error: in-class not integral (use constexpr rather than const)
    // ...
};
```

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;              // don't repeat initializer here
const int* p = &Curious::c1;        // OK: Curious::c1 has been defined
```

The main use of member constants is to provide symbolic names for constants needed elsewhere in the class declaration. For example:

```
template<class T, int N>
class Fixed {    // fixed-size array
public:
    static constexpr int max = N;
    // ...
private:
    T a[max];
};
```

For integers, enumerators (§8.4) offer an alternative for defining symbolic constants within a class declaration. For example:

```
class X {
    enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17 };
    // ...
};
```

17.5 Copy and Move

When we need to transfer a value from `a` to `b`, we usually have two logically distinct options:

- *Copy* is the conventional meaning of `x=y`; that is, the effect is that the values of `x` and `y` are both equal to `y`'s value before the assignment.
- *Move* leaves `x` with `y`'s former value and `y` with some *moved-from state.* For the most interesting cases, containers, that moved-from state is "empty."

This simple logical distinction is confounded by tradition and by the fact that we use the same notation for both move and copy.

Typically, a move cannot throw, whereas a copy might (because it may need to acquire a resource), and a move is often more efficient than a copy. When you write a move operation, you should leave the source object in a valid but unspecified state because it will eventually be destroyed and the destructor cannot destroy an object left in an invalid state. Also, standard-library algorithms rely on being able to assign to (using move or copy) a moved-from object. So, design your moves not to throw, and to leave their source objects in a state that allows destruction and assignment.

To save us from tedious repetitive work, copy and move have default definitions (§17.6.2).

17.5.1 Copy

Copy for a class `X` is defined by two operations:

- Copy constructor: `X(const X&)`
- Copy assignment: `X& operator=(const X&)`

You can define these two operations with more adventurous argument types, such as `volatile X&`, but don't; you'll just confuse yourself and others. A copy constructor is supposed to make a copy of an object without modifying it. Similarly, you can use `const X&` as the return type of the copy assignment. My opinion is that doing so causes more confusion than it is worth, so my discussion of copy assumes that the two operations have the conventional types.

Consider a simple two-dimensional `Matrix`:

```
template<class T>
class Matrix {
    array<int,2> dim;    // two dimensions
    T* elem;             // pointer to dim[0]*dim[1] elements of type T
public:
    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1*d2]} {}      // simplified (no error handling)
    int size() const { return dim[0]*dim[1]; }

    Matrix(const Matrix&);                // copy constructor
    Matrix& operator=(const Matrix&);     // copy assignment

    Matrix(Matrix&&);                     // move constructor
    Matrix& operator=(Matrix&&);          // move assignment

    ~Matrix() { delete[] elem; }
    // ...
};
```

First we note that the default copy (copy the members) would be disastrously wrong: the **Matrix** elements would not be copied, the **Matrix** copy would have a pointer to the same elements as the source, and the **Matrix** destructor would delete the (shared) elements twice (§3.3.1).

However, the programmer can define any suitable meaning for these copy operations, and the conventional one for a container is to copy the contained elements:

```
template<class T>
Matrix:: Matrix(const Matrix& m)          // copy constructor
      : dim{m.dim},
        elem{new T[m.size()]}
{
      uninitialized_copy(m.elem,m.elem+m.size(),elem);      // copy elements
}

template<class T>
Matrix& Matrix::operator=(const Matrix& m)        // copy assignment
{
      if (dim[0]!=m.dim[0] || dim[1]!=m.dim[1])
            throw runtime_error("bad size in Matrix =");
      copy(m.elem,m.elem+m.size(),elem);     // copy elements
}
```

A copy constructor and a copy assignment differ in that a copy constructor initializes uninitialized memory, whereas the copy assignment operator must correctly deal with an object that has already been constructed and may own resources.

The **Matrix** copy assignment operator has the property that if a copy of an element throws an exception, the target of the assignment may be left with a mixture of its old value and the new. That is, that **Matrix** assignment provided the basic guarantee, but not the strong guarantee (§13.2). If that is not considered acceptable, we can avoid it by the fundamental technique of first making a copy and then swapping representations:

```
Matrix& Matrix::operator=(const Matrix& m)        // copy assignment
{
      Matrix tmp {m};          // make a copy
      swap(tmp,*this);         // swap tmp's representation with *this's
      return *this;
}
```

The **swap()** will be done only if the copy was successful. Obviously, this **operator=()** works only if the implementation **swap()** does not use assignment (**std::swap()** does not); see §17.5.2.

Usually a copy constructor must copy every non-**static** member (§17.4.1). If a copy constructor cannot copy an element (e.g., because it needs to acquire an unavailable resource to do so), it can throw an exception.

Note that I did not protect **Matrix**'s copy assignment against self-assignment, **m=m**. The reason I did not test is that self-assignment of the members is already safe: both my implementations of **Matrix**'s copy assignment will work correctly and reasonably efficiently for **m=m**. Also, self-assignment is rare, so test for self-assignment in a copy assignment only if you are sure that you need to.

17.5.1.1 Beware of Default Constructors

When writing a copy operation, be sure to copy every base and member. Consider:

```
class X {
        string s;
        string s2;
        vector<string> v;

        X(const X&)             // copy constructor
                :s{a.s}, v{a.v}  // probably sloppy and probably wrong
        {
        }
        // ...
};
```

Here, I ''forgot'' to copy `s2`, so it gets default initialized (to `""`). This is unlikely to be right. It is also unlikely that I would make this mistake for a simple class. However, for larger classes the chances of forgetting go up. Worse, when someone long after the initial design adds a member to a class, it is easy to forget to add it to the list of members to be copied. This is one reason to prefer the default (compiler-generated) copy operations (§17.6).

17.5.1.2 Copy of Bases

For the purposes of copying, a base is just a member: to copy an object of a derived class you have to copy its bases. For example:

```
struct B1 {
        B1();
        B1(const B1&);
        // ...
};

struct B2 {
        B2(int);
        B2(const B2&);
        // ...
};

struct D : B1, B2 {
        D(int i) :B1{}, B2{i}, m1{}, m2{2*i} {}
        D(const D& a) :B1{a}, B2{a}, m1{a.m1}, m2{a.m2} {}
        B1 m1;
        B2 m2;
};

D d {1};    // construct with int argument
D dd {d};   // copy construct
```

The order of initialization is the usual (base before member), but for copying the order had better not matter.

A **virtual** base (§21.3.5) may appear as a base of several classes in a hierarchy. A default copy constructor (§17.6) will correctly copy it. If you define your own copy constructor, the simplest technique is to repeatedly copy the **virtual** base. Where the base object is small and the **virtual** base occurs only a few times in a hierarchy, that can be more efficient than techniques for avoiding the replicated copies.

17.5.1.3 The Meaning of Copy

What does a copy constructor or copy assignment have to do to be considered ''a proper copy operation''? In addition to be declared with a correct type, a copy operation must have the proper copy semantics. Consider a copy operation, **x=y**, of two objects of the same type. To be suitable for value-oriented programming in general (§16.3.4), and for use with the standard library in particular (§31.2.2), the operation must meet two criteria:

- *Equivalence*: After **x=y**, operations on **x** and **y** should give the same result. In particular, if **==** is defined for their type, we should have **x==y** and **f(x)==f(y)** for any function **f()** that depends only on the values of **x** and **y** (as opposed to having its behavior depend on the addresses of **x** and **y**).
- *Independence*: After **x=y**, operations on **x** should not implicitly change the state of **y**, that is **f(x)** does not change the value of **y** as long as **f(x)** doesn't refer to **y**.

This is the behavior that **int** and **vector** offer. Copy operations that provide equivalence and independence lead to simpler and more maintainable code. This is worth stating because code that violate these simple rules is not uncommon, and programmers don't always realize that such violations are the root cause of some of their nastier problems. A copy that provides equivalence and independence is part of the notion of a regular type (§24.3.1).

First consider the requirement of equivalence. People rarely violate this requirement deliberately, and the default copy operations do not violate it; they do memberwise copy (§17.3.1, §17.6.2). However, tricks, such as having the meaning of copy depend on ''options,'' occasionally appear and typically cause confusion. Also, it is not uncommon for an object to contain members that are not considered part of its value. For example, a copy of a standard container does not copy its allocator because the allocator is considered part of the container, rather than part of its value. Similarly, counters for statistics gathering and cached values are sometimes not simply copied. Such ''non-value'' parts of an object's state should not affect the result of comparison operators. In particular, **x=y** should imply **x==y**. Furthermore, slicing (§17.5.1.4) can lead to ''copies'' that behave differently, and is most often a bad mistake.

Now consider the requirement of independence. Most of the problems related to (lack of) independence have to do with objects that contain pointers. The default meaning of copy is memberwise copy. A default copy operation copies a pointer member, but does not copy the object (if any) that it points to. For example:

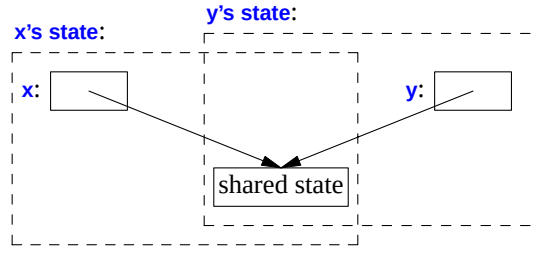
```
struct S {
    int* p;     // a pointer
};

S x {new int{0}};
```

```
void f()
{
    S y {x};            // "copy" x

    *y.p = 1;           // change y; affects x
    *x.p = 2;           // change x; affects y
    delete y.p;         // affects x and y
    y.p = new int{3};   // OK: change y; does not affect x
    *x.p = 4;           // oops: write to deallocated memory
}
```

Here I violated the rule of independence. After the "copy" of **x** into **y**, we can manipulate part of **x**'s state through **y**. This is sometimes called *shallow copy* and (too) often praised for "efficiency." The obvious alternative of copying the complete state of an object is called *deep copy*. Often, the better alternative to deep copy is not a shallow copy, but a move operation, which minimizes copying without adding complexity (§3.3.2, §17.5.2).

A shallow copy leaves two objects (here, **x** and **y**) with a *shared state*, and has a huge potential for confusion and errors. We say that the objects **x** and **y** have become *entangled* when the requirement of independence have been violated. It is not possible to reason about an entangled object in isolation. For example, it is not obvious from the source code that the two assignments to ***x.p** can have dramatically different effects.

We can represent two entangled objects graphically:



Note that entanglement can arise in a variety of ways. Often, it is not obvious that entanglement has happened until probems arise. For example, a type like **S** may incautiously be used as a member of an otherwise well-behaved class. The original author of **S** may be aware of the entanglement and prepared to cope with it, but someone naively assuming that copying an **S** meant copying its complete value could be surprised, and someone who finds an **S** deeply nested in other classes could be very surprised.

We can address problems related to the lifetime of a shared subobject by introducing a form of garbage collection. For example:

```
struct S2 {
    shared_ptr<int> p;
};

S2 x {new int{0}};
```

```
void f()
{
        S2 y {x};                   // "copy" x

        *y.p = 1;                   // change y, affects x
        *x.p = 2;                   // change x; affects y
        y.p.reset(new int{3});      // change y; affects x
        *x.p = 4;                   // change x; affects y
}
```

In fact, shallow copy and such entangled objects are among the sources of demands for garbage collection. Entangled objects lead to code that is very hard to manage without some form of garbage collection (e.g., **shared_ptr**s).

However, a **shared_ptr** is still a pointer, so we cannot consider objects containing a **shared_ptr** in isolation. Who can update the pointed-to object? How? When? If we are running in a multi-threaded system, is synchronization needed for access to the shared data? How can we be sure? Entangled objects (here, resulting from a shallow copy) is a source of complexity and errors that is at best partially solved by garbage collection (in any form).

Note that an immutable shared state is not a problem. Unless we compare addresses, we cannot tell whether two equal values happen to be represented as one or two copies. This is a useful observation because many copies are never modified. For example, objects passed by value are rarely written to. This observation leads to the notion of *copy-on-write*. The idea is that a copy doesn't actually need independence until a shared state is written to, so we can delay the copying of the shared state until just before the first write to it. Consider:

```
class Image {
public:
        // ...
        Image(const Image& a);          // copy constructor
        // ...
        void write_block(Descriptor);
        // ...
private:
        Representation* clone();        // copy *rep
        Representation* rep;
        bool shared;
};
```

Assume that a **Representation** can be huge and that a **write_block()** is expensive compared to testing a **bool**. Then, depending on the use of **Image**s, it can make sense to implement the copy constructor as a shallow copy:

```
Image::Image(const Image& a)        // do shallow copy and prepare for copy-on-write
        :rep{a.rep},
         shared{true}
{
}
```

We protect the argument to that copy constructor by copying the **Representation** before a write:

```
void write_block(Descriptor d)
{
        if (shared) {
                rep =clone();           // make a copy of *rep
                shared = false;         // no more sharing
        }
        // ... now we can safely write to our own copy of rep ...
}
```

Like any other technique, copy-on-write is not a panacea, but it can be an effective combination of the simplicity of true copy and the efficiency of shallow copy.

17.5.1.4 Slicing

A pointer to a derived class implicitly converts to a pointer to its public base class. When applied to a copy operation, this simple and necessary rule (§3.2.4, §20.2) leads to a trap for the unwary. Consider:

```
struct Base {
        int b;
        Base(const Base&);
        // ...
};

struct Derived : Base {
        int d;
        Derived(const Derived&);
        // ...
};

void naive(Base* p)
{
        B b2 = *p;      // may slice: invokes Base::Base(const Base&)
        // ...
}

void user()
{
        Derived d;
        naive(&d);
        Base bb = d;    // slices: invokes Base::Base(const Base&), not Derived::Derived(const Derived&)
        // ...
}
```

The variables **b2** and **bb** contain copies of the **Base** part of **d**, that is, a copy of **d.b**. The member **d.d** is not copied. This phenomenon is called *slicing*. It may be exactly what you intended (e.g., see the copy constructor for **D** in §17.5.1.2 where we pass selected information to a base class), but typically it is a subtle bug. If you don't want slicing, you have two major tools to prevent it:

[1] Prohibit copying of the base class: **delete** the copy operations (§17.6.4).
[2] Prevent conversion of a pointer to a derived to a pointer to a base: make the base class a **private** or **protected** base (§20.5).

The former would make the initializations of **b2** and **bb** errors; the latter would make the call of **naive()** and the initialization of **bb** errors.

17.5.2 Move

The traditional way of getting a value from **a** to **b** is to copy it. For an integer in a computer's memory, that's just about the only thing that makes sense: that's what the hardware can do with a single instruction. However, from a general and logical point of view that's not so. Consider the obvious implementation of **swap()** exchanging the value of two objects:

```
template<class T>
void swap(T& a, T& b)
{
    const T tmp = a;    // put a copy of a into tmp
    a = b;              // put a copy of b into a
    b = tmp;            // put a copy of tmp into b
};
```

After the initialization of **tmp**, we have two copies of **a**'s value. After the assignment to **tmp**, we have two copies of **b**'s value. After the assignment to **b**, we have two copies of **tmp**'s value (that is, the original value of **a**). Then we destroy **tmp**. That sounds like a lot of work, and it can be. For example:

```
void f(string& s1, string& s2,
    vector<string>& vs1, vector<string>& vs2,
    Matrix& m1, Matrix& m2)
{
    swap(s1,s2);
    swap(vs1.vs2);
    swap(m1,m2);
}
```

What if **s1** has a thousand characters? What if **vs2** has a thousand elements each of a thousand characters? What if **m1** is a 1000∗1000 matrix of **double**s? The cost of copying those data structures could be significant. In fact, the standard-library **swap()** has always been carefully designed to avoid such overhead for **string** and **vector**. That is, effort has been made to avoid copying (taking advantage of the fact that **string** and **vector** objects really are just handles to their elements). Similar work must be done to avoid a serious performance problem for **swap()** of **Matrix**es. If the only operation we have is copy, similar work must be done for huge numbers of functions and data structures that are not part of the standard.

The fundamental problem is that we really didn't want to do any copying at all: we just wanted to exchange pairs of values.

We can also look at the issue of copying from a completely different point of view: we don't usually copy physical things unless we absolutely have to. If you want to borrow my phone, I pass my phone to you rather than making you your own copy. If I lend you my car, I give you a key and

you drive away in my car, rather than in your freshly made copy of my car. Once I have given you an object, you have it and I no longer do. Consequently, we talk about ''giving away,'' ''handing over,'' ''transferring ownership of,'' and ''moving'' physical objects. Many objects in a computer resemble physical objects (which we don't copy without need and only at considerable cost) more than integer values (which we typically copy because that's easier and cheaper than alternatives). Examples are locks, sockets, file handles, threads, long strings, and large vectors.

To allow the user to avoid the logical and performance problems of copying, C++ directly supports the notion of *moving* as well as the notion of *copying*. In particular, we can define *move constructors* and *move assignments* to move rather than copy their argument. Consider again the simple two-dimensional **Matrix** from §17.5.1:

```
template<class T>
class Matrix {
    std::array<int,2> dim;
    T* elem;  // pointer to sz elements of type T

    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1*d2]} {}
    int size() const { return dim[0]*dim[1]; }

    Matrix(const Matrix&);              // copy constructor
    Matrix(Matrix&&);                   // move constructor

    Matrix& operator=(const Matrix&);   // copy assignment
    Matrix& operator=(Matrix&&);        // move assignment

    ~Matrix();  // destructor
    // ...
};
```

The **&&** indicates an rvalue reference (§7.7.2).

The idea behind a move assignment is to handle lvalues separately from rvalues: copy assignment and copy constructors take lvalues whereas move assignment and move constructors take rvalues. For a **return** value, the move constructor is chosen.

We can define **Matrix**'s move constructor to simply take the representation from its source and replace it with an empty **Matrix** (which is cheap to destroy). For example:

```
template<class T>
Matrix<T>::Matrix(Matrix&& a)       // move constructor
    :dim{a.dim}, elem{a.elem}       // grab a's representation
{
    a.dim = {0,0};                  // clear a's representation
    a.elem = nullptr;
}
```

For the move assignment, we can simply do a swap. The idea behind using a swap to implement a move assignment is that the source is just about to be destroyed, so we can just let the destructor for the source do the necessary cleanup work for us:

```
template<class T>
Matrix<T>& Matrix<T>::operator=(Matrix&& a)          // move assignment
{
        swap(dim,a.dim);                 // swap representations
        swap(elem,a.elem);
        return *this;
}
```

Move constructors and move assignments take non-**const** (rvalue) reference arguments: they can, and usually do, write to their argument. However, the argument of a move operation must always be left in a state that the destructor can cope with (and preferably deal with very cheaply and easily).

For resource handles, move operations tend to be significantly simpler and more efficient than copy operations. In particular, move operations typically do not throw exceptions; they don't acquire resources or do complicated operations, so they don't need to. In this, they differ from many copy operations (§17.5).

How does the compiler know when it can use a move operation rather than a copy operation? In a few cases, such as for a return value, the language rules say that it can (because the next action is defined to destroy the element). However, in general we have to tell it by giving an rvalue reference argument. For example:

```
template<class T>
void swap(T& a, T& b)     // "perfect swap" (almost)
{
        T tmp = std::move(a);
        a = std::move(b);
        b = std::move(tmp);
}
```

The **move()** is a standard-library function returning an rvalue reference to its argument (§35.5.1): **move(x)** means "give me an rvalue reference to **x**." That is, **std::move(x)** does not move anything; instead, it allows a user to move **x**. It would have been better if **move()** had been called **rval()**, but the name **move()** has been used for this operation for years.

Standard-library containers have move operations (§3.3.2, §35.5.1) and so have other standard-library types, such as **pair** (§5.4.3, §34.2.4.1) and **unique_ptr** (§5.2.1, §34.3.1). Furthermore, operations that insert new elements into standard-library containers, such as **insert()** and **push_back()**, have versions that take rvalue references (§7.7.2). The net result is that the standard containers and algorithms deliver better performance than they would have been able to if they had to copy.

What if we try to swap objects of a type that does not have a move constructor? We copy and pay the price. In general, a programmer is responsible for avoiding excessive copying. It is not the compiler's job to decide what is excessive and what is necessary. To get the copy-to-move optimization for your own data structures, you have to provide move operations (either explicitly or implicitly; see §17.6).

Built-in types, such as **int** and **double***, are considered to have move operations that simply copy. As usual, you have to be careful about data structures containing pointers (§3.3.1). In particular, don't assume that a moved-from pointer is set to **nullptr**.

Having move operations affects the idiom for returning large objects from functions. Consider:

```
Matrix operator+(const Matrix& a, const Matrix& b)
    // res[i][j] = a[i][j]+b[i][j] for each i and j
{
    if (a.dim[0]!=b.dim[0] || a.dim[1]!=b.dim[1])
        throw std::runtime_error("unequal Matrix sizes in +");

    Matrix res{a.dim[0],a.dim[1]};
    constexpr auto n = a.size();
    for (int i = 0; i!=n; ++i)
        res.elem[i] = a.elem[i]+b.elem[i];
    return res;
}
```

Matrix has a move constructor so that ''return by value'' is simple and efficient as well as ''natural.'' Without move operations, we have performance problems and must resort to workarounds. We might have considered:

```
Matrix& operator+(const Matrix& a, const Matrix& b)    // beware!
{
    Matrix& res = *new Matrix;    // allocate on free store
    // res[i][j] = a[i][j]+b[i][j] for each i and j
    return res;
}
```

The use of **new** within **operator+()** is not obvious and forces the user of **+** to deal with tricky memory management issues:

- How does the object created by **new** get **delete**d?
- Do we need a garbage collector?
- Should we use a pool of **Matrix**es rather than the general **new**?
- Do we need use-counted **Matrix** representations?
- Should we redesign the interface of our **Matrix** addition?
- Must the caller of **operator+()** remember to **delete** the result?
- What happens to the newly allocated memory if the computation throws an exception?

None of the alternatives are elegant or general.

17.6 Generating Default Operations

Writing conventional operations, such as a copy and a destructor, can be tedious and error-prone, so the compiler can generate them for us as needed. By default, a class provides:

- A default constructor: **X()**
- A copy constructor: **X(const X&)**
- A copy assignment: **X& operator=(const X&)**
- A move constructor: **X(X&&)**
- A move assignment: **X& operator=(X&&)**
- A destructor: **~X()**

By default, the compiler generates each of these operations if a program uses it. However, if the programmer takes control by defining one or more of those operations, the generation of related operations is suppressed:

- If the programmer declares any constructor for a class, the default constructor is not generated for that class.
- If the programmer declares a copy operation, a move operation, or a destructor for a class, no copy operation, move operation, or destructor is generated for that class.

Unfortunately, the second rule is only incompletely enforced: for backward compatibility, copy constructors and copy assignments are generated even if a destructor is defined. However, that generation is deprecated in the ISO standard (§iso.D), and you should expect a modern compiler to warn against it.

If necessary, we can be explicit about which functions are generated (§17.6.1) and which are not (§17.6.4).

17.6.1 Explicit Defaults

Since the generation of otherwise default operations can be suppressed, there has to be a way of getting back a default. Also, some people prefer to see a complete list of operations in the program text even if that complete list is not needed. For example, we can write:

```
class gslice {
        valarray<size_t>  size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        gslice() = default;
        ~gslice() = default;
        gslice(const gslice&) = default;
        gslice(gslice&&) = default;
        gslice& operator=(const gslice&) = default;
        gslice& operator=(gslice&&) = default;
        // ...
};
```

This fragment of the implementation of `std::gslice` (§40.5.6) is equivalent to:

```
class gslice {
        valarray<size_t> size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        // ...
};
```

I prefer the latter, but I can see the point of using the former in code bases maintained by less experienced C++ programmers: what you don't see, you might forget about.

Using `=default` is always better than writing your own implementation of the default semantics. Someone assuming that it is better to write something, rather than nothing, might write:

```
class gslice {
        valarray<size_t> size;
        valarray<size_t> stride;
        valarray<size_t> d1;
public:
        // ...
        gslice(const gslice& a);
};

gslice::gslice(const gslice& a)
        : size{a.size },
          stride{a.stride},
          d1{a.d1}
{
}
```

This is not only verbose, making it harder to read the definition of **gslice**, but also opens the opportunity for making mistakes. For example, I might forget to copy one of the members and get it default initialized (rather than copied). Also, when the user provides a function, the compiler no longer knows the semantics of that function and some optimizations become inhibited. For the default operations, those optimizations can be significant.

17.6.2 Default Operations

The default meaning of each generated operation, as implemented when the compiler generates it, is to apply the operation to each base and non-**static** data member of the class. That is, we get memberwise copy, memberwise default construction, etc. For example:

```
struct S {
        string a;
        int b;
};

S f(S arg)
{
        S s0 {};    // default construction: {"",0}
        S s1 {s0}; // copy construction
        s1 = arg;  // copy assignment
        return s1; // move construction
}
```

The copy construction of **s1** copies **s0.a** and **s0.b**. The **return** of **s1** moves **s1.a** and **s1.b**, leaving **s1.a** as the empty string and **s1.b** unchanged.

Note that the value of a moved-from object of a built-in type is unchanged. That's the simplest and fastest thing for the compiler to do. If we want something else done for a member of a class, we have to write our move operations for that class.

The default moved-from state is one for which the default destructor and default copy assignment work correctly. It is not guaranteed (or required) that an arbitrary operation on a moved-from object will work correctly. If you need stronger guarantees, write your own operations.

17.6.3 Using Default Operations

This section presents a few examples demonstrating how copy, move, and destructors are logically linked. If they were not linked, errors that are obvious when you think about them would not be caught by the compiler.

17.6.3.1 Default Constructors

Consider:

```
struct X {
    X(int);     // require an int to initialize an X
};
```

By declaring a constructor that requires an integer argument, the programmer clearly states that a user needs to provide an `int` to initialize an `X`. Had we allowed the default constructor to be generated, that simple rule would have been violated. We have:

```
X a {1};    // OK
X b {};     // error: no default constructor
```

If we also want the default constructor, we can define one or declare that we want the default generated by the compiler. For example:

```
struct Y {
    string s;
    int n;
    Y(const string& s);  // initialize Y with a string
    Y() = default;       // allow default initialization with the default meaning
};
```

The default (i.e., generated) default constructor default constructs each member. Here, `Y()` sets `s` to the empty string. The ''default initialization'' of a built-in member leaves that member uninitialized. Sigh! Hope for a compiler warning.

17.6.3.2 Maintaining Invariants

Often, a class has an invariant. If so, we want copy and move operations to maintain it and the destructor to free any resources involved. Unfortunately, the compiler cannot in every case know what a programmer considers an invariant. Consider a somewhat far-fetched example:

```
struct Z { // invariant:
           // my_favorite is the index of my favorite element of elem
           // largest points to the element with the highest value in elem
    vector<int> elem;
    int my_favorite;
    int* largest;
};
```

The programmer stated an invariant in the comment, but the compiler doesn't read comments. Furthermore, the programmer did not leave a hint about how that invariant is to be established and maintained. In particular, there are no constructors or assignments declared. That invariant is

implicit. The result is that a **Z** can be copied and moved using the default operations:

```
Z v0;                    // no initialization (oops! possibility of undefined values)
Z val {{1,2,3},1,&val[2]}; // OK, but ugly and error-prone
Z v2 = val;              // copies: v2.largest points into val
Z v3 = move(val);        // moves: val.elem becomes empty; v3.my_favorite is out of range
```

This is a mess. The root problem is that **Z** is badly designed because critical information is ''hidden'' in a comment or completely missing. The rules for the generation of default operations are heuristic intended to catch common mistakes and to encourage a systematic approach to construction, copy, move, and destruction. Wherever possible

[1] Establish an invariant in a constructor (including possibly resource acquisition).
[2] Maintain the invariant with copy and move operations (with the usual names and types).
[3] Do any needed cleanup in the destructor (incl. possibly resource release).

17.6.3.3 Resource Invariants

Many of the most critical and obvious uses of invariants relate to resource management. Consider a simple **Handle**:

```
template<class T> class Handle {
    T* p;
public:
    Handle(T* pp) :p{pp} { }
    T& operator*() { return *p; }
    ~Handle() { delete p; }
};
```

The idea is that you construct a **Handle** given a pointer to an object allocated using **new**. The **Handle** provides access to the object pointed to and eventually **delete**s that object. For example:

```
void f1()
{
    Handle<int> h {new int{99}};
    // ...
}
```

Handle declares a constructor that takes an argument: this suppresses the generation of the default constructor. That's good because a default constructor could leave **Handle<T>::p** uninitialized:

```
void f2()
{
    Handle<int> h; // error: no default constructor
    // ...
}
```

The absence of a default constructor saves us from the possibility of a **delete** with a random memory address.

Also, **Handle** declares a destructor: this suppresses the generation of copy and move operations. Again, that saves us from a nasty problem. Consider:

```
void f3()
{
    Handle<int> h1 {new int{7}};
    Handle<int> h2 {h1};          // error: no copy constructor
    // ...
}
```

Had **Handle** had a default copy constructor, both **h1** and **h2** would have had a copy of the pointer and both would have **delete**d it. The results would be undefined and most likely disastrous (§3.3.1). Caveat: the generation of copy operations is only deprecated, not banned, so if you ignore warnings, you might get this example past the compiler. In general, if a class has a pointer member, the default copy and move operations should be considered suspicious. If that pointer member represents ownership, memberwise copy is wrong. If that pointer member does not represent ownership and memberwise copy *is* appropriate, explicit **=default** and a comment are most likely a good idea.

If we wanted copy construction, we could define something like:

```
template<class T>
class Handle {
    // ...
    Handle(const T& a) :p{new T{*a.p}} { }          // clone
};
```

17.6.3.4 Partially Specified Invariants

Troublesome examples that rely on invariants but only partially express them through constructors or destructors are rarer but not unheard of. Consider:

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}    // always 9 positions

    Tic_tac_toe& operator=(const Tic_tac_toe& arg)
    {
        for(int i = 0; i<9; ++i)
            pos.at(i) = arg.pos.at(i);
        return *this;
    }

    // ... other operations ...

    enum State { empty, nought, cross };
private:
    vector<State> pos;
};
```

This was reported to have been part of a real program. It uses the ''magic number'' **9** to implement a copy assignment that accesses its argument **arg** without checking that the argument actually has nine elements. Also, it explicitly implements the copy assignment, but not the copy constructor. This is not what I consider good code.

We defined copy assignment, so we must also define the destructor. That destructor can be **=default** because all it needs to do is to ensure that the member **pos** is destroyed, which is what would have been done anyway had the copy assignment not been defined. At this point, we notice that the user-defined copy assignment is essentially the one we would have gotten by default, so we can **=default** that also. Add a copy constructor for completeness and we get:

```
class Tic_tac_toe {
public:
    Tic_tac_toe(): pos(9) {}    // always 9 positions
    Tic_tac_toe(const Tic_tac_toe&) = default;
    Tic_tac_toe& operator=(const Tic_tac_toe& arg) = default;
    ~Tic_tac_toe() = default;

    // ... other operations ...

    enum State { empty, nought, cross };
private:
    vector<State> pos;
};
```

Looking at this, we realize that the net effect of these **=default**s is just to eliminate move operations. Is that what we want? Probably not. When we made the copy assignment **=default**, we eliminated the nasty dependence on the magic constant **9**. Unless other operations on **Tic_tac_toe**, not mentioned so far, are also ‘‘hardwired with magic numbers,’’ we can safely add move operations. The simplest way to do that is to remove the explicit **=default**s, and then we see that **Tic_tac_toe** is really a perfectly ordinary type:

```
class Tic_tac_toe {
public:
    // ... other operations ...
    enum State { empty, nought, cross };
private:
    vector<State> pos {Vector<State>(9)};        // always 9 positions
};
```

One conclusion that I draw from this and other examples where an ‘‘odd combination’’ of the default operations is defined is that we should be highly suspicious of such types: their irregularity often hides design flaws. For every class, we should ask:

[1] Is a default constructor needed (because the default one is not adequate or has been suppressed by another constructor)?

[2] Is a destructor needed (e.g., because some resource needs to be released)?

[3] Are copy operations needed (because the default copy semantics is not adequate, e.g., because the class is meant to be a base class or because it contains pointers to objects that must be deleted by the class)?

[4] Are move operations needed (because the default semantics is not adequate, e.g., because an empty object doesn’t make sense)?

In particular, we should never just consider one of these operations in isolation.

17.6.4 deleted Functions

We can ‘‘delete’’ a function; that is, we can state that a function does not exist so that it is an error to try to use it (implicitly or explicitly). The most obvious use is to eliminate otherwise defaulted functions. For example, it is common to want to prevent the copying of classes used as bases because such copying easily leads to slicing (§17.5.1.4):

```
class Base {
        // ...
        Base& operator=(const Base&) = delete;// disallow copying
        Base(const Base&) = delete;

        Base& operator=(Base&&) = delete;           // disallow moving
        Base(Base&&) = delete;
};

Base x1;
Base x2 {x1};   // error: no copy constructor
```

Enabling and disabling copy and move is typically more conveniently done by saying what we want (using `=default`; §17.6.1) rather than saying what we don’t want (using `=delete`). However, we can `delete` any function that we can declare. For example, we can eliminate a specialization from the set of possible specializations of a function template:

```
template<class T>
T* clone(T* p)  // return copy of *p
{
        return new T{*p};
};

Foo* clone(Foo*) = delete;          // don't try to clone a Foo

void f(Shape* ps, Foo* pf)
{
        Shape* ps2 = clone(ps);     // fine
        Foo* pf2 = clone(pf);       // error: clone(Foo*) deleted
}
```

Another application is to eliminate an undesired conversion. For example:

```
struct Z {
        // ...
        Z(double);          // can initialize with a double
        Z(int) = delete;    // but not with an integer
};

void f()
{
        Z z1 {1};       // error: Z(int) deleted
        Z z2 {1.0};     // OK
}
```

A further use is to control where a class can be allocated:

```
class Not_on_stack {
    // ...
    ~Not_on_stack() = delete;
};

class Not_on_free_store {
    // ...
    void* operator new(size_t) = delete;
};
```

You can't have a local variable that can't be destroyed (§17.2.2), and you can't allocate an object on the free store when you have =deleted its class's memory allocation operator (§19.2.5). For example:

```
void f()
{
    Not_on_stack v1;              // error: can't destroy
    Not_on_free_store v2;         // OK

    Not_on_stack* p1 = new Not_on_stack;              // OK
    Not_on_free_store* p2 = new Not_on_free_store;    // error: can't allocate
}
```

However, we can never delete that Not_on_stack object. The alternative technique of making the destructor private (§17.2.2) can address that problem.

Note the difference between a =deleted function and one that simply has not been declared. In the former case, the compiler notes that the programmer has tried to use the deleted function and gives an error. In the latter case, the compiler looks for alternatives, such as not invoking a destructor or using a global operator new().

17.7 Advice

[1] Design constructors, assignments, and the destructor as a matched set of operations; §17.1.
[2] Use a constructor to establish an invariant for a class; §17.2.1.
[3] If a constructor acquires a resource, its class needs a destructor to release the resource; §17.2.2.
[4] If a class has a virtual function, it needs a virtual destructor; §17.2.5.
[5] If a class does not have a constructor, it can be initialized by memberwise initialization; §17.3.1.
[6] Prefer {} initialization over = and () initialization; §17.3.2.
[7] Give a class a default constructor if and only if there is a "natural" default value; §17.3.3.
[8] If a class is a container, give it an initializer-list constructor; §17.3.4.
[9] Initialize members and bases in their order of declaration; §17.4.1.
[10] If a class has a reference member, it probably needs copy operations (copy constructor and copy assignment); §17.4.1.1.

[11] Prefer member initialization over assignment in a constructor; §17.4.1.1.
[12] Use in-class initializers to provide default values; §17.4.4.
[13] If a class is a resource handle, it probably needs copy and move operations; §17.5.
[14] When writing a copy constructor, be careful to copy every element that needs to be copied (beware of default initializers); §17.5.1.1.
[15] A copy operations should provide equivalence and independence; §17.5.1.3.
[16] Beware of entangled data structures; §17.5.1.3.
[17] Prefer move semantics and copy-on-write to shallow copy; §17.5.1.3.
[18] If a class is used as a base class, protect against slicing; §17.5.1.4.
[19] If a class needs a copy operation or a destructor, it probably needs a constructor, a destructor, a copy assignment, and a copy constructor; §17.6.
[20] If a class has a pointer member, it probably needs a destructor and non-default copy operations; §17.6.3.3.
[21] If a class is a resource handle, it needs a constructor, a destructor, and non-default copy operations; §17.6.3.3.
[22] If a default constructor, assignment, or destructor is appropriate, let the compiler generate it (don't rewrite it yourself); §17.6.
[23] Be explicit about your invariants; use constructors to establish them and assignments to maintain them; §17.6.3.2.
[24] Make sure that copy assignments are safe for self-assignment; §17.5.1.
[25] When adding a new member to a class, check to see if there are user-defined constructors that need to be updated to initialize the member; §17.5.1.

18

Operator Overloading

*When **I** use a word it means just what*
I choose it to mean – neither more nor less.
– Humpty Dumpty

- Introduction
- Operator Functions
  Binary and Unary Operators; Predefined Meanings for Operators; Operators and User-Defined Types; Passing Objects; Operators in Namespaces
- A Complex Number Type
  Member and Nonmember Operators; Mixed-Mode Arithmetic; Conversions; Literals; Accessor Functions; Helper Functions
- Type Conversion
  Conversion Operators; **explicit** Conversion Operators; Ambiguities
- Advice

18.1 Introduction

Every technical field – and most nontechnical fields – has 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. Consider:

```
class complex {                 // very simplified complex
        double re, im;
public:
        complex(double r, double i) :re{r}, im{i} { }
        complex operator+(complex);
        complex operator*(complex);
};
```

This 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 {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**.

Note that the C++ grammar is written so that the **{}** notation can only be used for initializers and on the right-hand side of an assignment:

```
void g(complex a, complex b)
{
        a = {1,2};              // OK: right hand side of assignment
        a += {1,2};             // OK: right hand side of assignment
        b = a+{1,2};            // syntax error
        b = a+complex{1,2};     // OK
        g(a,{1,2});             // OK: a function argument is considered an initializer
        {a,b} = {b,a};          // syntax error
}
```

There seems to be no fundamental reason not to use **{}** in more places, but the technical problems of writing a grammar allowing **{}** everywhere in an expression (e.g., how would you know if a **{** after a semicolon was the start of an expression or a block?) and also giving good error messages led to a more limited use of **{}** in expressions.

Many of the most obvious uses of operator overloading are for numeric types. However, the usefulness of user-defined operators is not restricted to numeric types. For example, the design of general and abstract interfaces often leads to the use of operators such as **->**, **[]**, and **()**.

18.2 Operator Functions

Functions defining meanings for the following operators (§10.3) can be declared:

+	**–**	*****	**/**	**%**	**^**	**&**
|	**~**	**!**	**=**	**<**	**>**	**+=**
–=	***=**	**/=**	**%=**	**^=**	**&=**	**|=**
<<	**>>**	**>>=**	**<<=**	**==**	**!=**	**<=**
>=	**&&**	**||**	**++**	**––**	**->***	**,**
->	**[]**	**()**	**new**	**new[]**	**delete**	**delete[]**

The following operators cannot be defined by a user:

:: scope resolution (§6.3.4, §16.2.12)
. member selection (§8.2)
.* member selection through pointer to member (§20.6)

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]. The named ‘‘operators’’cannot be overloaded because they report fundamental facts about their operands:

sizeof size of object (§6.2.8)
alignof alignment of object (§6.2.9)
typeid **type_info** of an object (§22.5)

Finally, the ternary conditional expression operator cannot be overloaded (for no particularly fundamental reason):

?: conditional evaluation (§9.4.1)

In addition, user-defined literals (§19.2.6) are defined by using the **operator""** notation. This is a kind of syntactic subterfuge because there is no operator called **""**. Similarly, **operator T()** defines a conversion to a type **T** (§18.4).

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, 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)**? There are solutions to all such technical questions. However, it is most uncertain if applying subtle technical rules will lead to more readable and maintainable code. If in doubt, use a named function.

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.

18.2.1 Binary and Unary Operators

A binary operator can be defined by either a non-**static** 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 (§12.3) 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 non-**static** 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 (§12.3) 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 §19.2.4. If both are defined, overload resolution (§12.3) determines which, if any, interpretation is used. An operator can be declared only for the syntax defined for it in the grammar (§iso.A). For example, a user cannot define a unary **%** or a ternary **+**. Consider:

```
class X {
public:         // members (with implicit this pointer):

        X* operator&();         // prefix unary & (address of)
        X operator&(X);         // binary & (and)
        X operator++(int);      // postfix increment (see §19.2.4)
        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 §19.2.1, operator **()** in §19.2.2, operator **–>** in §19.2.3, operators **++** and **––** in §19.2.4, and the allocation and deallocation operators in §11.2.4 and §19.2.5.

The operators **operator=** (§18.2.2), **operator[]** (§19.2.1), **operator()** (§19.2.2), and **operator–>** (§19.2.3) must be non-**static** member functions.

The default meaning of **&&**, **||**, and **,** (comma) involves sequencing: the first operand is evaluated before the second (and for **&&** and **||** the second operand is not always evaluated). This special rule does not hold for user-defined versions of **&&**, **||**, and **,** (comma); instead these operators are treated exactly like other binary operators.

18.2.2 Predefined Meanings for Operators

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 defines them to. For example, a compiler will not generate a definition of **Z::operator+=()** from the definitions of **Z::operator+()** and **Z::operator=()**.

The operators **=** (assignment), **&** (address-of), and **,** (sequencing; §10.3.2) have predefined meanings when applied to class objects. These predefined meanings can be eliminated (‘‘deleted’’; §17.6.4):

```
class X {
public:
        // ...
        void operator=(const X&) = delete;
        void operator&() = delete;
        void operator,(const X&) = delete;
        // ...
};

void f(X a, X b)
{
        a = b;      // error: no operator=()
        &a;         // error: no operator&()
        a,b;        // error: no operator,()
}
```

Alternatively, they can be given new meanings by suitable definitions.

18.2.3 Operators and User-Defined Types

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 built-in type (§6.2.1) 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 the operator is commutative and so interpret `2+aa` as `aa+2`. This example is trivially handled using one or more nonmember functions (§18.3.2, §19.4).

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 : static_cast<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 overload resolution rules in §12.3.

18.2.4 Passing Objects

When we define an operator, we typically want to provide a conventional notation, for example, `a=b+c`. Consequently, we have limited choices of how to pass arguments to the operator function and how it returns its value. For example, we cannot require pointer arguments and expect programmers to use the address-of operator or return a pointer and expect the user to dereference it: `*a=&b+&c` is not acceptable.

For arguments, we have two main choices (§12.2):

- Pass-by-value
- Pass-by-reference

For small objects, say, one to four words, call-by-value is typically a viable alternative and often the one that gives the best performance. However, performance of argument passing and use depends on machine architecture, compiler interface conventions (Application Binary Interfaces; ABIs), and the number of times an argument is accessed (it almost always is faster to access an argument passed by value than one passed by reference). For example, assume that a `Point` is represented as a pair of `int`s:

```
void Point::operator+=(Point delta);      // pass-by-value
```

Larger objects, we pass by reference. For example, because a **Matrix** (a simple matrix of **double**s; §17.5.1) is most likely larger than a few words, we use pass-by-reference:

```
Matrix operator+(const Matrix&, const Matrix&);    // pass-by-const-reference
```

In particular, we use **const** references to pass large objects that are not meant to be modified by the called function (§12.2.1).

Typically, an operator returns a result. Returning a pointer or a reference to a newly created object is usually a very bad idea: using a pointer gives notational problems, and referring to an object on the free store (whether by a pointer or by a reference) results in memory management problems. Instead, return objects by value. For large objects, such as a **Matrix**, define move operations to make such transfers of values efficient (§3.3.2, §17.5.2). For example:

```
Matrix operator+(const Matrix& a, const Matrix& b)      // return-by-value
{
    Matrix res {a};
    return res+=b;
}
```

Note that operators that return one of their argument objects can – and usually do – return a reference. For example, we could define **Matrix**'s operator **+=** like this:

```
Matrix& Matrix::operator+=(const Matrix& a)      // return-by-reference
{
    if (dim[0]!=a.dim[0] || dim[1]!=a.dim[1])
        throw std::exception("bad Matrix += argument");

    double* p = elem;
    double* q = a.elem;
    double* end = p+dim[0]*dim[1];
    while(p!=end)
        *p++ += *q++

    return *this;
}
```

This is particularly common for operator functions that are implemented as members.

If a function simply passes an object to another function, an rvalue reference argument should be used (§17.4.3, §23.5.2.1, §28.6.3).

18.2.5 Operators in Namespaces

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 string {
        // ...
    };
```

```
        class ostream {
                // ...
                ostream& operator<<(const char*);          // output C-style string
        };

        extern ostream cout;

        ostream& operator<<(ostream&, const string&);   // output std::string
} // namespace std

int main()
{
        const 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 on my best behavior and didn't pollute the global namespace or in other ways introduce unnecessary dependencies.

The output operator for C-style strings 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 as functions can be found based on their argument types (§14.2.4). 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&)
```

Consider a binary operator `@`. If `x` is of type `X` and `y` is of type `Y`, `x@y` is resolved like this:

- If `X` is a class, look for `operator@` as a member of `X` or as a member of a base of `X`; and
- look for declarations of `operator@` in the context surrounding `x@y`; and
- if `X` is defined in namespace `N`, look for declarations of `operator@` in `N`; and
- if `Y` is defined in namespace `M`, look for declarations of `operator@` in `M`.

Declarations for several `operator@`s may be found and overload resolution rules (§12.3) 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 (§18.3.2, §18.4) will be considered. Note that a type alias is just a synonym and not a separate user-defined type (§6.5).

Unary operators are resolved analogously.

Note that in operator lookup no preference is given to members over nonmembers. This differs from lookup of named functions (§14.2.4). The lack of hiding of operators ensures that built-in operators are never inaccessible and that users can supply new meanings for an operator without modifying existing class declarations. For example:

```
X operator!(X);

struct Z {
        Z operator!();                  // does not hide ::operator!()
        X f(X x) { /* ... */ return !x; }       // invoke ::operator!(X)
        int f(int x) { /* ... */ return !x; }   // invoke the built-in ! for ints
};
```

In particular, the standard **iostream** library defines **<<** member functions to output built-in types, and a user can define **<<** to output user-defined types without modifying class **ostream** (§38.4.2).

18.3 A Complex Number Type

The implementation of complex numbers presented in §18.1 is too restrictive to please anyone. For example, we would expect this to work:

```
void f()
{
        complex a {1,2};
        complex b {3};
        complex c {a+2.3};
        complex d {2+b};
        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 §16.3. 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.

The **complex** type developed in this section uses **double** for its scalars and is roughly equivalent to the standard-library **complex<double>** (§40.4).

18.3.1 Member and Nonmember Operators

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 their 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)
{
        return a += b;   // access representation through +=
}
```

The arguments to this **operator+()** are passed by value, so **a+b** does not modify its operands.

Given these declarations, we can write:

```
void f(complex x, complex y, complex z)
{
        complex r1 {x+y+z};// r1 = operator+(operator+(x,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;
}
```

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 §19.4 discusses ways of defining operators with direct access to the representation of classes.

18.3.2 Mixed-Mode Arithmetic

To cope with **2+z**, where **z** is a **complex**, 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;
        }

        // ...
};
```

The three variants of **operator+()** can be defined outside **complex**:

```
complex operator+(complex a, complex b)
{
        return a += b;   // calls complex::operator+=(complex)
}

complex operator+(complex a, double b)
{
        return {a.real()+b,a.imag()};
}

complex operator+(double a, complex b)
{
        return {a+b.real(),b.imag()};
}
```

The access functions **real()** and **imag()** are defined in §18.3.6.

Given these declarations of **+**, we can write:

```
void f(complex x, complex y)
{
        auto r1 = x+y;   // calls operator+(complex,complex)
        auto r2 = x+2;   // calls operator+(complex,double)
        auto r3 = 2+x;   // calls operator+(double,complex)
        auto r4 = 2+3;   // built-in integer addition
}
```

I added the integer addition for completeness.

18.3.3 Conversions

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

void comp(complex x)
{
    x = 4;      // should mean x.re=4, x.im=0
    // ...
}
```

We can achieve that by providing a constructor that takes a single argument. 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} { }   // build a complex from a double
    // ...
};
```

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 {3,0};
```

A user-defined conversion is implicitly applied only if it is unique (§12.3). If you don't want a constructor to be used implicitly, declare it **explicit** (§16.2.6).

Naturally, we still need the constructor that takes two **double**s, 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} { }
        // ...
};
```

By default, copying **complex** values is defined as copying the real and imaginary parts (§16.2.2). For example:

```
void f()
{
        complex z;
        complex x {1,2};
        complex y {x}; // y also has the value {1,2}
        z = x;         // z also has the value {1,2}
}
```

18.3.3.1 Conversions of Operands

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, 27 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 overhead, 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 – contain 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 §12.3).

An object constructed by explicit or implicit use of a constructor in an expression is automatic and will be destroyed at the first opportunity (see §10.3.4).

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 approximate the notion that an operator requires an lvalue as its left-hand operand by making that operator a member. However, that is only an approximation because it is possible to access a temporary with a modifying operation, such as `operator+=()`:

```
complex x {4,5}
complex z {sqrt(x)+={1,2}};         // like tmp=sqrt(x), tmp+={1,2}
```

If we don't want implicit conversions, we can use `explicit` to suppress them (§16.2.6, §18.4.2).

18.3.4 Literals

We have literals of built-in types. For example, `1.2` and `12e3` are literals of type `double`. For `complex`, we can come pretty close to that by declaring constructors `constexpr` (§10.4). For example:

```
class complex {
public:
    constexpr complex(double r =0, double i =0) : re{r}, im{i} { }
    // ...
}
```

Given that, a `complex` can be constructed from its constituent parts at compile time just like a literal from a built-in type. For example:

```
complex z1 {1.2,12e3};
constexpr complex z2 {1.2,12e3};    // guaranteed compile-time initialization
```

When constructors are simple and inline, and especially when they are `constexpr`, it is quite reasonable to think of constructor invocations with literal arguments as literals.

It is possible to go further and introduce a user-defined literal (§19.2.6) in support of our `complex` type. In particular, we could define `i` to be a suffix meaning ''imaginary.'' For example:

```
constexpr complex<double> operator "" i(long double d)      // imaginary literal
{
    return {0,d};    // complex is a literal type
}
```

This would allow us to write:

```
complex z1 {1.2+12e3i};

complex f(double d)
{
        auto x {2.3i};
        return x+sqrt(d+12e3i)+12e3i;
}
```

This user-defined literal gives us one advantage over what we get from **constexpr** constructors: we can use user-defined literals in the middle of expressions where the **{}** notation can only be used when qualified by a type name. The example above is roughly equivalent to:

```
complex z1 {1.2,12e3};

complex f(double d)
{
        complex x {0,2.3};
        return x+sqrt(complex{d,12e3})+complex{0,12e3};
}
```

I suspect that the choice of style of literal depends on your sense of aesthetics and the conventions of your field of work. The standard-library **complex** uses **constexpr** constructors rather than a user-defined literal.

18.3.5 Accessor Functions

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 and change the value of the real and imaginary parts:

```
class complex {
        double re, im;
public:
        constexpr double real() const { return re; }
        constexpr double imag() const { return im; }

        void real(double r) { re = r; }
        void imag(double i) { im = i; }
        // ...
};
```

I don't consider it a good idea to provide individual access to all members of a class; in general, it is not. For many types, individual access (sometimes referred to as *get-and-set functions*) is an invitation to disaster. If we are not careful, individual access could compromise an invariant, and it typically complicates changes to the representation. For example, consider the opportunities for misuse from providing getters and setters for every member of the **Date** from §16.3 or (even more so) for the **String** from §19.3. However, for **complex**, **real()** and **imag()** are semantically significant: some algorithms are most cleanly written if they can set the real and imaginary parts independently.

For example, given **real()** and **imag()**, we can simplify simple, common, and useful operations, such as **==**, as nonmember functions (without compromising performance):

```
inline bool operator==(complex a, complex b)
{
        return a.real()==b.real() && a.imag()==b.imag();
}
```

18.3.6 Helper Functions

If we put all the bits and pieces together, the **complex** class becomes:

```
class complex {
        double re, im;
public:
        constexpr complex(double r =0, double i =0) : re(r), im(i) { }

        constexpr double real() const { return re; }
        constexpr double imag() const { return im; }

        void real(double r) { re = r; }
        void imag(double i) { im = i; }

        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);

// binary -, *, 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 definitions of most of the following helper functions similarly rely 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 (§5.6.2, §40.4).

18.4 Type Conversion

Type conversion can be accomplished by

- A constructor taking a single argument (§16.2.5)
- A conversion operator (§18.4.1)

In either case the conversion can be

- **explicit**; that is, the conversion is only performed in a direct initialization (§16.2.6), i.e., as an initializer not using a **=**.
- Implicit; that is, it will be applied wherever it can be used unambiguously (§18.4.3), e.g., as a function argument.

18.4.1 Conversion Operators

Using a constructor taking a single argument to specify type conversion is convenient but has implications that can be undesirable. Also, a constructor cannot specify

[1] an implicit conversion from a user-defined type to a built-in type (because the built-in 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, we could define a 6-bit non-negative integer, **Tiny**, that can mix freely with integers in arithmetic operations. **Tiny** throws **Bad_range** if its operations overflow or underflow:

```
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 §38.4.4). 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.

18.4.2 `explicit` Conversion Operators

Conversion operators tend to be defined so that they can be used everywhere. However, it is possible to declare a conversion operator **explicit** and have it apply only for direct initialization (§16.2.6), where an equivalent **explicit** constructor would have been used. For example, the standard-library **unique_ptr** (§5.2.1, §34.3.1) has an explicit conversion to **bool**:

```
template <typename T, typename D = default_delete<T>>
class unique_ptr {
public:
        // ...
        explicit operator bool() const noexcept;        // does *this hold a pointer (that is not nullptr)?
        // ...
};
```

The reason to declare this conversion operator **explicit** is to avoid its use in surprising contexts. Consider:

```
void use(unique_ptr<Record> p, unique_ptr<int> q)
{
        if (!p)                 // OK: we want this use
                throw Invalid_uninque_ptr{};

        bool b = p;       // error; suspicious use
        int x = p+q;      // error; we definitly don't want this
}
```

Had **unique_ptr**'s conversion to bool not been **explicit**, the last two definitions would have compiled. The value of **b** would have become **true** and the value of **x** would have become **1** or **2** (depending on whether **q** was valid or not).

18.4.3 Ambiguities

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(const 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 a call cannot be resolved without them (i.e., using only built-in conversions). 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 (§12.3).

The rules for conversion are neither the simplest to implement, nor the simplest to document, nor the most general that could be devised. They are, however, considerably safer, and the resulting resolutions are typically less surprising than alternatives. 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 floating-point 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.

18.5 Advice

[1] Define operators primarily to mimic conventional usage; §18.1.
[2] Redefine or prohibit copying if the default is not appropriate for a type; §18.2.2.
[3] For large operands, use **const** reference argument types; §18.2.4.
[4] For large results, use a move constructor; §18.2.4.

[5] Prefer member functions over nonmembers for operations that need access to the representation; §18.3.1.
[6] Prefer nonmember functions over members for operations that do not need access to the representation; §18.3.2.
[7] Use namespaces to associate helper functions with ‘‘their’’ class; §18.2.5.
[8] Use nonmember functions for symmetric operators; §18.3.2.
[9] Use member functions to express operators that require an lvalue as their left-hand operand; §18.3.3.1.
[10] Use user-defined literals to mimic conventional notation; §18.3.4.
[11] Provide ‘‘**set()** and **get()** functions’’ for a data member only if the fundamental semantics of a class require them; §18.3.5.
[12] Be cautious about introducing implicit conversions; §18.4.
[13] Avoid value-destroying (‘‘narrowing’’) conversions; §18.4.1.
[14] Do not define the same conversion as both a constructor and a conversion operator; §18.4.3.

19

Special Operators

We are all special cases.
– Albert Camus

- Introduction
- Special Operators
  Subscripting; Function Call; Dereferencing; Increment and Decrement; Allocation and Deallocation; User-Defined Literals
- A String Class
  Essential Operations; Access to Characters; Representation; Member Functions; Helper Functions; Using Our String
- Friends
  Finding Friends; Friends and Members
- Advice

19.1 Introduction

Overloading is not just for arithmetic and logical operations. In fact, operators are crucial in the design of containers (e.g., **vector** and **map**; §4.4), ''smart pointers'' (e.g., **unique_ptr** and **shared_ptr**; §5.2.1), iterators (§4.5), and other classes concerned with resource management.

19.2 Special Operators

The operators

[] () -> ++ -- new delete

are special only in that the mapping from their use in the code to a programmer's definition differs slightly from that used for conventional unary and binary operators, such as **+**, **<**, and **~** (§18.2.3). The **[]** (subscript) and **()** (call) operators are among the most useful user-defined operators.

19.2.1 Subscripting

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, we can define a simple associative array type like this:

```
struct Assoc {
        vector<pair<string,int>> vec;  // vector of {name,value} pairs

        const int& operator[] (const string&) const;
        int& operator[](const string&);
};
```

An **Assoc** keeps a vector of **std::pair**s. The implementation uses the same trivial and inefficient search method as in §7.7:

```
int& Assoc::operator[](const string& s)
        // search for s; return a reference to its value if found;
        // otherwise, make a new pair {s,0} and return a reference to its value
{
        for (auto x : vec)
                if (s == x.first) return x.second;

        vec.push_back({s,0});           // initial value: 0

        return vec.back().second;       // return last element (§31.2.2)
}
```

We can use **Assoc** like this:

```
int main()      // count the occurrences of each word on input
{
        Assoc values;
        string buf;
        while (cin>>buf) ++values[buf];
        for (auto x : values.vec)
                cout << '{' << x.first << ',' << x.second << "}\n";
}
```

The standard-library **map** and **unordered_map** are further developments of the idea of an associative array (§4.4.3, §31.4.3) with less naive implementations.

An **operator[]()** must be a non-**static** member function.

19.2.2 Function 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. For example:

```
struct Action {
        int operator()(int);
        pair<int,int> operator()(int,int);
        double operator()(double);
        // ...
};

void f(Action act)
{
        int x = act(2);
        auto y = act(3,4);
        double z = act(2.3);
        // ...
};
```

An argument list for an `operator()()` is evaluated and checked according to the usual argument-passing rules. Overloading the function call operator seems to be useful primarily for defining types that have only a single operation and for types for which one operation is predominant. The *call operator* is also known as the *application operator*.

The most obvious and 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* (§3.4.3). Such function objects allow us to write code that takes nontrivial operations as parameters. In many cases, it is essential that function objects can hold data needed to perform their operation. For example, we can define a class with an `operator()()` that adds a stored value to its argument:

```
class Add {
        complex val;
public:
        Add(complex c) :val{c} { }                          // save a value
        Add(double r, double i) :val{{r,i}} { }

        void operator()(complex& c) const { c += val; }     // add a 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>& vec, list<complex>& lst, complex z)
{
        for_each(vec.begin(),vec.end(),Add{2,3});
        for_each(lst.begin(),lst.end(),Add{z});
}
```

This will add `complex{2,3}` to every element of the `vector` and `z` to every element of the `list`. Note that `Add{z}` constructs an object that is used repeatedly by `for_each()`: `Add{z}`'s `operator()()` is called for each element of the sequence.

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<typename Iter, typename Fct>
Fct for_each(Iter b, Iter e, Fct f)
{
        while (b != e) f(*b++);
        return f;
}
```

At first glance, this technique may look esoteric, but it is simple, efficient, and extremely useful (§3.4.3, §33.4).

Note that a lambda expression (§3.4.3, §11.4) is basically a syntax for defining a function object. For example, we could have written:

```
void h2(vector<complex>& vec, list<complex>& lst, complex z)
{
        for_each(vec.begin(),vec.end(),[](complex& a){ a+={2,3}; });
        for_each(lst.begin(),lst.end(),[](complex& a){ a+=z; });
}
```

In this case, each of the lambda expressions generates the equivalent of the function object **Add**.

Other popular uses of **operator()()** are as a substring operator and as a subscripting operator for multidimensional arrays (§29.2.2, §40.5.2).

An **operator()()** must be a non-**static** member function.

Function call operators are often templates (§29.2.2, §33.5.3).

19.2.3 Dereferencing

The dereferencing operator, **–>** (also known as the *arrow* operator), can be defined as a unary postfix operator. For example:

```
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. The standard-library ''smart pointers'' **unique_ptr** and **shared_ptr** (§5.2.1) provide operator **–>**.

As an example, we could define a class **Disk_ptr** for accessing objects stored on disk. **Disk_ptr**'s constructor takes a name that can be used to find the object on disk, **Disk_ptr::operator–>()** brings the object into main memory when accessed through its **Disk_ptr**, and **Disk_ptr**'s destructor eventually writes the updated object back out to disk:

```
template<typename T>
class Disk_ptr {
        string identifier;
        T* in_core_address;
        // ...
public:
        Disk_ptr(const string& s) : identifier{s}, in_core_address{nullptr} { }
        ~Disk_ptr() { write_to_disk(in_core_address,identifier); }

        T* operator–>()
        {
                if (in_core_address == nullptr)
                        in_core_address = read_from_disk(identifier);
                return in_core_address;
        }
};
```

Disk_ptr might be used like this:

```
struct Rec {
        string name;
        // ...
};

void update(const string& s)
{
        Disk_ptr<Rec> p {s};            // get Disk_ptr for s

        p–>name = "Roscoe";             // update s; if necessary, first retrieve from disk
        // ...
}                                       // p's destructor writes back to disk
```

Naturally, 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 a class **Y** for which **–>**, *, and **[]** have their default meaning and a **Y*** called **p**, then:

```
p–>m == (*p).m          // is true
(*p).m == p[0].m        // is true
p–>m == p[0].m          // is true
```

As usual, no such guarantee is provided for user-defined operators. The equivalence can be provided where desired:

```
template<typename T>
class Ptr {
        Y* p;
public:
        Y* operator->() { return p; }          // dereference to access member
        Y& operator*() { return *p; }          // dereference to access whole object
        Y& operator[](int i) { return p[i]; }  // dereference to access element
        // ...
};
```

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 `X` if `++`, `+=`, `=`, and `+` are provided.

The overloading of `->` is important to a class of interesting programs and is 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 33) provide an important example of this.

Operator `->` must be a non-`static` member function. If used, its return type must be a pointer or an object of a class to which you can apply `->`. The body of a template class member function is only checked if the function is used (§26.2.1), so we can define `operator->()` without worrying about types, such as `Ptr<int>`, for which `->` does not make sense.

Despite the similarity between `->` and `.` (dot), there is no way of overloading operator `.` (dot).

19.2.4 Increment and Decrement

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(X a)            // traditional use
{
        X v[200];
        X* p = &v[0];
        p--;
        *p = a;     // oops: p out of range, uncaught
        ++p;
        *p = a;     // OK
}
```

Here, we might want to replace the `X*` with an object of a class `Ptr<X>` that can be dereferenced only if it actually points to an `X`. We would also like to ensure that `p` can be incremented and decremented only if it points to an object within an array and the increment and decrement operations yield an object within that array. That is, we would like something like this:

```
void f2(Ptr<X> a)          // checked
{
    X v[200];
    Ptr<X> p(&v[0],v);
    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 for `Ptr<T>`. For example:

```
template<typename T>
class Ptr {
    T* ptr;
    T* array;
    int sz;
public:
    template<int N>
        Ptr(T* p, T(&a)[N]);      // bind to array a, sz==N, initial value p
    Ptr(T* p, T* a, int s);       // bind to array a of size s, initial value p
    Ptr(T* p);                    // bind to single object, sz==0, initial value p

    Ptr& operator++();            // prefix
    Ptr operator++(int);          // postfix

    Ptr& operator−−();            // prefix
    Ptr 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 `−−`.

Consider omitting postfix `++` and `−−` in a design. They are not only odd syntactically, they tend to be marginally harder to implement than the postfix versions, less efficient, and less frequently used. For example:

```
template<typename T>
Ptr& Ptr<T>::operator++()          // return the current object after incrementing
{
    // ... check that ptr+1 can be pointed to ...
    return *++ptr;
}
```

```
template<typename T>
Ptr Ptr<T>::operator++(int)          // increment and return a Ptr with the old value
{
    // ... check that ptr+1 can be pointed to ...
    Ptr<T> old {ptr,array,sz};
    ++ptr;
    return old;
}
```

The pre-increment operator can return a reference to its object. The post-increment operator must make a new object to return.

Using **Ptr**, the example is equivalent to:

```
void f3(T a)          // checked
{
    T v[200];
    Ptr<T> p(&v[0],v,200);
    p.operator--(0);          // suffix: p--
    p.operator*() = a;        // run-time error: p out of range
    p.operator++();           // prefix: ++p
    p.operator*() = a;        // OK
}
```

Completing class **Ptr** is left as an exercise. A pointer template that behaves correctly with respect to inheritance is presented in §27.2.2.

19.2.5 Allocation and Deallocation

Operator **new** (§11.2.3) acquires its memory by calling an **operator new()**. Similarly, operator **delete** frees its memory by calling an **operator delete()**. A user can redefine the global **operator new()** and **operator delete()** or define **operator new()** and **operator delete()** for a particular class.

Using the standard-library type alias **size_t** (§6.2.8) for sizes, the declarations of the global versions look like this:

```
void* operator new(size_t);              // use for individual object
void* operator new[](size_t);            // use for array
void operator delete(void*, size_t);     // use for individual object
void operator delete[](void*, size_t);   // use for array

// for more versions, see §11.2.4
```

That is, when **new** needs memory on the free store for an object of type **X**, it calls **operator new(sizeof(X))**. Similarly, when **new** needs memory on the free store for an array of **N** objects of type **X**, it calls **operator new[](N*sizeof(X))**. A **new** expression may ask for more memory than is indicated by **N*sizeof(X)**, but it will always do so in terms of a number of characters (i.e., a number of bytes). Replacing the global **operator new()** and **operator delete()** is not for the faint-hearted and not recommended. 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 a class **Employee** 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);

    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)
{
    if (p) {    // delete only if p!=0; see §11.2, §11.2.3
        // 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 being **delete**d. Deleting a ''plain'' **Employee** gives an argument value of **sizeof(Employee)**; deleting a **Manager** derived from **Employee** that does not have its own **operator delete()** 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 (as a general-purpose allocator must) and ignore the **size_t** argument to **operator delete()**. However, doing so 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()**? The type specified in the **delete** operation matches the type of the object being **delete**d. If we **delete** an object through a pointer to a base class, that base class must have a **virtual** destructor (§17.2.5) for the correct size to be given:

```
Employee* p = new Manager;  // potential trouble (the exact type is lost)
// ...
delete p;                   // hope Employee has a virtual destructor
```

In principle, deallocation is then done by the destructor (which knows the size of its class).

19.2.6 User-defined Literals

C++ provides literals for a variety of built-in types (§6.2.6):

```
123         // int
1.2         // double
1.2F        // float
'a'         // char
1ULL        // unsigned long long
0xD0        // hexadecimal unsigned
"as"        // C-style string (const char[3])
```

In addition, we can define literals for user-defined types and new forms of literals for built-in types. For example:

```
"Hi!"s                  // string, not "zero-terminated array of char"
1.2i                    // imaginary
101010111000101b        // binary
123s                    // seconds
123.56km                // not miles! (units)
12345678901234567890123456789012345678901234567890x   // extended-precision
```

Such *user-defined literals* are supported through the notion of *literal operators* that map literals with a given suffix into a desired type. The name of a literal operator is **operator""** followed by the suffix. For example:

```
constexpr complex<double> operator"" i(long double d)      // imaginary literal
{
    return {0,d};    // complex is a literal type
}

std::string operator"" s(const char* p, size_t n)     // std::string literal
{
    return string{p,n};   // requires free-store allocation
}
```

These two operators define suffixes **i** and **s**, respectively. I use **constexpr** to enable compile-time evaluation. Given those, we can write:

```
template<typename T> void f(const T&);

void g()
{
    f("Hello");       // pass pointer to char*
    f("Hello"s);      // pass (five-character) string object
    f("Hello\n"s);    // pass (six-character) string object

    auto z = 2+1i;    // complex{2,1}
}
```

The basic (implementation) idea is that after parsing what could be a literal, the compiler always checks for a suffix. The user-defined literal mechanism simply allows the user to specify a new

suffix and define what is to be done with the literal before it. It is not possible to redefine the meaning of a built-in literal suffix or to augment the syntax of literals.

There are four kinds of literals that can be suffixed to make a user-defined literal (§iso.2.14.8):

- An integer literal (§6.2.4.1): accepted by a literal operator taking an **unsigned long long** or a **const char*** argument or by a template literal operator, for example, **123m** or **12345678901234567890X**
- A floating-point literal (§6.2.5.1): accepted by a literal operator taking a **long double** or a **const char*** argument or by a template literal operator, for example, **12345678901234567890.976543210x** or **3.99s**
- A string literal (§7.3.2): accepted by a literal operator taking a (**const char***, **size_t**) pair of arguments, for example, **"string"s** and **R"(Foo\bar)"_path**
- A character literal (§6.2.3.2): accepted by a literal operator taking a character argument of type **char**, **wchar_t**, **char16_t**, or **char32_t**, for example, **'f'_runic** or **u'BEEF'_w**.

For example, we could define a literal operator to collect digits for integer values that cannot be represented in any of the built-in integer types:

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);

f(12345678901234567890123456789012345678901234567890123456789012345x);
```

Here, the C-style string **"12345678901234567890123456789012345678901234567890123456789012345"** is passed to **operator"" x()**. Note that I did not put those digits in double quotes. I requested a C-style string for my operator, and the compiler delivered it from the digits provided.

To get a C-style string from the program source text into a literal operator, we request both the string and its number of characters. For example:

```
string operator"" s(const char* p, size_t n);

string s12 = "one two"s;      // calls operator ""("one two",7)
string s22 = "two\ntwo"s;     // calls operator ""("two\ntwo",7)
string sxx = R"(two\ntwo)"s;  // calls operator ""("two\\ntwo",8)
```

In the raw string (§7.3.2.1), **"\n"** represents the two characters **'\'** and **'n'**.

The rationale for requiring the number of characters is that if we want to have ‘‘a different kind of string,’’ we almost always want to know the number of characters anyway.

A literal operator that takes just a **const char*** argument (and no size) can be applied to integer and floating-point literals. For example:

```
string operator"" SS(const char* p);        // warning: this will not work as expected

string s12 = "one two"SS;                   // error: no applicable literal operator
string s13 = 13SS;                          // OK, but why would anyone do that?
```

A literal operator converting numerical values to strings could be quite confusing.

A *template literal operator* is a literal operator that takes its argument as a template parameter pack, rather than as a function argument. For example:

```
template<char...>
constexpr int operator"" _b3();          // base 3, i.e., ternary
```

Given that, we get:

```
201_b3    // means operator"" b3<'2','0','1'>(); so 9*2+0*3+1 == 19
241_b3    // means operator"" b3<'2','4','1'>(); so error: 4 isn't a ternary digit
```

The variadic template techniques (§28.6) can be disconcerting, but it is the only way of assigning nonstandard meanings to digits at compile time.

To define **operator"" _b3()**, we need some helper functions:

```
constexpr int ipow(int x, int n) // x to the nth power for n>=0
{
    return (n>0) ? x*ipow(n–1) : 1;
}

template<char c>          // handle the single ternary digit case
constexpr int b3_helper()
{
    static_assert(c<'3',"not a ternary digit");
    return c;
}

template<char c, char... tail>   // peel off one ternary digit
constexpr int b3_helper()
{
    static_assert(c<'3',"not a ternary digit");
    return ipow(3,sizeof...(tail))*(c–'0')+b3_helper(tail...);
}
```

Given that, we can define our base 3 literal operator:

```
template<char... chars>
constexpr int operator"" _b3()       // base 3, i.e., ternary
{
    return b3_helper(chars...);
}
```

Many suffixes will be short (e.g., **s** for **std::string**, **i** for imaginary, **m** for meter (§28.7.3), and **x** for extended), so different uses could easily clash. Use namespaces to prevent clashes:

```
namespace Numerics {
    // ...

    class Bignum { /* ... */ };

    namespace literals {
        Bignum operator"" x(char const*);
    }
```

```
        // ...
}

using namespace Numerics::literals;
```

The standard library reserves all suffixes not starting with an initial underscore, so define your suffixes starting with an underscore or risk your code breaking in the future:

```
123km       // reserved by the standard library
123_km      // available for your use
```

19.3 A String Class

The relatively simple string class presented in this section illustrates several techniques that are useful for the design and implementation of classes using conventionally defined operators. This **String** is a simplified version of the standard-library **string** (§4.2, Chapter 36). **String** provides value semantics, checked and unchecked access to characters, stream I/O, support for range-**for** loops, equality operations, and concatenation operators. I also added a **String** literal, which **std::string** does not (yet) have.

To allow simple interoperability with C-style strings (including string literals (§7.3.2)), I represent strings as zero-terminated arrays of characters. For realism, I implement the *short string optimization*. That is, a **String** with only a few characters stores those characters in the class object itself, rather than on the free store. This optimizes string usage for small strings. Experience shows that for a huge number of applications most strings are short. This optimization is particularly important in multi-threaded systems where sharing through pointers (or references) is infeasible and free-store allocation and deallocation relatively expensive.

To allow **String**s to efficiently ''grow'' by adding characters at the end, I implement a scheme for keeping extra space for such growth similar to the one used for **vector** (§13.6.1). This makes **String** a suitable target for various forms of input.

Writing a better string class and/or one that provides more facilities is a good exercise. That done, we can throw away our exercises and use **std::string** (Chapter 36).

19.3.1 Essential Operations

Class **String** provides the usual set of constructors, a destructor, and assignment operations (§17.1):

```
class String {
public:
        String();                               // default constructor: x{""}

        explicit String(const char* p);         // constructor from C-style string: x{"Euler"}

        String(const String&);                  // copy constructor
        String& operator=(const String&);       // copy assignment
```

```
        String(String&& x);                     // move constructor
        String& operator=(String&& x);          // move assignment

        ~String() { if (short_max<sz) delete[] ptr; }   // destructor

        // ...
};
```

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 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**. Where it makes sense, I prefer value semantics; examples are **complex**, **vector**, **Matrix**, and **string**. However, for value semantics to be affordable, we need to pass **String**s by reference when we don't need copies and to implement move semantics (§3.3.2, §17.5.2) to optimize **return**s.

The slightly nontrivial representation of **String** is presented in §19.3.3. Note that it requires user-defined versions of the copy and move operations.

19.3.2 Access to Characters

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. Here, I follow the standard library by providing efficient unchecked operations with the conventional **[]** subscript notation plus range-checked **at()** operations:

```
class String {
public:
        // ...

        char& operator[](int n) { return ptr[n]; }          // unchecked element access
        char operator[](int n) const { return ptr[n]; }

        char& at(int n) { check(n); return ptr[n]; }        // range-checked element access
        char at(int n) const { check(n); return ptr[n]; }

        String& operator+=(char c);                         // add c at end

        const char* c_str() { return ptr; }                 // C-style string access
        const char* c_str() const { return ptr; }

        int size() const { return sz; }                     // number of elements
        int capacity() const                                // elements plus available space
              { return (sz<=short_max) ? short_max : sz+space; }

        // ...
};
```

The idea is to use **[]** for ordinary use. For example:

```
int hash(const String& s)
{
    int h {s[0]};
    for (int i {1}; i!=s.size(); i++) h ^= s[i]>>1;        // unchecked access to s
    return h;
}
```

Here, using the checked **at()** would be redundant because we correctly access **s** only from **0** to **s.size()–1**.

We can use **at()** where we see a possibility of mistakes. For example:

```
void print_in_order(const String& s,const vector<int>& index)
{
    for (x : index) cout << s.at(x) << '\n';
}
```

Unfortunately, assuming that people will use **at()** consistently where mistakes can be made is overly optimistic, so some implementations of **std::string** (from which the **[]**/**at()** convention is borrowed) also check **[]**. I personally prefer a checked **[]** at least during development. However, for serious string manipulation tasks, a range check on each character access could impose quite noticeable overhead.

I provide **const** and non-**const** versions of the access functions to allow them to be used for **const** as well as other objects.

19.3.3 Representation

The representation for **String** was chosen to meet three goals:

- To make it easy to convert a C-style string (e.g., a string literal) to a **String** and to allow easy access to the characters of a **String** as a C-style string
- To minimize the use of the free store
- To make adding characters to the end of a **String** efficient

The result is clearly messier than a simple {pointer,size} representation, but much more realistic:

```
class String {
/*
    A simple string that implements the short string optimization

    size()==sz is the number of elements
    if size()<= short_max, the characters are held in the String object itself;
    otherwise the free store is used.

    ptr points to the start of the character sequence
    the character sequence is kept zero-terminated: ptr[size()]==0;
    this allows us to use C library string functions and to easily return a C-style string: c_str()

    To allow efficient addition of characters at end, String grows by doubling its allocation;
    capacity() is the amount of space available for characters
    (excluding the terminating 0): sz+space
*/
```

```
public:
    // ...
private:
    static const int short_max = 15;
    int sz;                         // number of characters
    char* ptr;
    union {
        int space;                  // unused allocated space
        char ch[short_max+1];       // leave space for terminating 0
    };

    void check(int n) const         // range check
    {
        if (n<0 || sz<=n)
            throw std::out_of_range("String::at()");
    }

    // ancillary member functions:
    void copy_from(const String& x);
    void move_from(String& x);
};
```

This supports what is known as the *short string optimization* by using two string representations:

- If `sz<=short_max`, the characters are stored in the `String` object itself, in the array named `ch`.
- If `!(sz<=short_max)`, the characters are stored on the free store and we may allocate extra space for expansion. The member named `space` is the number of such characters.

In both cases, the number of elements is kept in `sz` and we look at `sz`, to determine which implementation scheme is used for a given string.

In both cases, `ptr` points to the elements. This is essential for performance: the access functions do not need to test which representation is used; they simply use `ptr`. Only the constructors, assignments, moves, and the destructor (§19.3.4) must care about the two alternatives.

We use the array `ch` only if `sz<=short_max` and the integer `space` only if `!(sz<=short_max)`. Consequently, it would be a waste to allocate space for both `ch` and `space` in a `String` object. To avoid such waste, I use a `union` (§8.3). In particular, I used a form of `union` called an *anonymous union* (§8.3.2), which is specifically designed to allow a class to manage alternative representations of objects. All members of an anonymous union are allocated in the same memory, starting at the same address. Only one member may be used at any one time, but otherwise they are accessed and used exactly as if they were separate members of the scope surrounding the anonymous union. It is the programmer's job to make sure that they are never misused. For example, all member functions of `String` that use `space` must make sure that it really was `space` that was set and not `ch`. That is done by looking at `sz<=short_max`. In other words, `Shape` is (among other things) a discriminated union with `sz<=short_max` as the discriminant.

19.3.3.1 Ancillary Functions

In addition to functions intended for general use, I found that my code became cleaner when I provided three ancillary functions as ''building blocks'' to help me with the somewhat tricky representation and to minimize code replication. Two of those need to access the representation of **String**, so I made them members. However, I made them **private** members because they don't represent operations that are generally useful and safe to use. For many interesting classes, the implementation is not just the representation plus the **public** functions. Ancillary functions can lead to less duplication of code, better design, and improved maintainability.

The first such function moves characters into newly allocated memory:

```
char* expand(const char* ptr, int n)      // expand into free store
{
    char* p = new char[n];
    strcpy(p,ptr);              // §43.4
    return p;
}
```

This function does not access the **String** representation, so I did not make it a member.

The second implementation function is used by copy operations to give a **String** a copy of the members of another:

```
void String::copy_from(const String& x)
    // make *this a copy of x
{
    if (x.sz<=short_max) {              // copy *this
        memcpy(this,&x,sizeof(x));      // §43.5
        ptr = ch;
    }
    else {                              // copy the elements
        ptr = expand(x.ptr,x.sz+1);
        sz = x.sz;
        space = 0;
    }
}
```

Any necessary cleanup of the target **String** is the task of callers of **copy_from()**; **copy_from()** unconditionally overwrites its target. I use the standard-library **memcpy()** (§43.5) to copy the bytes of the source into the target. That's a low-level and sometimes pretty nasty function. It should be used only where there are no objects with constructors or destructors in the copied memory because **memcpy()** knows nothing about types. Both **String** copy operations use **copy_from()**.

The corresponding function for move operations is:

```
void String::move_from(String& x)
{
    if (x.sz<=short_max) {              // copy *this
        memcpy(this,&x,sizeof(x));      // §43.5
        ptr = ch;
    }
```

```
        else {                      // grab the elements
            ptr = x.ptr;
            sz = x.sz;
            space = x.space;
            x.ptr = x.ch;           // x = ""
            x.sz = 0;
            x.ch[0]=0;
        }
    }
```

It too unconditionally makes its target a copy of its argument. However, it does not leave its argument owning any free store. I could also have used **memcpy()** in the long string case, but since a long string representation uses only part of **String**'s representation, I decided to copy the used members individually.

19.3.4 Member Functions

The default constructor defines a **String** to be empty:

```
String::String()                // default constructor: x{""}
        : sz{0}, ptr{ch}        // ptr points to elements, ch is an initial location (§19.3.3)
    {
        ch[0] = 0;              // terminating 0
    }
```

Given **copy_from()** and **move_from()**, the constructors, moves, and assignments are fairly simple to implement. The constructor that takes a C-style string argument must determine the number of characters and store them appropriately:

```
String::String(const char* p)
    :sz{strlen(p)},
     ptr{(sz<=short_max) ? ch : new char[sz+1]},
     space{0}
{
    strcpy(ptr,p);   // copy characters into ptr from p
}
```

If the argument is a short string, **ptr** is set to point to **ch**; otherwise, space is allocated on the free store. In either case, the characters are copied from the argument string into the memory managed by **String**.

The copy constructor simply copies the representation of its arguments:

```
String::String(const String& x)     // copy constructor
{
    copy_from(x);  // copy representation from x
}
```

I didn't bother trying to optimize the case where the size of the source equals the size of the target (as was done for **vector**; §13.6.3). I don't know if that would be worthwhile.

Similarly, the move constructor moves the representation from its source (and possibly sets it argument to be the empty string):

```
String::String(String&& x)      // move constructor
{
    move_from(x);
}
```

Like the copy constructor, the copy assignment uses **copy_from()** to clone its argument's representation. In addition, it has to **delete** any free store owned by the target and make sure it does not get into trouble with self-assignment (e.g., **s=s**):

```
String& String::operator=(const String& x)
{
    if (this==&x) return *this;             // deal with self-assignment
    char* p = (short_max<sz) ? ptr : 0;
    copy_from(x);
    delete[] p;
    return *this;
}
```

The **String** move assignment deletes its target's free store (if there is any) and then moves:

```
String& String::operator=(String&& x)
{
    if (this==&x) return *this;             // deal with self-assignment (x = move(x) is insanity)
    if (short_max<sz) delete[] ptr;         // delete target
    move_from(x);                           // does not throw
    return *this;
}
```

It is logically possible to move a source into itself (e.g., **s=std::move(s)**), so again we have to protect against self-assignment (however unlikely).

The logically most complicated **String** operation is **+=**, which adds a character to the end of the string, increasing its size by one:

```
String& String::operator+=(char c)
{
    if (sz==short_max) {        // expand to long string
        int n = sz+sz+2;        // double the allocation (+2 because of the terminating 0)
        ptr = expand(ptr,n);
        space = n–sz–2;
    }
    else if (short_max<sz) {
        if (space==0) {             // expand in free store
            int n = sz+sz+2;        // double the allocation (+2 because of the terminating 0)
            char* p = expand(ptr,n);
            delete[] ptr;
            ptr = p;
            space = n–sz–2;
        }
        else
            ––space;
    }
```

```
        ptr[sz] = c;            // add c at end
        ptr[++sz] = 0;          // increase size and set terminator

        return *this;
    }
```

There is a lot going on here: `operator+=()` has to keep track of which representation (short or long) is used and whether there is extra space available to expand into. If more space is needed, `expand()` is called to allocate that space and move the old characters into the new space. If there was an old allocation that needs deleting, it is returned, so that `+=` can delete it. Once enough space is available, it is trivial to put the new character `c` into it and to add the terminating `0`.

Note the calculation of available memory for `space`. Of all the `String` implementation that took the longest to get right: its a messy little calculation prone to off-by-one errors. That repeated constant `2` feels awfully like a "magic constant."

All `String` members take care not to modify a new representation before they are certain that a new one can be put in place. In particular, they don't `delete` until after any possible `new` operations have been done. In fact, the `String` members provide the strong exception guarantee (§13.2).

If you don't like the kind of fiddly code presented as part of the implementation of `String`, simply use `std::string`. To a large extent, the standard-library facilities exist to save us from programming at this low level most of the time. Stronger: writing a string class, a vector class, or a map is an excellent exercise. However, once the exercise is done, one outcome should be an appreciation of what the standard offers and a desire not to maintain your own version.

19.3.5 Helper Functions

To complete class `String`, I provide a set of useful functions, stream I/O, support for range-`for` loops, comparison, and concatenation. These all mirror the design choices used for `std::string`. In particular, `<<` just prints the characters without added formatting, and `>>` skips initial whitespace before reading until it finds terminating whitespace (or the end of the stream):

```
ostream& operator<<(ostream& os, const String& s)
{
    return os << s.c_str();     // §36.3.3
}

istream& operator>>(istream& is, String& s)
{
    s = "";     // clear the target string
    is>>ws;     // skip whitespace (§38.4.5.1)
    char ch = ' ';
    while(is.get(ch) && !isspace(ch))
        s += ch;
    return is;
}
```

I provide `==` and `!=` for comparison:

```
bool operator==(const String& a, const String& 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;
}

bool operator!=(const String& a, const String& b)
{
    return !(a==b);
}
```

Adding `<`, etc., would be trivial.

To support the range-`for` loop, we need `begin()` and `end()` (§9.5.1). Again, we can provide those as freestanding (nonmember) functions without direct access to the `String` implementation:

```
char* begin(String& x)          // C-string-style access
{
    return x.c_str();
}

char* end(String& x)
{
    return x.c_str()+x.size();
}

const char* begin(const String& x)
{
    return x.c_str();
}

const char* end(const String& x)
{
    return x.c_str()+x.size();
}
```

Given the member function `+=` that adds a character at the end, concatenation operators are easily provided as nonmember functions:

```
String& operator+=(String& a, const String& b)      // concatenation
{
    for (auto x : b)
        a+=x;
    return a;
}
```

```
String operator+(const String& a, const String& b) // concatenation
{
        String res {a};
        res += b;
        return res;
}
```

I feel that I may have slightly ''cheated'' here. Should I have provided a member `+=` that added a C-style string to the end? The standard-library `string` does, but without it, concatenation with a C-style string still works. For example:

```
String s = "Njal ";
s += "Gunnar";      // concatenate: add to the end of s
```

This use of `+=` is interpreted as `operator+=(s,String("Gunnar"))`. My guess is that I could provide a more efficient `String::operator+=(const char*)`, but I have no idea if the added performance would be worthwhile in real-world code. In such cases, I try to be conservative and deliver the minimal design. Being able to do something is not by itself a good reason for doing it.

Similarly, I do not try to optimize `+=` by taking the size of a source string into account.

Adding `_s` as a string literal suffix meaning `String` is trivial:

```
String operator"" _s(const char* p, size_t)
{
        return String{p};
}
```

We can now write:

```
void f(const char*);        // C-style string
void f(const String&);      // our string

void g()
{
        f("Madden's");              // f(const char*)
        f("Christopher's"_s);       // f(const String&);
}
```

19.3.6 Using Our String

The main program simply exercises the `String` operators a bit:

```
int main()
{
        String s ("abcdefghij");
        cout << s << '\n';
        s += 'k';
        s += 'l';
        s += 'm';
        s += 'n';
        cout << s << '\n';
```

```
        String s2 = "Hell";
        s2 += " and high water";
        cout << s2 << '\n';

        String s3 = "qwerty";
        s3 = s3;
        String s4 ="the quick brown fox jumped over the lazy dog";
        s4 = s4;
        cout << s3 << " " << s4 << "\n";
        cout << s + ". " + s3 + String(". ") + "Horsefeathers\n";

        String buf;
        while (cin>>buf && buf!="quit")
            cout << buf << " " << buf.size() << " " << buf.capacity() << '\n';
}
```

This `String` lacks many features that you might consider important or even essential. However, for what it does it closely resembles `std::string` (Chapter 36) and illustrates techniques used for the implementation of the standard-library `string`.

19.4 Friends

An ordinary member function declaration specifies three logically distinct things:

[1] The function can access the private part of the class declaration.
[2] The function is in the scope of the class.
[3] The function must be invoked on an object (has a `this` pointer).

By declaring a member function `static` (§16.2.12), we can give it the first two properties only. By declaring a nonmember function a `friend`, we can give it the first property only. That is, a function declared `friend` is granted access to the implementation of a class just like a member function but is otherwise independent of that class.

For example, we could define an operator that multiplies a `Matrix` by a `Vector`. Naturally, `Vector` and `Matrix` hide their respective representations 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:

```
constexpr rc_max {4};    // row and column size

class Matrix;

class Vector {
    float v[rc_max];
    // ...
    friend Vector operator*(const Matrix&, const Vector&);
};
```

```
class Matrix {
        Vector v[rc_max];
        // ...
        friend Vector operator*(const Matrix&, const Vector&);
};
```

Now **operator*()** can reach into the implementation of both **Vector** and **Matrix**. That would allow sophisticated implementation techniques, but a simple implementation would be:

```
Vector operator*(const Matrix& m, const Vector& v)
{
        Vector r;
        for (int i = 0; i!=rc_max; i++) {          // r[i] = m[i] * v;
                r.v[i] = 0;
                for (int j = 0; j!=rc_max; 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();
        // ...
};
```

There is a shorthand for making all functions of one class friends of another. For example:

```
class List {
        friend class List_iterator;
        // ...
};
```

This **friend** declaration makes all of **List_iterator**'s member functions friends of **List**.

Declaring a class a **friend** grants access to every function of that class. That implies that we cannot know the set of functions that can access the granting class's representation just by looking at the class itself. In this, a friend class declaration differs from the declaration of a member function and a friend function. Clearly, friend classes should be used with caution and only to express closely connected concepts.

It is possible to make a template argument a **friend**:

```
template<typename T>
class X {
    friend T;
    friend class T;  // redundant "class"
    // ...
};
```

Often, there is a choice between making a class a member (a nested class) or a nonmember friend (§18.3.1).

19.4.1 Finding Friends

A friend must be previously declared in an enclosing scope or defined in the non-class scope immediately enclosing the class that is declaring it to be a **friend**. Scopes outside the innermost enclosing namespace scope are not considered for a name first declared as a **friend** (§iso.7.3.1.2). Consider a technical example:

```
class C1 { };    // will become friend of N::C
void f1();       // will become friend of N::C

namespace N {
    class C2 { };        // will become friend of C
    void f2() { }        // will become friend of C

    class C {
        int x;
    public:
        friend class C1;     // OK (previously defined)
        friend void f1();

        friend class C3;     // OK (defined in enclosing namespace)
        friend void f3();
        friend class C4;     // First declared in N and assumed to be in N
        friend void f4();
    };

    class C3 {};             // friend of C
    void f3() {  C x; x.x = 1; }  // OK: friend of C
} // namespace N

class C4 { };                // not friend of N::C
void f4() { N::C x; x.x = 1; }    // error: x is private and f4() is not a friend of N::C
```

A friend function can be found through its arguments (§14.2.4) even if it was not declared in the immediately enclosing scope. For example:

```
void f(Matrix& m)
{
    invert(m);       // Matrix's friend invert()
}
```

Thus, a friend function should be explicitly declared in an enclosing scope or take an argument of its class or a class derived from that. If not, the friend cannot be called. For example:

```
// no f() in this scope

class X {
        friend void f();              // useless
        friend void h(const X&);      // can be found through its argument
};

void g(const X& x)
{
        f();        // no f() in scope
        h(x);       // X's friend h()
}
```

19.4.2 Friends and 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 (§3.2.3, §17.2.5) – but typically there is a choice. Because member names are local to the class, a function that requires direct access to the representation should be a member unless there is a specific reason for it to be a nonmember.

Consider a class **X** supplying alternative ways of presenting an operation:

```
class X {
        // ...
        X(int);

        int m1();                     // member
        int m2() const;

        friend int f1(X&);            // friend, not member
        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 to the leftmost operand of a **.** or **->** (but see §19.2.3). For example:

```
void g()
{
        99.m1();   // error: X(99).m1() not tried
        99.m2();   // error: X(99).m2() not tried
}
```

The global function **f1()** has a similar property because implicit conversions are not used for non-**const** reference arguments (§7.7). However, conversions may be applied to the arguments of **f2()** and **f3()**:

```
void h()
{
    f1(99);    // error: f1(X(99)) not tried: non-const X& argument
    f2(99);    // OK: f2(X(99)); const X& argument
    f3(99);    // OK: f3(X(99)); X argument
}
```

An operation modifying the state of a class object should therefore be a member or a function taking a non-**const** reference argument (or a non-**const** pointer argument).

Operators that modify an operand (e.g., **=**, ***=**, and **++**) 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 (e.g., **+**, **–**, and **||**). However, such operators often need access to the representations of their operand class. Consequently, binary operators are the most common source of friend functions.

Unless 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 **m2=inv(m)** for producing a inverted **Matrix** from **m** to the alternative **m2=m.inv()**. On the other hand, if **inv()** inverts **m** itself, rather than producing a new **Matrix** that is the inverse of **m**, it should be a member.

All other things considered equal, implement operations that need direct access to a representation as member functions:

- It is not possible to know if someone someday will define a conversion operator.
- The member function call syntax makes it clear to the user that the object may be modified; a reference argument is far less obvious.
- 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.
- Member names are local to a class, so they tend to be shorter than the names of nonmember functions.
- If we have defined a member **f()** and we later feel the need for a nonmember **f(x)**, we can simply define it to mean **x.f()**.

Conversely, operations that do not need direct access to a representation are often best represented as nonmember functions, possibly in a namespace that makes their relationship with the class explicit (§18.3.6).

19.5 Advice

[1] Use `operator[]()` for subscripting and for selection based on a single value; §19.2.1.
[2] Use `operator()()` for call semantics, for subscripting, and for selection based on multiple values; §19.2.2.
[3] Use `operator–>()` to dereference ‘‘smart pointers’’; §19.2.3.
[4] Prefer prefix `++` over suffix `++`; §19.2.4.
[5] Define the global `operator new()` and `operator delete()` only if you really have to; §19.2.5.
[6] Define member `operator new()` and member `operator delete()` to control allocation and deallocation of objects of a specific class or hierarchy of classes; §19.2.5.
[7] Use user-defined literals to mimic conventional notation; §19.2.6.
[8] Place literal operators in separate namespaces to allow selective use; §19.2.6.
[9] For nonspecialized uses, prefer the standard `string` (Chapter 36) to the result of your own exercises; §19.3.
[10] Use a friend function if you need a nonmember function to have access to the representation of a class (e.g., to improve notation or to access the representation of two classes); §19.4.
[11] Prefer member functions to friend functions for granting access to the implementation of a class; §19.4.2.

20

Derived Classes

Do not multiply objects without necessity.
– William Occam

- Introduction
- Derived Classes
  Member Functions; Constructors and Destructors
- Class Hierarchies
  Type Fields; Virtual Functions; Explicit Qualification; Override Control; **using** Base Members; Return Type Relaxation
- Abstract Classes
- Access Control
  protected Members; Access to Base Classes; **using**-Declarations and Access Control
- Pointers to Members
  Pointers to Function Members; Pointers to Data Members; Base and Derived Members
- Advice

20.1 Introduction

From Simula, C++ borrowed the ideas of classes and class hierarchies. In addition, it borrowed the design idea 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 ideas distinguishes effective use of C++. Using language constructs as just notational props for traditional types of programming is to miss key strengths of C++.

A concept (idea, notion, etc.) does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with other 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 among concepts. However, we can't express arbitrary relationships directly in a programming language. Even if we could, we wouldn't want to. To be useful, 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 explicitly define class **Circle** and class **Triangle** to have class **Shape** in common. In that case, the common class, here **Shape**, is referred to as the *base* class or *superclass* and classes derived from that, here **Circle** and **Triangle**, are referred to as *derived* classes or *subclasses*. Representing a circle and a triangle in a program without involving the notion of a shape would be to miss 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 language features support building new classes from existing ones:

- *Implementation inheritance*: to save implementation effort by sharing facilities provided by a base class
- *Interface inheritance*: to allow different derived classes to be used interchangeably through the interface provided by a common base class

Interface inheritance is often referred to as *run-time polymorphism* (or *dynamic polymorphism*). In contrast, the uniform use of classes not related by inheritance provided by templates (§3.4, Chapter 23) is often referred to as *compile-time polymorphism* (or *static polymorphism*).

The discussion of class hierarchies is organized into three chapters:

- *Derived Classes* (Chapter 20): This chapter introduces the basic language features supporting object-oriented programming. Base and derived classes, virtual functions, and access control are covered.
- *Class Hierarchies* (Chapter 21): This chapter focuses on the use of base and derived classes to effectively organize code around the notion of class hierarchies. Most of this chapter is devoted to discussion of programming techniques, but technical aspects of multiple inheritance (classes with more than one base class) are also covered.
- *Run-time Type Identification* (Chapter 22): This chapter describes the techniques for explicitly navigating class hierarchies. In particular, the type conversion operations **dynamic_cast** and **static_cast** are presented, as is the operation for determining the type of an object given one of its base classes (**typeid**).

A brief introduction to the basic idea of hierarchical organization of types can be found in Chapter 3: base and derived classes (§3.2.2) and virtual functions (§3.2.3). These chapters examine these fundamental features and their associated programming and design techniques in greater detail.

20.2 Derived Classes

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
        list<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:

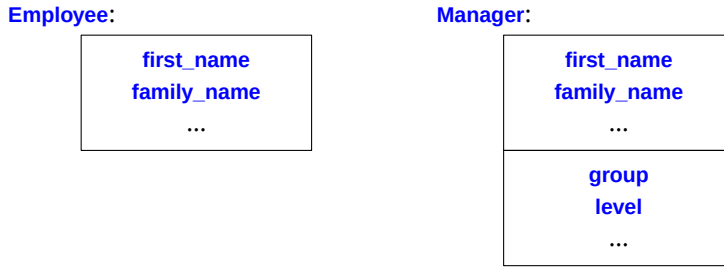
```
struct Manager : public Employee {
        list<Employee*> group;
        short level;
        // ...
};
```

The **Manager** is *derived* from **Employee**, and conversely, **Employee** is a *base class* for **Manager**. The class **Manager** has the members of class **Employee** (**first_name**, **department**, 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 typically larger (and never smaller) 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:



No memory overhead is implied by deriving a class. The space required is just the space required by the members.

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 {&m1,&e1);
    // ...
}
```

A **Manager** is (also) an **Employee**, so a **Manager*** can be used as an **Employee***. Similarly, a **Manager&** can be used as an **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 (§20.5) **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 §22.2.

Using a class as a base is equivalent to defining an (unnamed) object of that class. Consequently, a class must be defined in order to be used as a base (§8.2.2):

```
class Employee;     // declaration only, no definition

class Manager : public Employee { // error: Employee not defined
    // ...
};
```

20.2.1 Member Functions

Simple data structures, such as **Employee** and **Manager**, are really not that interesting and often not particularly useful. We need to provide a proper type with a suitable set of operations, and we need to do so without being tied to the details of a particular representation. For example:

```
class Employee {
public:
    void print() const;
    string full_name() const { return first_name + ' ' + middle_initial + ' ' + family_name; }
    // ...
private:
    string first_name, family_name;
    char middle_initial;
    // ...
};

class Manager : public Employee {
public:
    void print() const;
    // ...
};
```

A member of a derived class can use the public – and protected (see §20.5) – members of a 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 access private members of a base class:

```
void Manager::print() const
{
    cout << " name is " << family_name << '\n';          // error!
    // ...
}
```

This second version of **Manager::print()** will not compile because **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 (§20.5).

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
}
```

The result is a sequence of recursive calls ending with some form of program crash.

20.2.2 Constructors and Destructors

As usual, constructors and destructors are as essential:

- Objects are constructed from the bottom up (base before member and member before derived) and destroyed top-down (derived before member and member before base); §17.2.3.
- Each class can initialize its members and bases (but not directly members or bases of its bases); §17.4.1.
- Typically, destructors in a hierarchy need to be **virtual**; §17.2.5.
- Copy constructors of classes in a hierarchy should be used with care (if at all) to avoid slicing; §17.5.1.4.
- The resolution of a virtual function call, a **dynamic_cast**, or a **typeid()** in a constructor or destructor reflects the stage of construction and destruction (rather than the type of the yet-to-be-completed object); §22.4.

In computer science ‘‘up’’ and ‘‘down’’ can get very confused. In source text, definitions of base classes must occur before the definitions of their derived classes. This implies that for small examples, the bases appear above the derived classes on a screen. Furthermore, we tend to draw trees with the root on top. However, when I talk about constructing objects from the bottom up, I mean

starting with the most fundamental (e.g., base classes) and building what depends on that (e.g., derived classes) later. We build from the roots (base classes) toward the leaves (derived classes).

20.3 Class Hierarchies

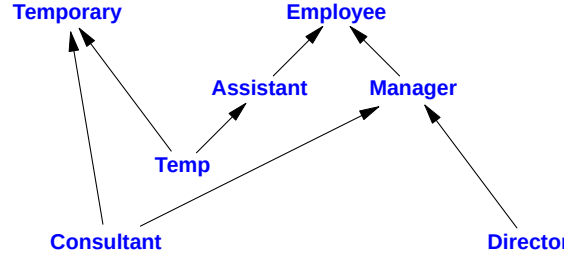
A derived class can itself be a base class. For example:

```
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 Assistant : public Employee { /* ... */ };
class Temp : public Temporary, public Assistant { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

or graphically:



Thus, as is explained in detail in §21.3, C++ can express a directed acyclic graph of classes.

20.3.1 Type Fields

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:

[1] Ensure that only objects of a single type are pointed to (§3.4, Chapter 23).
[2] Place a type field in the base class for the functions to inspect.
[3] Use `dynamic_cast` (§22.2, §22.6).
[4] Use virtual functions (§3.2.3, §20.3.2).

Unless you have used `final` (§20.3.4.2), solution 1 relies on more knowledge about the types involved than is available to the compiler. In general, it is not a good idea to try to be smarter than the type system, but (especially in combination with the use of templates) it can be used to implement homogeneous containers (e.g., the standard-library `vector` and `map`) with unsurpassed performance. 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 typically best avoided. The manager/employee example could be redefined like this:

```
struct Employee {
        enum Empl_type { man, empl };
        Empl_type type;

        Employee() : type{empl} { }

        string first_name, family_name;
        char middle_initial;

        Date hiring_date;
        short department;
        // ...
};

struct Manager : public Employee {
        Manager() { type = man; }

        list<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::empl:
                cout << e->family_name << '\t' << e->department << '\n';
                // ...
                break;
        case Employee::man:
        {       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 (auto x : elist)
        print_employee(x);
}
```

This works fine, especially in a small program maintained by a single person. However, it has a 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 often organized to take advantage of the commonality of the classes involved:

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::man) {
        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 a 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 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. In a large class hierarchy, accessible (not **private**) data in a common base class becomes the ‘‘global variables’’ of the hierarchy. For clean design and simpler maintenance, we want to keep separate issues separate and avoid mutual dependencies.

20.3.2 Virtual Functions

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 linker will guarantee the correct correspondence between objects and the functions applied to them. For example: