An Overview of the C++ Programming Language

Article · January 1998

DOI: 10.1201/9780849331350.sec3 · Source: CiteSeer

1 author:

Bjarne Stroustrup
Morgan Stanley
157 PUBLICATIONS **11,126** CITATIONS

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

An Overview of the C++ Programming Language

Bjarne Stroustrup

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

ABSTRACT

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

1 Introduction and Overview

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

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

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

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

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

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

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

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

2 The Design and Evolution of C++

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

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

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

2.1 C++ Design Aims

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

These aims can be summarized:

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

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

2.2 Design Principles

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

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

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

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

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

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

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

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

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

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

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

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

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

3 The C Programming Model

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

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

3.1 Arrays and Pointers

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

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

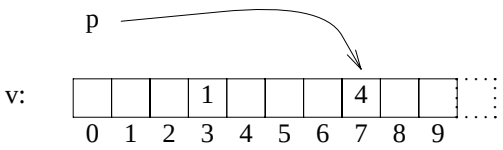
The subscript notation `[]` is used both in declarations to indicate an array and in expressions referring to elements of an array.

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

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

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

This can be represented graphically:



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

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

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

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

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

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

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

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

3.2 Storage

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

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

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

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

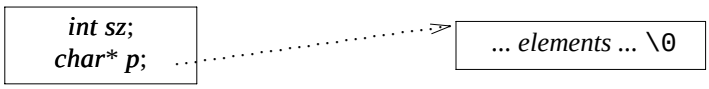
For example:

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

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

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

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

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



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

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

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

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

3.3 Compile, Link, and Execute

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

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

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

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

```
// file fact.c:

#include "fact.h"

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

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

```
// file fact.h:

long fact(long);
```

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

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

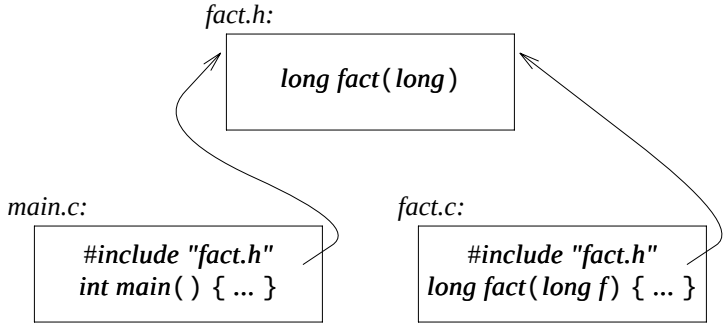
```
// file main.c:

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

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

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

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

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

Graphically, the program fragments can be represented like this:



3.4 Type checking

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

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

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

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

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

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

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

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

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

4 Abstraction

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

4.1 Concrete Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Abstract Types

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

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

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

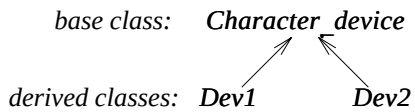
The ***Character_device*** is an abstract class specifying an interface only. This interface can be implemented in a variety of ways without affecting users. For example, a programmer might use this interface to device drivers on some hypothetical system like this:

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

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

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

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

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

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

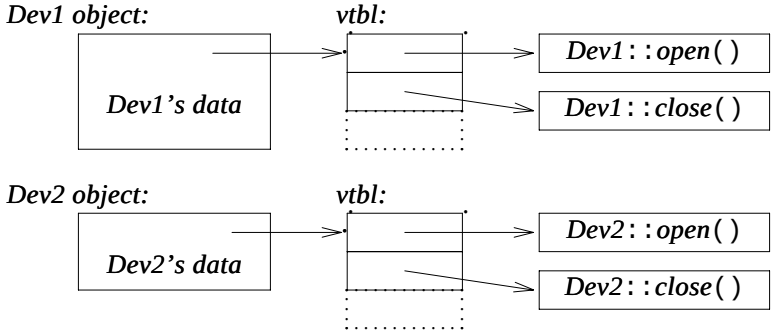


An arrow represents the *derived from* relationship. The ***user( )*** does not need to know whether a ***Dev1***, or a ***Dev2***, or some other class implementing the ***Character_device*** is actually used.

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

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

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



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

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

4.2.1 Destructors

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

A simple stack of characters can be defined like this:

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

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

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

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

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

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

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

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

4.3 Object-Oriented Programming

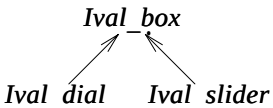
Object-oriented programming is a set of techniques that rely on hierarchies of classes to provide extensibility and flexibility. The basic language facilities used are the user-defined types themselves, the ability to derive a class from another, and *virtual* functions (§4.2). These features allow a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

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

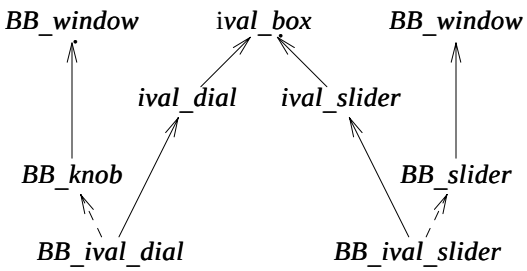
Naturally, there will be a variety of Ival_boxes:

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

This can be represented graphically like this:



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



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

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

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

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

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

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

4.3.1 Run-time Type Identification

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

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

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

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

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

4.4 Generic Programming

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

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

4.5 Containers

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

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

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

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

We can now use stacks like this:

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

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

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

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

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

4.5.1 Algorithms

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

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

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

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

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

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

5 Large-scale Programming

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

5.1 Exceptions and Error Handling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Namespaces

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

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

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

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

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

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

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

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

6 The C++ Standard Library

The standard library provides:

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

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

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

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

6.1 Strings and I/O

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

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

int main()
{
    using namespace std;

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

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

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

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

6.2 Containers

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

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

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

subscripting to be constructed incrementally. For example:

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;

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

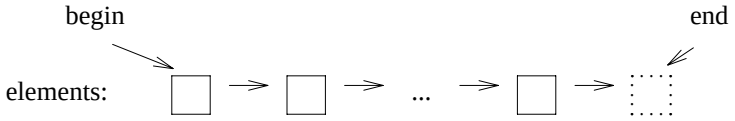
The containers are nonintrusive; that is, essentially every type can be used as a container element type. In particular, built-in types – such as ***int*** and ***char**** – and C-style data structures (***struct***s) can be container elements.

6.3 Algorithms

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

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

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



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

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

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

6.4 Numerics

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

6.4.1 Complex Numbers

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

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

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

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

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

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

6.4.2 Vector Arithmetic

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

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

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

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

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

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

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

7 Use of C++

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

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

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

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

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

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

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

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

8 Further Reading

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

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

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

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

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

9 Acknowledgements

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

10 References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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