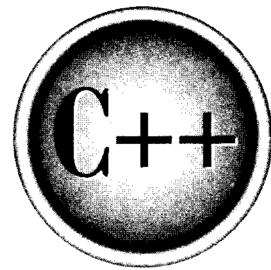
The Complete Reference

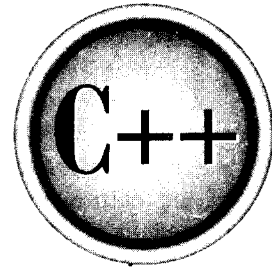


Part II

C++

Part One examined the C subset of C++. Part Two describes those features of the language specific to C++. That is, it discusses those features of C++ that it does not have in common with C. Because many of the C++ features are designed to support object-oriented programming (OOP), Part Two also provides a discussion of its theory and merits. We will begin with an overview of C++.

The Complete Reference



Chapter 11

An Overview of C++

This chapter provides an overview of the key concepts embodied in C++. C++ is an object-oriented programming language, and its object-oriented features are highly interrelated. In several instances, this interrelatedness makes it difficult to describe one feature of C++ without implicitly involving several others. Moreover, the object-oriented features of C++ are, in many places, so intertwined that discussion of one feature *implies* prior knowledge of another. To address this problem, this chapter presents a quick overview of the most important aspects of C++, including its history, its key features, and the difference between traditional and Standard C++. The remaining chapters examine C++ in detail.

The Origins of C++

C++ began as an expanded version of C. The C++ extensions were first invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes." However, in 1983 the name was changed to C++.

Although C was one of the most liked and widely used professional programming languages in the world, the invention of C++ was necessitated by one major programming factor: increasing complexity. Over the years, computer programs have become larger and more complex. Even though C is an excellent programming language, it has its limits. In C, once a program exceeds from 25,000 to 100,000 lines of code, it becomes so complex that it is difficult to grasp as a totality. The purpose of C++ is to allow this barrier to be broken. The essence of C++ is to allow the programmer to comprehend and manage larger, more complex programs.

Most additions made by Stroustrup to C support object-oriented programming, sometimes referred to as OOP. (See the next section for a brief explanation of object-oriented programming.) Stroustrup states that some of C++'s object-oriented features were inspired by another object-oriented language called Simula67. Therefore, C++ represents the blending of two powerful programming methods.

Since C++ was first invented, it has undergone three major revisions, with each adding to and altering the language. The first revision was in 1985 and the second in 1990. The third occurred during the standardization of C++. Several years ago, work began on a standard for C++. Toward that end, a joint ANSI (American National Standards Institute) and ISO (International Standards Organization) standardization committee was formed. The first draft of the proposed standard was created on January 25, 1994. In that draft, the ANSI/ISO C++ committee (of which I was a member) kept the features first defined by Stroustrup and added some new ones as well. But in general, this initial draft reflected the state of C++ at the time.

Soon after the completion of the first draft of the C++ standard, an event occurred that caused the language to be greatly expanded: the creation of the Standard Template Library (STL) by Alexander Stepanov. The STL is a set of generic routines that you can use to manipulate data. It is both powerful and elegant, but also quite large. Subsequent

to the first draft, the committee voted to include the STL in the specification for C++. The addition of the STL expanded the scope of C++ well beyond its original definition. While important, the inclusion of the STL, among other things, slowed the standardization of C++.

It is fair to say that the standardization of C++ took far longer than anyone had expected when it began. In the process, many new features were added to the language and many small changes were made. In fact, the version of C++ defined by the C++ committee is much larger and more complex than Stroustrup's original design. The final draft was passed out of committee on November 14, 1997 and an ANSI/ISO standard for C++ became a reality in 1998. This specification for C++ is commonly referred to as *Standard* C++.

The material in this book describes Standard C++, including all of its newest features. This is the version of C++ created by the ANSI/ISO standardization committee, and it is the one that is currently accepted by all major compilers.

What Is Object-Oriented Programming?

Since object-oriented programming (OOP) drove the creation of C++, it is necessary to understand its foundational principles. OOP is a powerful way to approach the job of programming. Programming methodologies have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of programs. For example, when computers were first invented, programming was done by toggling in the binary machine instructions using the computer's front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs, using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity. The first widespread language was, of course, FORTRAN. Although FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easy-to-understand programs.

The 1960s gave birth to structured programming. This is the method encouraged by languages such as C and Pascal. The use of structured languages made it possible to write moderately complex programs fairly easily. Structured languages are characterized by their support for stand-alone subroutines, local variables, rich control constructs, and their lack of reliance upon the GOTO. Although structured languages are a powerful tool, they reach their limit when a project becomes too large.

Consider this: At each milestone in the development of programming, techniques and tools were created to allow the programmer to deal with increasingly greater complexity. Each step of the way, the new approach took the best elements of the previous methods and moved forward. Prior to the invention of OOP, many projects were nearing (or exceeding) the point where the structured approach no longer

worked. Object-oriented methods were created to help programmers break through these barriers.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data." For example, a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program.

Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.

Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

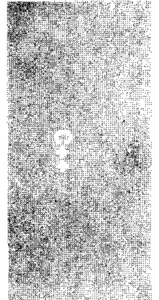
For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Polymorphism

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The

specific action selected is determined by the exact nature of the situation. A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat.

This same principle can also apply to programming. For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push( )** and **pop( )**, that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack—the functions **push( )** and **pop( )**—are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data.



Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the *general interface*.

The first object-oriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications. For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. As you will see, inheritance is an important aspect of object-oriented programming.

Some C++ Fundamentals

In Part One, the C subset of C++ was described and C programs were used to demonstrate those features. From this point forward, all examples will be "C++

programs." That is, they will be making use of features unique to C++. For ease of discussion, we will refer to these C++-specific features simply as "C++ features" from now on.

If you come from a C background, or if you have been studying the C subset programs in Part One, be aware that C++ programs differ from C programs in some important respects. Most of the differences have to do with taking advantage of C++'s object-oriented capabilities. But C++ programs differ from C programs in other ways, including how I/O is performed and what headers are included. Also, most C++ programs share a set of common traits that clearly identify them *as* C++ programs. Before moving on to C++'s object-oriented constructs, an understanding of the fundamental elements of a C++ program is required.

This section describes several issues relating to nearly all C++ programs. Along the way, some important differences with C and earlier versions of C++ are pointed out.

A Sample C++ Program

Let's start with the short sample C++ program shown here.

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

int main()
{
  int i;

  cout << "This is output.\n";  // this is a single line comment
  /* you can still use C style comments */

  // input a number using >>
  cout << "Enter a number: ";
  cin >> i;

  // now, output a number using <<
  cout << i << " squared is " << i*i << "\n";

  return 0;
}
```

As you can see, this program looks much different from the C subset programs found in Part One. A line-by-line commentary will be useful. To begin, the header **<iostream>** is included. This header supports C++-style I/O operations. (**<iostream>** is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the

name **iostream**. The reason is that **<iostream>** is one of the modern-style headers defined by Standard C++. Modern C++ headers do not use the **.h** extension.

The next line in the program is

```
using namespace std;
```

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. A namespace creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs. The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace you simplify access to the standard library. The programs in Part One, which use only the C subset, don't need a namespace statement because the C library functions are also available in the default, global namespace.

Note *Since both new-style headers and namespaces are recent additions to C++, you may encounter older code that does not use them. Also, if you are using an older compiler, it may not support them. Instructions for using an older compiler are found later in this chapter.*

Now examine the following line.

```
int main()
```

Notice that the parameter list in **main( )** is empty. In C++, this indicates that **main( )** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list, as shown here:

```
int main(void)
```

This was the way **main( )** was declared in the programs in Part One. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

The next line contains two C++ features.

```
cout << "This is output.\n";  // this is a single line comment
```

First, the statement

```
cout << "This is output.\n";
```

causes **This is output.** to be displayed on the screen, followed by a carriage return-linefeed combination. In C++, the << has an expanded role. It is still the left shift operator, but when it is used as shown in this example, it is also an *output operator.* The word **cout** is an identifier that is linked to the screen. (Actually, like C, C++ supports I/O redirection, but for the sake of discussion, assume that **cout** refers to the screen.) You can use **cout** and the << to output any of the built-in data types, as well as strings of characters.

Note that you can still use **printf( )** or any other of C's I/O functions in a C++ program. However, most programmers feel that using << is more in the spirit of C++. Further, while using **printf( )** to output a string is virtually equivalent to using << in this case, the C++ I/O system can be expanded to perform operations on objects that you define (something that you cannot do using **printf( )**).

What follows the output expression is a C++ *single-line comment*. As mentioned in Chapter 10, C++ defines two types of comments. First, you may use a multiline comment, which works the same in C++ as in C. You can also define a single-line comment by using //; whatever follows such a comment is ignored by the compiler until the end of the line is reached. In general, C++ programmers use multiline comments when a longer comment is being created and use single-line comments when only a short remark is needed.

Next, the program prompts the user for a number. The number is read from the keyboard with this statement:

```
cin >> i;
```

In C++, the >> operator still retains its right shift meaning. However, when used as shown, it also is C++'s *input operator.* This statement causes **i** to be given a value read from the keyboard. The identifier **cin** refers to the standard input device, which is usually the keyboard. In general, you can use **cin** >> to input a variable of any of the basic data types plus strings.

Note *The line of code just described is not misprinted. Specifically, there is not supposed to be an & in front of the* ***i****. When inputting information using a C-based function like* ***scanf( )****, you have to explicitly pass a pointer to the variable that will receive the information. This means preceding the variable name with the "address of" operator, &. However, because of the way the >> operator is implemented in C++, you do not need (in fact, must not use) the &. The reason for this is explained in Chapter 13.*

Although it is not illustrated by the example, you are free to use any of the C-based input functions, such as **scanf( )**, instead of using >>. However, as with **cout**, most programmers feel that **cin** >> is more in the spirit of C++.

Another interesting line in the program is shown here:

```
cout << i << "squared is " << i*i << "\n";
```

Assuming that **i** has the value 10, this statement causes the phrase **10 squared is 100** to be displayed, followed by a carriage return-linefeed. As this line illustrates, you can run together several << output operations.

The program ends with this statement:

```
return 0;
```

This causes zero to be returned to the calling process (which is usually the operating system). This works the same in C++ as it does in C. Returning zero indicates that the program terminated normally. Abnormal program termination should be signaled by returning a nonzero value. You may also use the values **EXIT_SUCCESS** and **EXIT_FAILURE** if you like.

A Closer Look at the I/O Operators

As stated, when used for I/O, the << and >> operators are capable of handling any of C++'s built-in data types. For example, this program inputs a **float**, a **double**, and a string and then outputs them:

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

int main()
{
  float f;
  char str[80];
  double d;

  cout << "Enter two floating point numbers: ";
  cin >> f >> d;

  cout << "Enter a string: ";
  cin >> str;

  cout << f << " " << d << " " << str;

  return 0;
}
```

When you run this program, try entering **This is a test.** when prompted for the string. When the program redisplays the information you entered, only the word "This" will be displayed. The rest of the string is not shown because the >> operator stops reading input when the first white-space character is encountered. Thus, "is a test" is

never read by the program. This program also illustrates that you can string together several input operations in a single statement.

The C++ I/O operators recognize the entire set of backslash character constants described in Chapter 2. For example, it is perfectly acceptable to write

```
cout << "A\tB\tC";
```

This statement outputs the letters A, B, and C, separated by tabs.

Declaring Local Variables

If you come from a C background, you need to be aware of an important difference between C and C++ regarding when local variables can be declared. In C89, you must declare all local variables used within a block at the start of that block. You cannot declare a variable in a block after an "action" statement has occurred. For example, in C89, this fragment is incorrect:

```
/* Incorrect in C89. OK in C++. */
int f()
{
  int i;
  i = 10;

  int j;  /* won't compile as a C program */
  j = i*2;

  return j;
}
```

In a C89 program, this function is in error because the assignment intervenes between the declaration of **i** and that of **j**. However, when compiling it as a C++ program, this fragment is perfectly acceptable. In C++ (and C99) you may declare local variables at any point within a block—not just at the beginning.

Here is another example. This version of the program from the preceding section declares **str** just before it is needed.

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

int main()
{
  float f;
```

```
    double d;
    cout << "Enter two floating point numbers: ";
    cin >> f >> d;

    cout << "Enter a string: ";
    char str[80];  // str declared here, just before 1st use
    cin >> str;

    cout << f << " " << d << " " << str;

    return 0;
}
```

Whether you declare all variables at the start of a block or at the point of first use is completely up to you. Since much of the philosophy behind C++ is the encapsulation of code and data, it makes sense that you can declare variables close to where they are used instead of just at the beginning of the block. In the preceding example, the declarations are separated simply for illustration, but it is easy to imagine more complex examples in which this feature of C++ is more valuable.

Declaring variables close to where they are used can help you avoid accidental side effects. However, the greatest benefit of declaring variables at the point of first use is gained in large functions. Frankly, in short functions (like many of the examples in this book), there is little reason not to simply declare variables at the start of a function. For this reason, this book will declare variables at the point of first use only when it seems warranted by the size or complexity of a function.

There is some debate as to the general wisdom of localizing the declaration of variables. Opponents suggest that sprinkling declarations throughout a block makes it harder, not easier, for someone reading the code to find quickly the declarations of all variables used in that block, making the program harder to maintain. For this reason, some C++ programmers do not make significant use of this feature. This book will not take a stand either way on this issue. However, when applied properly, especially in large functions, declaring variables at the point of their first use can help you create bug-free programs more easily.

No Default to int

A few years ago, there was a change to C++ that may affect older C++ code as well as C code being ported to C++. Both C89 and the original specification for C++ state that when no explicit type is specified in a declaration, type **int** is assumed. However, the "default-to-int" rule was dropped from C++ during standardization. C99 also drops this rule. However, there is still a large body of C and older C++ code that uses this rule.

The most common use of the "default-to-int" rule is with function return types. It was common practice to not specify **int** explicitly when a function returned an integer result. For example, in C89 and older C++ code the following function is valid.

```
func(int i)
{
  return i*i;
}
```

In Standard C++, this function must have the return type of **int** specified, as shown here.

```
int func(int i)
{
  return i*i;
}
```

As a practical matter, nearly all C++ compilers still support the "default-to-int" rule for compatibility with older code. However, you should not use this feature for new code because it is no longer allowed.

The bool Data Type

C++ defines a built-in Boolean type called **bool**. Objects of type **bool** can store only the values **true** or **false**, which are keywords defined by C++. As explained in Part One, automatic conversions take place which allow **bool** values to be converted to integers, and vice versa. Specifically, any non-zero value is converted to **true** and zero is converted to **false**. The reverse also occurs; **true** is converted to 1 and **false** is converted to zero. Thus, the fundamental concept of zero being false and non-zero being true is still fully entrenched in the C++ language.

Note *Although C89 (the C subset of C++) does not define a Boolean type, C99 adds to the C language a type called* ***_Bool****, which is capable of storing the values 1 and 0 (i.e., true/false). Unlike C++, C99 does not define* ***true*** *and* ***false*** *as keywords. Thus,* ***_Bool*** *as defined by C99 is incompatible with* ***bool*** *as defined by C++.*

The reason that C99 specifies ***_Bool*** *rather than* ***bool*** *as a keyword is that many preexisting C programs have aleady defined their own custom versions of* ***bool****. By defining the Boolean type as* ***_Bool****, C99 avoids breaking this preexisting code. However, it is possible to achieve compatibility between C++ and C99 on this point because C99 adds the header* ***<stdbool.h>*** *which defines the macros* ***bool****,* ***true****, and* ***false****. By including this header, you can create code that is compatible with both C99 and C++.*

Old-Style vs. Modern C++

As explained, C++ underwent a rather extensive evolutionary process during its development and standardization. As a result, there are really two versions of C++. The first is the traditional version that is based upon Bjarne Stroustrup's original designs. The second is Standard C++, which was created by Stroustrup and the ANSI/ISO standardization committee. While these two versions of C++ are very similar at their core, Standard C++ contains several enhancements not found in traditional C++. Thus, Standard C++ is essentially a superset of traditional C++.

This book describes Standard C++. This is the version of C++ defined by the ANSI/ISO standardization committee and the one implemented by all modern C++ compilers. The code in this book reflects the contemporary coding style and practices as encouraged by Standard C++. However, if you are using an older compiler, it may not accept all of the programs in this book. Here's why. During the process of standardization, the ANSI/ISO committee added many new features to the language. As these features were defined, they were implemented by compiler developers. Of course, there is always a lag time between when a new feature is added to the language and when it is available in commercial compilers. Since features were added to C++ over a period of years, an older compiler might not support one or more of them. This is important because two recent additions to the C++ language affect every program that you will write—even the simplest. If you are using an older compiler that does not accept these new features, don't worry. There is an easy work-around, which is described here.

The key differences between old-style and modern code involve two features: new-style headers and the **namespace** statement. To understand the differences, we will begin by looking at two versions of a minimal, do-nothing C++ program. The first version shown here reflects the way C++ programs were written using old-style coding.

```
/*
   An old-style C++ program.
*/

#include <iostream.h>

int main()
{
  return 0;
}
```

Pay special attention to the **#include** statement. It includes the file **iostream.h**, not the header **<iostream>**. Also notice that no **namespace** statement is present.

Here is the second version of the skeleton, which uses the modern style.

```
/*
  A modern-style C++ program that uses
  the new-style headers and a namespace.
*/
#include <iostream>
using namespace std;

int main()
{
  return 0;
}
```

This version uses the C++-style header and specifies a namespace. Both of these features were mentioned in passing earlier. Let's look closely at them now.

The New C++ Headers

As you know, when you use a library function in a program, you must include its header. This is done using the **#include** statement. For example, in C, to include the header for the I/O functions, you include **stdio.h** with a statement like this:

```
#include <stdio.h>
```

Here, **stdio.h** is the name of the file used by the I/O functions, and the preceding statement causes that file to be included in your program. The key point is that this **#include** statement normally *includes a file*.

When C++ was first invented and for several years after that, it used the same style of headers as did C. That is, it used *header files*. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ created a new kind of header that is used by the Standard C++ library. The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that may be mapped to files by the compiler, although they need not be. The new-style C++ headers are an abstraction that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Since the new-style headers are not filenames, they do not have a **.h** extension. They consist solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++.

<iostream> <fstream> <vector> <string>

The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.

Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** or **ctype.h** are still available. However, Standard C++ also defines new-style headers that you can use in place of these header files. The C++ versions of the C standard headers simply add a "c" prefix to the filename and drop the **.h**. For example, the C++ new-style header for **math.h** is **<cmath>**. The one for **string.h** is **<cstring>**. Although it is currently permissible to include a C-style header file when using C library functions, this approach is deprecated by Standard C++ (that is, it is not recommended). For this reason, from this point forward, this book will use new-style C++ headers in all **#include** statements. If your compiler does not support new-style headers for the C function library, then simply substitute the old-style, C-like headers.

Since the new-style header is a relatively recent addition to C++, you will still find many, many older programs that don't use it. These programs employ C-style headers, in which a filename is specified. As the old-style skeletal program shows, the traditional way to include the I/O header is as shown here.

```
#include <iostream.h>
```

This causes the file **iostream.h** to be included in your program. In general, an old-style header file will use the same name as its corresponding new-style header with a **.h** appended.

As of this writing, all C++ compilers support the old-style headers. However, the old-style headers have been declared obsolete and their use in new programs is not recommended. This is why they are not used in this book.

Remember *While still common in existing C++ code, old-style headers are obsolete.*

Namespaces

When you include a new-style header in your program, the contents of that header are contained in the **std** namespace. A *namespace* is simply a declarative region. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. Elements declared in one namespace are separate from elements declared in another. Originally, the names of the C++ library functions, etc., were simply put into the global namespace (as they are in C). However, with the advent of the new-style headers, the contents of these headers were placed in the **std** namespace. We will look closely at namespaces later in this book. For now, you won't need to worry about them because the statement

```
using namespace std;
```

brings the **std** namespace into visibility (i.e., it puts **std** into the global namespace). After this statement has been compiled, there is no difference between working with an old-style header and a new-style one.

One other point: for the sake of compatibility, when a C++ program includes a C header, such as **stdio.h**, its contents are put into the global namespace. This allows a C++ compiler to compile C-subset programs.

Working with an Old Compiler

As explained, both namespaces and the new-style headers are fairly recent additions to the C++ language, added during standardization. While all new C++ compilers support these features, older compilers may not. When this is the case, your compiler will report one or more errors when it tries to compile the first two lines of the sample programs in this book. If this is the case, there is an easy work-around: simply use an old-style header and delete the **namespace** statement. That is, just replace

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

with

```
#include <iostream.h>
```

This change transforms a modern program into an old-style one. Since the old-style header reads all of its contents into the global namespace, there is no need for a **namespace** statement.

One other point: for now and for the next few years, you will see many C++ programs that use the old-style headers and do not include a **using** statement. Your C++ compiler will be able to compile them just fine. However, for new programs, you should use the modern style because it is the only style of program that complies with the C++ Standard. While old-style programs will continue to be supported for many years, they are technically noncompliant.

Introducing C++ Classes

This section introduces C++'s most important feature: the class. In C++, to create an object, you first must define its general form by using the keyword **class**. A **class** is similar syntactically to a structure. Here is an example. The following class defines a type called **stack**, which will be used to create a stack:

```
#define SIZE 100
```

```
// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};
```

A **class** may contain private as well as public parts. By default, all items defined in a **class** are private. For example, the variables **stck** and **tos** are private. This means that they cannot be accessed by any function that is not a member of the **class**. This is one way that encapsulation is achieved—access to certain items of data may be tightly controlled by keeping them private. Although it is not shown in this example, you can also define private functions, which then may be called only by other members of the **class**.

To make parts of a **class** public (that is, accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after **public** can be accessed by all other functions in the program. Essentially, the rest of your program accesses an object through its public functions. Although you can have public variables, good practice dictates that you should try to limit their use. Instead, you should make all data private and control access to it through public functions. One other point: Notice that the **public** keyword is followed by a colon.

The functions **init( )**, **push( )**, and **pop( )** are called *member functions* because they are part of the class **stack**. The variables **stck** and **tos** are called *member variables* (or *data members*). Remember, an object forms a bond between code and data. Only member functions have access to the private members of their class. Thus, only **init( )**, **push( )**, and **pop( )** may access **stck** and **tos**.

Once you have defined a **class**, you can create an object of that type by using the class name. In essence, the class name becomes a new data type specifier. For example, this creates an object called **mystack** of type **stack**:

```
stack mystack;
```

When you declare an object of a class, you are creating an *instance* of that class. In this case, **mystack** is an instance of **stack**. You may also create objects when the **class** is defined by putting their names after the closing curly brace, in exactly the same way as you would with a structure.

To review: In C++, **class** creates a new data type that may be used to create objects of that type. Therefore, an object is an instance of a class in just the same way that some other variable is an instance of the **int** data type, for example. Put differently, a class is a

logical abstraction, while an object is real. (That is, an object exists inside the memory of the computer.)

The general form of a simple **class** declaration is

class *class-name* {
  private data and functions
public:
  public data and functions
} *object name list;*

Of course, the *object name list* may be empty.

Inside the declaration of **stack**, member functions were identified using their prototypes. In C++, all functions must be prototyped. Prototypes are not optional. The prototype for a member function within a class definition serves as that function's prototype in general.

When it comes time to actually code a function that is the member of a class, you must tell the compiler which class the function belongs to by qualifying its name with the name of the class of which it is a member. For example, here is one way to code the **push( )** function:

```
void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}
```

The :: is called the *scope resolution operator.* Essentially, it tells the compiler that this version of **push( )** belongs to the **stack** class or, put differently, that this **push( )** is in **stack**'s scope. In C++, several different classes can use the same function name. The compiler knows which function belongs to which class because of the scope resolution operator.

When you refer to a member of a class from a piece of code that is not part of the class, you must always do so in conjunction with an object of that class. To do so, use the object's name, followed by the dot operator, followed by the name of the member. This rule applies whether you are accessing a data member or a function member. For example, this calls **init( )** for object **stack1**.

```
stack stack1, stack2;

stack1.init();
```

This fragment creates two objects, **stack1** and **stack2**, and initializes **stack1**. Understand that **stack1** and **stack2** are two separate objects. This means, for example, that initializing **stack1** does *not* cause **stack2** to be initialized as well. The only relationship **stack1** has with **stack2** is that they are objects of the same type.

Within a class, one member function can call another member function or refer to a data member directly, without using the dot operator. It is only when a member is referred to by code that does not belong to the class that the object name and the dot operator must be used.

The program shown here puts together all the pieces and missing details and illustrates the **stack** class:

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

#define SIZE 100

// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  void init();
  void push(int i);
  int pop();
};

void stack::init()
{
  tos = 0;
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
```

```
      cout << "Stack underflow.\n";
      return 0;
    }
    tos--;
    return stck[tos];
  }

  int main()
  {
    stack stack1, stack2;  // create two stack objects

    stack1.init();
    stack2.init();

    stack1.push(1);
    stack2.push(2);

    stack1.push(3);
    stack2.push(4);

    cout << stack1.pop() << " ";
    cout << stack1.pop() << " ";
    cout << stack2.pop() << " ";
    cout << stack2.pop() << "\n";

    return 0;
  }
```

The output from this program is shown here.

```
3 1 4 2
```

One last point: Recall that the private members of an object are accessible only by functions that are members of that object. For example, a statement like

```
stack1.tos = 0; // Error, tos is private.
```

could not be in the **main( )** function of the previous program because **tos** is private.

Function Overloading

One way that C++ achieves polymorphism is through the use of function overloading. In C++, two or more functions can share the same name as long as their parameter declarations are different. In this situation, the functions that share the same name are said to be *overloaded*, and the process is referred to as *function overloading*.

To see why function overloading is important, first consider three functions defined by the C subset: **abs( )**, **labs( )**, and **fabs( )**. The **abs( )** function returns the absolute value of an integer, **labs( )** returns the absolute value of a **long**, and **fabs( )** returns the absolute value of a **double**. Although these functions perform almost identical actions, in C three slightly different names must be used to represent these essentially similar tasks. This makes the situation more complex, conceptually, than it actually is. Even though the underlying concept of each function is the same, the programmer has to remember three things, not just one. However, in C++, you can use just one name for all three functions, as this program illustrates:

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

// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
  cout << abs(-10) << "\n";

  cout << abs(-11.0) << "\n";

  cout << abs(-9L) << "\n";

  return 0;
}

int abs(int i)
{
  cout << "Using integer abs()\n";
```

```
  return i<0 ? -i : i;
}

double abs(double d)
{
  cout << "Using double abs()\n";

  return d<0.0 ? -d : d;
}

long abs(long l)
{
  cout << "Using long abs()\n";

  return l<0 ? -l : l;
}
```

The output from this program is shown here.

```
Using integer abs()
10
Using double abs()
11
Using long abs()
9
```

This program creates three similar but different functions called **abs( )**, each of which returns the absolute value of its argument. The compiler knows which function to call in each situation because of the type of the argument. The value of overloaded functions is that they allow related sets of functions to be accessed with a common name. Thus, the name **abs( )** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific method* for a particular circumstance. You need only remember the general action being performed. Due to polymorphism, three things to remember have been reduced to one. This example is fairly trivial, but if you expand the concept, you can see how polymorphism can help you manage very complex programs.

In general, to overload a function, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction when overloading a function: the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types. They must differ in the types or number of their parameters. (Return types do not provide sufficient information in all cases for the compiler to decide which function to use.) Of course, overloaded functions *may* differ in their return types, too.

Here is another example that uses overloaded functions:

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

void stradd(char *s1, char *s2);
void stradd(char *s1, int i);

int main()
{
  char str[80];

  strcpy(str, "Hello ");
  stradd(str, "there");
  cout << str << "\n";

  stradd(str, 100);
  cout << str << "\n";

  return 0;
}

// concatenate two strings
void stradd(char *s1, char *s2)
{
  strcat(s1, s2);
}

// concatenate a string with a "stringized" integer
void stradd(char *s1, int i)
{
  char temp[80];

  sprintf(temp, "%d", i);
  strcat(s1, temp);
}
```

In this program, the function **stradd( )** is overloaded. One version concatenates two strings (just like **strcat( )** does). The other version "stringizes" an integer and then appends that to a string. Here, overloading is used to create one interface that appends either a string or an integer to another string.

You can use the same name to overload unrelated functions, but you should not. For example, you could use the name **sqr( )** to create functions that return the *square* of an **int** and the *square root* of a **double**. However, these two operations are fundamentally different; applying function overloading in this manner defeats its purpose (and, in fact, is considered bad programming style). In practice, you should overload only closely related operations.

Operator Overloading

Polymorphism is also achieved in C++ through operator overloading. As you know, in C++, it is possible to use the << and >> operators to perform console I/O operations. They can perform these extra operations because in the **<iostream>** header, these operators are overloaded. When an operator is overloaded, it takes on an additional meaning relative to a certain class. However, it still retains all of its old meanings.

In general, you can overload most of C++'s operators by defining what they mean relative to a specific class. For example, think back to the **stack** class developed earlier in this chapter. It is possible to overload the **+** operator relative to objects of type **stack** so that it appends the contents of one stack to the contents of another. However, the **+** still retains its original meaning relative to other types of data.

Because operator overloading is, in practice, somewhat more complex than function overloading, examples are deferred until Chapter 14.

Inheritance

As stated earlier in this chapter, inheritance is one of the major traits of an object-oriented programming language. In C++, inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a *base class*, which defines those qualities common to all objects to be derived from the base. The base class represents the most general description. The classes derived from the base are usually referred to as *derived classes*. A derived class includes all features of the generic base class and then adds qualities specific to the derived class. To demonstrate how this works, the next example creates classes that categorize different types of buildings.

To begin, the **building** class is declared, as shown here. It will serve as the base for two derived classes.

```
class building {
  int rooms;
  int floors;
  int area;
```

```
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};
```

Because (for the sake of this example) all buildings have three common features—one or more rooms, one or more floors, and a total area—the **building** class embodies these components into its declaration. The member functions beginning with **set** set the values of the private data. The functions starting with get return those values.

You can now use this broad definition of a building to create derived classes that describe specific types of buildings. For example, here is a derived class called **house:**

```
// house is derived from building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};
```

Notice how **building** is inherited. The general form for inheritance is

class *derived-class : access base-class* {
  // *body of new class*
}

Here, *access* is optional. However, if present, it must be **public**, **private**, or **protected**. (These options are further examined in Chapter 12.) For now, all inherited classes will use **public**. Using **public** means that all of the public members of the base class will become public members of the derived class. Therefore, the public members of the class **building** become public members of the derived class **house** and are available to the member functions of **house** just as if they had been declared inside **house**. However, **house**'s member functions *do not* have access to the private elements of **building**. This is an important point. Even though **house** inherits **building**, it has access only to the

public members of **building**. In this way, inheritance does not circumvent the principles of encapsulation necessary to OOP.

Remember *A derived class has direct access to both its own members and the public members of the base class.*

Here is a program illustrating inheritance. It creates two derived classes of **building** using inheritance; one is **house**, the other, **school**.

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

class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};

// house is derived from building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};

// school is also derived from building
class school : public building {
  int classrooms;
  int offices;
public:
  void set_classrooms(int num);
```

```
  int get_classrooms();
  void set_offices(int num);
  int get_offices();
};

void building::set_rooms(int num)
{
  rooms = num;
}

void building::set_floors(int num)
{
  floors = num;
}

void building::set_area(int num)
{
  area = num;
}

int building::get_rooms()
{
  return rooms;
}

int building::get_floors()
{
  return floors;
}

int building::get_area()
{
  return area;
}

void house::set_bedrooms(int num)
{
  bedrooms = num;
}

void house::set_baths(int num)
{
```

```
  baths = num;
}

int house::get_bedrooms()
{
  return bedrooms;
}

int house::get_baths()
{
  return baths;
}

void school::set_classrooms(int num)
{
  classrooms = num;
}

void school::set_offices(int num)
{
  offices = num;
}

int school::get_classrooms()
{
  return classrooms;
}

int school::get_offices()
{
  return offices;
}

int main()
{
  house h;
  school s;

  h.set_rooms(12);
  h.set_floors(3);
  h.set_area(4500);
  h.set_bedrooms(5);
```

```
    h.set_baths(3);

    cout << "house has " << h.get_bedrooms();
    cout << " bedrooms\n";

    s.set_rooms(200);
    s.set_classrooms(180);
    s.set_offices(5);
    s.set_area(25000);

    cout << "school has " << s.get_classrooms();
    cout << " classrooms\n";
    cout << "Its area is " << s.get_area();

    return 0;
}
```

The output produced by this program is shown here.

```
house has 5 bedrooms
school has 180 classrooms
Its area is 25000
```

As this program shows, the major advantage of inheritance is that you can create a general classification that can be incorporated into more specific ones. In this way, each object can precisely represent its own subclass.

When writing about C++, the terms *base* and *derived* are generally used to describe the inheritance relationship. However, the terms *parent* and *child* are also used. You may also see the terms *superclass* and *subclass*.

Aside from providing the advantages of hierarchical classification, inheritance also provides support for run-time polymorphism through the mechanism of **virtual** functions. (Refer to Chapter 16 for details.)

Constructors and Destructors

It is very common for some part of an object to require initialization before it can be used. For example, think back to the **stack** class developed earlier in this chapter. Before the stack could be used, **tos** had to be set to zero. This was performed by using the function **init( )**. Because the requirement for initialization is so common, C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function.

A *constructor* is a special function that is a member of a class and has the same name as that class. For example, here is how the **stack** class looks when converted to use a constructor for initialization:

```
// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // constructor
  void push(int i);
  int pop();
};
```

Notice that the constructor **stack( )** has no return type specified. In C++, constructors cannot return values and, thus, have no return type.

The **stack( )** constructor is coded like this:

```
// stack's constructor
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\n";
}
```

Keep in mind that the message **Stack Initialized** is output as a way to illustrate the constructor. In actual practice, most constructors will not output or input anything. They will simply perform various initializations.

An object's constructor is automatically called when the object is created. This means that it is called when the object's declaration is executed. If you are accustomed to thinking of a declaration statement as being passive, this is not the case for C++. In C++, a declaration statement is a statement that is executed. This distinction is not just academic. The code executed to construct an object may be quite significant. An object's constructor is called once for global or **static** local objects. For local objects, the constructor is called each time the object declaration is encountered.

The complement of the constructor is the *destructor*. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened. In C++, it is the destructor that handles deactivation events. The destructor has the same name as the

constructor, but it is preceded by a ~. For example, here is the **stack** class and its constructor and destructor. (Keep in mind that the **stack** class does not require a destructor; the one shown here is just for illustration.)

```
// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // constructor
  ~stack(); // destructor
  void push(int i);
  int pop();
};

// stack's constructor
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\n";
}

// stack's destructor
stack::~stack()
{
  cout << "Stack Destroyed\n";
}
```

Notice that, like constructors, destructors do not have return values.

To see how constructors and destructors work, here is a new version of the **stack** program examined earlier in this chapter. Observe that **init( )** is no longer needed.

```
// Using a constructor and destructor.
#include <iostream>
using namespace std;

#define SIZE 100

// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
```

```
  stack();  // constructor
  ~stack(); // destructor
  void push(int i);
  int pop();
};

// stack's constructor
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\n";
}

// stack's destructor
stack::~stack()
{
  cout << "Stack Destroyed\n";
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack a, b;  // create two stack objects
```

```
  a.push(1);
  b.push(2);

  a.push(3);
  b.push(4);

  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << "\n";

  return 0;
}
```

This program displays the following:

```
Stack Initialized
Stack Initialized
3 1 4 2
Stack Destroyed
Stack Destroyed
```

The C++ Keywords

There are 63 keywords currently defined for Standard C++. These are shown in Table 11-1. Together with the formal C++ syntax, they form the C++ programming language. Also, early versions of C++ defined the **overload** keyword, but it is obsolete. Keep in mind that C++ is a case-sensitive language and it requires that all keywords be in lowercase.

asm	auto	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export

Table 11-1. *The C++ keywords*

extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Table 11-1. *The C++ keywords* (continued)

The General Form of a C++ Program

Although individual styles will differ, most C++ programs will have this general form:

```
#includes
base-class declarations
derived class declarations
nonmember function prototypes
int main( )
{
    //...
}
nonmember function definitions
```

In most large projects, all **class** declarations will be put into a header file and included with each module. But the general organization of a program remains the same.

The remaining chapters in this section examine in greater detail the features discussed in this chapter, as well as all other aspects of C++.