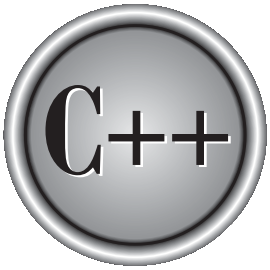
The Complete Reference



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

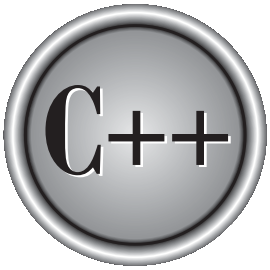
Although individual styles will differ, most C++ programs will have this general form:

```
#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++.

The Complete Reference



Chapter 12

Classes and Objects

In C++, the class forms the basis for object-oriented programming. The class is used to define the nature of an object, and it is C++'s basic unit of encapsulation. This chapter examines classes and objects in detail.

Classes

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class.

A class declaration is similar syntactically to a structure. In Chapter 11, a simplified general form of a class declaration was shown. Here is the entire general form of a **class** declaration that does not inherit any other class.

```
class class-name {
   private data and functions
access-specifier:
   data and functions
access-specifier:
   data and functions
// ...
access-specifier:
   data and functions
} object-list;
```

The *object-list* is optional. If present, it declares objects of the class. Here, *access-specifier* is one of these three C++ keywords:

public
private
protected

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The **public** access specifier allows functions or data to be accessible to other parts of your program. The **protected** access specifier is needed only when inheritance is involved (see Chapter 15). Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

You may change access specifications as often as you like within a **class** declaration. For example, you may switch to **public** for some declarations and then switch back to **private** again. The class declaration in the following example illustrates this feature:

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

class employee {
  char name[80]; // private by default
public:
  void putname(char *n); // these are public
  void getname(char *n);
private:
  double wage; // now, private again
public:
  void putwage(double w); // back to public
  double getwage();
};

void employee::putname(char *n)
{
  strcpy(name, n);
}

void employee::getname(char *n)
{
  strcpy(n, name);
}

void employee::putwage(double w)
{
  wage = w;
}

double employee::getwage()
{
  return wage;
}

int main()
{
  employee ted;
  char name[80];

  ted.putname("Ted Jones");
  ted.putwage(75000);
```

```
  ted.getname(name);
  cout << name << " makes $";
  cout << ted.getwage() << " per year.";

  return 0;
}
```

Here, **employee** is a simple class that is used to store an employee's name and wage. Notice that the **public** access specifier is used twice.

Although you may use the access specifiers as often as you like within a class declaration, the only advantage of doing so is that by visually grouping various parts of a class, you may make it easier for someone else reading the program to understand it. However, to the compiler, using multiple access specifiers makes no difference. Actually, most programmers find it easier to have only one **private**, **protected**, and **public** section within each class. For example, most programmers would code the **employee** class as shown here, with all private elements grouped together and all public elements grouped together:

```
class employee {
  char name[80];
  double wage;
public:
  void putname(char *n);
  void getname(char *n);
  void putwage(double w);
  double getwage();
};
```

Functions that are declared within a class are called *member functions*. Member functions may access any element of the class of which they are a part. This includes all **private** elements. Variables that are elements of a class are called *member variables* or *data members*. (The term *instance variable* is also used.) Collectively, any element of a class can be referred to as a member of that class.

There are a few restrictions that apply to class members. A non-**static** member variable cannot have an initializer. No member can be an object of the class that is being declared. (Although a member can be a pointer to the class that is being declared.) No member can be declared as **auto**, **extern**, or **register**.

In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved. However, there may be situations in which you will need to make one or more variables public. (For example, a heavily

used variable may need to be accessible globally in order to achieve faster run times.) When a variable is public, it may be accessed directly by any other part of your program. The syntax for accessing a public data member is the same as for calling a member function: Specify the object's name, the dot operator, and the variable name. This simple program illustrates the use of a public variable:

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

class myclass {
public:
  int i, j, k; // accessible to entire program
};

int main()
{
  myclass a, b;

  a.i = 100; // access to i, j, and k is OK
  a.j = 4;
  a.k = a.i * a.j;

  b.k = 12; // remember, a.k and b.k are different
  cout << a.k << " " << b.k;

  return 0;
}
```


Structures and Classes Are Related

Structures are part of the C subset and were inherited from the C language. As you have seen, a **class** is syntactically similar to a **struct**. But the relationship between a **class** and a **struct** is closer than you may at first think. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In fact, the only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent. That is, in C++, a *structure defines a class type*. For example, consider this short program, which uses a structure to declare a class that controls access to a string:

```
// Using a structure to define a class.
#include <iostream>
#include <cstring>
```

```
using namespace std;

struct mystr {
  void buildstr(char *s); // public
  void showstr();
private: // now go private
  char str[255];
} ;

void mystr::buildstr(char *s)
{
  if(!*s) *str = '\0'; // initialize string
  else strcat(str, s);
}

void mystr::showstr()
{
  cout << str << "\n";
}

int main()
{
  mystr s;

  s.buildstr(""); // init
  s.buildstr("Hello ");
  s.buildstr("there!");

  s.showstr();

  return 0;
}
```

This program displays the string **Hello there!**.

The class **mystr** could be rewritten by using **class** as shown here:

```
class mystr {
  char str[255];
public:
  void buildstr(char *s); // public
  void showstr();
} ;
```

You might wonder why C++ contains the two virtually equivalent keywords **struct** and **class**. This seeming redundancy is justified for several reasons. First, there is no fundamental reason not to increase the capabilities of a structure. In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to port existing C programs to C++. Finally, although **struct** and **class** are virtually equivalent today, providing two different keywords allows the definition of a **class** to be free to evolve. In order for C++ to remain compatible with C, the definition of **struct** must always be tied to its C definition.

Although you can use a **struct** where you use a **class**, most programmers don't. Usually it is best to use a **class** when you want a class, and a **struct** when you want a C-like structure. This is the style that this book will follow. Sometimes the acronym *POD* is used to describe a C-style structure—one that does not contain member functions, constructors, or destructors. It stands for Plain Old Data.

Remember *In C++, a structure declaration defines a class type.*

Unions and Classes Are Related

Like a structure, a **union** may also be used to define a class. In C++, **union**s may contain both member functions and variables. They may also include constructors and destructors. A **union** in C++ retains all of its C-like features, the most important being that all data elements share the same location in memory. Like the structure, **union** members are public by default and are fully compatible with C. In the next example, a **union** is used to swap the bytes that make up an **unsigned short** integer. (This example assumes that short integers are 2 bytes long.)

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

union swap_byte {
  void swap();
  void set_byte(unsigned short i);
  void show_word();

  unsigned short u;
  unsigned char c[2];
};

void swap_byte::swap()
{
```

```
  unsigned char t;

  t = c[0];
  c[0] = c[1];
  c[1] = t;
}

void swap_byte::show_word()
{
  cout << u;
}

void swap_byte::set_byte(unsigned short i)
{
  u = i;
}

int main()
{
  swap_byte b;

  b.set_byte(49034);
  b.swap();
  b.show_word();

  return 0;
}
```

Like a structure, a **union** declaration in C++ defines a special type of class. This means that the principle of encapsulation is preserved.

There are several restrictions that must be observed when you use C++ unions. First, a **union** cannot inherit any other classes of any type. Further, a **union** cannot be a base class. A **union** cannot have virtual member functions. (Virtual functions are discussed in Chapter 17.) No **static** variables can be members of a **union**. A reference member cannot be used. A **union** cannot have as a member any object that overloads the = operator. Finally, no object can be a member of a **union** if the object has an explicit constructor or destructor function.

As with **struct**, the term POD is also commonly applied to unions that do not contain member functions, constructors, or destructors.

Anonymous Unions

There is a special type of **union** in C++ called an *anonymous union*. An anonymous union does not include a type name, and no objects of the union can be declared.

Instead, an anonymous union tells the compiler that its member variables are to share the same location. However, the variables themselves are referred to directly, without the normal dot operator syntax. For example, consider this program:

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

int main()
{
  // define anonymous union
  union {
    long l;
    double d;
    char s[4];
  } ;

  // now, reference union elements directly
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;

  return 0;
}
```

As you can see, the elements of the union are referenced as if they had been declared as normal local variables. In fact, relative to your program, that is exactly how you will use them. Further, even though they are defined within a **union** declaration, they are at the same scope level as any other local variable within the same block. This implies that the names of the members of an anonymous union must not conflict with other identifiers known within the same scope.

All restrictions involving **union**s apply to anonymous ones, with these additions. First, the only elements contained within an anonymous union must be data. No member functions are allowed. Anonymous unions cannot contain **private** or **protected** elements. Finally, global anonymous unions must be specified as **static**.

Friend Functions

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members

of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

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

class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// Note: sum() is not a member function of any class.
int sum(myclass x)
{
  /* Because sum() is a friend of myclass, it can
     directly access a and b. */

  return x.a + x.b;
}

int main()
{
  myclass n;

  n.set_ab(3, 4);

  cout << sum(n);

  return 0;
}
```

In this example, the **sum( )** function is not a member of **myclass**. However, it still has full access to its private members. Also, notice that **sum( )** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

Although there is nothing gained by making **sum( )** a **friend** rather than a member function of **myclass**, there are some circumstances in which **friend** functions are quite valuable. First, friends can be useful when you are overloading certain types of operators (see Chapter 14). Second, **friend** functions make the creation of some types of I/O functions easier (see Chapter 18). The third reason that **friend** functions may be desirable is that in some cases, two or more classes may contain members that are interrelated relative to other parts of your program. Let's examine this third usage now.

To begin, imagine two different classes, each of which displays a pop-up message on the screen when error conditions occur. Other parts of your program may wish to know if a pop-up message is currently being displayed before writing to the screen so that no message is accidentally overwritten. Although you can create member functions in each class that return a value indicating whether a message is active, this means additional overhead when the condition is checked (that is, two function calls, not just one). If the condition needs to be checked frequently, this additional overhead may not be acceptable. However, using a function that is a **friend** of each class, it is possible to check the status of each object by calling only this one function. Thus, in situations like this, a **friend** function allows you to generate more efficient code. The following program illustrates this concept:


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

const int IDLE = 0;
const int INUSE = 1;

class C2;  // forward declaration

class C1 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

class C2 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};
```

```
void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

int idle(C1 a, C2 b)
{
  if(a.status || b.status) return 0;
  else return 1;
}

int main()
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  x.set_status(INUSE);

  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  return 0;
}
```

Notice that this program uses a *forward declaration* (also called a *forward reference*) for the class **C2**. This is necessary because the declaration of **idle( )** inside **C1** refers to **C2** before it is declared. To create a forward declaration to a class, simply use the form shown in this program.

A **friend** of one class may be a member of another. For example, here is the preceding program rewritten so that **idle( )** is a member of **C1**:

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

const int IDLE = 0;
const int INUSE = 1;

class C2;  // forward declaration

class C1 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  int idle(C2 b);  // now a member of C1
};

class C2 {
  int status;  // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

// idle() is member of C1, but friend of C2
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main()
{
```

```
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(x.idle(y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  x.set_status(INUSE);

  if(x.idle(y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  return 0;
}
```

Because **idle( )** is a member of **C1**, it can access the **status** variable of objects of type **C1** directly. Thus, only objects of type **C2** need be passed to **idle( )**.

There are two important restrictions that apply to **friend** functions. First, a derived class does not inherit **friend** functions. Second, **friend** functions may not have a storage-class specifier. That is, they may not be declared as **static** or **extern**.

Friend Classes

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.
#include <iostream>
using namespace std;

class TwoValues {
  int a;
  int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;
};

class Min {
```

```
public:
  int min(TwoValues x);
};

int Min::min(TwoValues x)
{
  return x.a < x.b ? x.a : x.b;
}

int main()
{
  TwoValues ob(10, 20);
  Min m;

  cout << m.min(ob);

  return 0;
}
```

In this example, class **Min** has access to the private variables **a** and **b** declared within the **TwoValues** class.

It is critical to understand that when one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class.

Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

Inline Functions

There is an important feature in C++, called an *inline function*, that is commonly used with classes. Since the rest of this chapter (and the rest of the book) will make heavy use of it, inline functions are examined here.

In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the **inline** keyword. For example, in this program, the function **max( )** is expanded in line instead of called:

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

```
inline int max(int a, int b)
{
  return a>b ? a : b;
}

int main()
{
  cout << max(10, 20);
  cout << " " << max(99, 88);

  return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

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

int main()
{

  cout << (10>20 ? 10 : 20);
  cout << " " << (99>88 ? 99 : 88);

  return 0;
}
```

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

Like the **register** specifier, **inline** is actually just a *request*, not a command, to the compiler. The compiler can choose to ignore it. Also, some compilers may not inline

all types of functions. For example, it is common for a compiler not to inline a recursive function. You will need to check your compiler's documentation for any restrictions to **inline.** Remember, if a function cannot be inlined, it will simply be called as a normal function.

Inline functions may be class member functions. For example, this is a perfectly valid C++ program:

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

class myclass {
  int a, b;
public:
  void init(int i, int j);
  void show();
};

// Create an inline function.
inline void myclass::init(int i, int j)
{
  a = i;
  b = j;
}

// Create another inline function.
inline void myclass::show()
{
  cout << a << " " << b << "\n";
}

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```


Note *The **inline** keyword is not part of the C subset of C++. Thus, it is not defined by C89. However, it has been added by C99.*

Defining Inline Functions Within a Class

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an **inline** function (if possible). It is not necessary (but not an error) to precede its declaration with the **inline** keyword. For example, the preceding program is rewritten here with the definitions of **init( )** and **show( )** contained within the declaration of **myclass**:

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

class myclass {
  int a, b;
public:
  // automatic inline
  void init(int i, int j) { a=i; b=j; }
  void show() { cout << a << " " << b << "\n"; }
};

int main()
{
  myclass x;

  x.init(10, 20);
  x.show();

  return 0;
}
```

Notice the format of the function code within **myclass**. Because inline functions are often short, this style of coding within a class is fairly typical. However, you are free to use any format you like. For example, this is a perfectly valid way to rewrite the **myclass** declaration:

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

class myclass {
  int a, b;
public:
  // automatic inline
  void init(int i, int j)
  {
    a = i;
```

```
    b = j;
  }

  void show()
  {
    cout << a << " " << b << "\n";
  }
};
```

Technically, the inlining of the **show( )** function is of limited value because (in general) the amount of time the I/O statement will take far exceeds the overhead of a function call. However, it is extremely common to see all short member functions defined inside their class in C++ programs. (In fact, it is rare to see short member functions defined outside their class declarations in professionally written C++ code.)

Constructor and destructor functions may also be inlined, either by default, if defined within their class, or explicitly.

Parameterized Constructors

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, here is a simple class that includes a parameterized constructor:

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

class myclass {
  int a, b;
public:
  myclass(int i, int j) {a=i; b=j;}
  void show() {cout << a << " " << b;}
};

int main()
{
  myclass ob(3, 5);

  ob.show();

  return 0;
}
```

Notice that in the definition of **myclass( ),** the parameters **i** and **j** are used to give initial values to **a** and **b**.

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor. Specifically, this statement

```
myclass ob(3, 4);
```

causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass( )**. You may also pass arguments using this type of declaration statement:

```
myclass ob = myclass(3, 4);
```

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book. Actually, there is a small technical difference between the two types of declarations that relates to copy constructors. (Copy constructors are discussed in Chapter 14.)

Here is another example that uses a parameterized constructor. It creates a class that stores information about library books.

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

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
  char author[40];
  char title[40];
  int status;
public:
  book(char *n, char *t, int s);
  int get_status() {return status;}
  void set_status(int s) {status = s;}
  void show();
};

book::book(char *n, char *t, int s)
{
  strcpy(author, n);
```

```
  strcpy(title, t);
  status = s;
}

void book::show()
{
  cout << title << " by " << author;
  cout << " is ";
  if(status==IN) cout << "in.\n";
  else cout << "out.\n";
}

int main()
{
  book b1("Twain", "Tom Sawyer", IN);
  book b2("Melville", "Moby Dick", CHECKED_OUT);

  b1.show();
  b2.show();

  return 0;
}
```


Parameterized constructors are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient. Also, notice that the short **get_status( )** and **set_status( )** functions are defined in line, within the **book** class. This is a common practice when writing C++ programs.

Constructors with One Parameter: A Special Case

If a constructor only has one parameter, there is a third way to pass an initial value to that constructor. For example, consider the following short program.

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

class X {
  int a;
public:
```

```
  X(int j) { a = j; }
  int geta() { return a; }
};

int main()
{
  X ob = 99; // passes 99 to j

  cout << ob.geta(); // outputs 99

  return 0;
}
```

Here, the constructor for **X** takes one parameter. Pay special attention to how **ob** is declared in **main( )**. In this form of initialization, 99 is automatically passed to the **j** parameter in the **X( )** constructor. That is, the declaration statement is handled by the compiler as if it were written like this:

```
X ob = X(99);
```

In general, any time you have a constructor that requires only one argument, you can use either *ob*(*i*) or *ob* = *i* to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

Remember that the alternative shown here applies only to constructors that have exactly one parameter.

Static Class Members

Both function and data members of a class can be made **static**. This section explains the consequences of each.

Static Data Members

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created.

When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

To understand the usage and effect of a **static** data member, consider this program:

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

class shared {
  static int a;
  int b;
public:
  void set(int i, int j) {a=i; b=j;}
  void show();
} ;

int shared::a; // define a

void shared::show()
{
  cout << "This is static a: " << a;
  cout << "\nThis is non-static b: " << b;
  cout << "\n";
}

int main()
{
  shared x, y;

  x.set(1, 1); // set a to 1
  x.show();

  y.set(2, 2); // change a to 2
  y.show();

  x.show(); /* Here, a has been changed for both x and y
               because a is shared by both objects. */

  return 0;
}
```

This program displays the following output when run.

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

Notice that the integer **a** is declared both inside **shared** and outside of it. As mentioned earlier, this is necessary because the declaration of **a** inside **shared** does not allocate storage.

Note

As a convenience, older versions of C++ did not require the second declaration of a ***static*** *member variable. However, this convenience gave rise to serious inconsistencies and it was eliminated several years ago. However, you may still find older C++ code that does not redeclare* ***static*** *member variables. In these cases, you will need to add the required definitions.*

A **static** member variable exists *before* any object of its class is created. For example, in the following short program, **a** is both **public** and **static**. Thus it may be directly accessed in **main( )**. Further, since **a** exists before an object of **shared** is created, **a** can be given a value at any time. As this program illustrates, the value of **a** is unchanged by the creation of object **x**. For this reason, both output statements display the same value: 99.

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

class shared {
public:
  static int a;
} ;

int shared::a; // define a

int main()
{
  // initialize a before creating any objects
  shared::a = 99;

  cout << "This is initial value of a: " << shared::a;
```

```
  cout << "\n";

  shared x;

  cout << "This is x.a: " << x.a;

  return 0;
}
```

Notice how **a** is referred to through the use of the class name and the scope resolution operator. In general, to refer to a **static** member independently of an object, you must qualify it by using the name of the class of which it is a member.

One use of a **static** member variable is to provide access control to some shared resource used by all objects of a class. For example, you might create several objects, each of which needs to write to a specific disk file. Clearly, however, only one object can be allowed to write to the file at a time. In this case, you will want to declare a **static** variable that indicates when the file is in use and when it is free. Each object then interrogates this variable before writing to the file. The following program shows how you might use a **static** variable of this type to control access to a scarce resource:

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

class cl {
  static int resource;
public:
  int get_resource();
  void free_resource() {resource = 0;}
};

int cl::resource; // define resource

int cl::get_resource()
{
  if(resource) return 0; // resource already in use
  else {
    resource = 1;
    return 1;  // resource allocated to this object
  }
}
```

```
int main()
{
  cl ob1, ob2;

  if(ob1.get_resource()) cout << "ob1 has resource\n";

  if(!ob2.get_resource()) cout << "ob2 denied resource\n";

  ob1.free_resource();  // let someone else use it

  if(ob2.get_resource())
    cout << "ob2 can now use resource\n";

  return 0;
}
```

Another interesting use of a **static** member variable is to keep track of the number of objects of a particular class type that are in existence. For example,

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

class Counter {
public:
  static int count;
  Counter() { count++; }
  ~Counter() { count--; }
};
int Counter::count;

void f();

int main(void)
{
  Counter o1;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";

  Counter o2;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
```

```
  f();
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";

  return 0;
}

void f()
{
  Counter temp;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  // temp is destroyed when f() returns
}
```

This program produces the following output.

```
Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2
```

As you can see, the **static** member variable **count** is incremented whenever an object is created and decremented when an object is destroyed. This way, it keeps track of how many objects of type **Counter** are currently in existence.

By using **static** member variables, you should be able to virtually eliminate any need for global variables. The trouble with global variables relative to OOP is that they almost always violate the principle of encapsulation.

Static Member Functions

Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.) A **static** member function does not have a **this** pointer. (See Chapter 13 for information on **this**.) There cannot be a **static** and a non-**static** version of the same function. A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile**.

Following is a slightly reworked version of the shared-resource program from the previous section. Notice that **get_resource( )** is now declared as **static**. As the program illustrates, **get_resource( )** may be called either by itself, independently of any object, by using the class name and the scope resolution operator, or in connection with an object.

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

class cl {
  static int resource;
public:
  static int get_resource();
  void free_resource() { resource = 0; }
};

int cl::resource; // define resource

int cl::get_resource()
{
  if(resource) return 0; // resource already in use
  else {
    resource = 1;
    return 1;  // resource allocated to this object
  }
}

int main()
{
  cl ob1, ob2;

  /* get_resource() is static so may be called independent
     of any object. */
  if(cl::get_resource()) cout << "ob1 has resource\n";

  if(!cl::get_resource()) cout << "ob2 denied resource\n";

  ob1.free_resource();

  if(ob2.get_resource()) // can still call using object syntax
    cout << "ob2 can now use resource\n";

  return 0;
}
```

Actually, **static** member functions have limited applications, but one good use for them is to "preinitialize" private **static** data before any object is actually created. For example, this is a perfectly valid C++ program:

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

class static_type {
  static int i;
public:
  static void init(int x) {i = x;}
  void show() {cout << i;}
};

int static_type::i; // define i

int main()
{
  // init static data before object creation
  static_type::init(100);

  static_type x;
  x.show(); // displays 100

  return 0;
}
```


When Constructors and Destructors Are Executed

As a general rule, an object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed. Precisely when these events occur is discussed here.

A local object's constructor is executed when the object's declaration statement is encountered. The destructors for local objects are executed in the reverse order of the constructor functions.

Global objects have their constructors execute *before* **main( )** begins execution. Global constructors are executed in order of their declaration, within the same file. You cannot know the order of execution of global constructors spread among several files. Global destructors execute in reverse order *after* **main( )** has terminated.

This program illustrates when constructors and destructors are executed:

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

```
class myclass {
public:
  int who;
  myclass(int id);
  ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
  cout << "Initializing " << id << "\n";
  who = id;
}

myclass::~myclass()
{
  cout << "Destructing " << who << "\n";
}

int main()
{
  myclass local_ob1(3);

  cout << "This will not be first line displayed.\n";

  myclass local_ob2(4);

  return 0;
}
```

It displays this output:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1
```

One thing: Because of differences between compilers and execution environments, you may or may not see the last two lines of output.

The Scope Resolution Operator

As you know, the **::** operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example, consider this fragment:

```
int i;  // global i

void f()
{
  int i; // local i

  i = 10; // uses local i
  .
  .
  .
}
```

As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f( )** needs to access the global version of **i**? It may do so by preceding the **i** with the **::** operator, as shown here.

```
int i;  // global i

void f()
{
  int i; // local i

  ::i = 10; // now refers to global i
  .
  .
  .
}
```

Nested Classes

It is possible to define one class within another. Doing so creates a *nested* class. Since a **class** declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class. Frankly, nested classes are seldom used. Because of C++'s flexible and powerful inheritance mechanism, the need for nested classes is virtually nonexistent.

Local Classes

A class may be defined within a function. For example, this is a valid C++ program:

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

void f();

int main()
{
  f();
  // myclass not known here
  return 0;
}

void f()
{
  class myclass {
    int i;
  public:
    void put_i(int n) { i=n; }
    int get_i() { return i; }
  } ob;

  ob.put_i(10);
  cout << ob.get_i();
}
```

When a class is declared within a function, it is known only to that function and unknown outside of it.

Several restrictions apply to local classes. First, all member functions must be defined within the class declaration. The local class may not use or access local variables of the function in which it is declared (except that a local class has access to **static** local variables declared within the function or those declared as **extern**). It may access type names and enumerators defined by the enclosing function, however. No **static** variables may be declared inside a local class. Because of these restrictions, local classes are not common in C++ programming.

Passing Objects to Functions

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by-value mechanism. Although the passing of objects is straightforward, some rather

unexpected events occur that relate to constructors and destructors. To understand why, consider this short program.

```
// Passing an object to a function.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  myclass(int n);
  ~myclass();
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

myclass::myclass(int n)
{
  i = n;
  cout << "Constructing " << i << "\n";
}

myclass::~myclass()
{
  cout << "Destroying " << i << "\n";
}

void f(myclass ob);

int main()
{
  myclass o(1);

  f(o);
  cout << "This is i in main: ";
  cout << o.get_i() << "\n";

  return 0;
}

void f(myclass ob)
{
  ob.set_i(2);
```

```
  cout << "This is local i: " << ob.get_i();
  cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

As the output shows, there is one call to the constructor, which occurs when **o** is created in **main( )**, but there are *two* calls to the destructor. Let's see why this is the case.

When an object is passed to a function, a copy of that object is made (and this copy becomes the parameter in the function). This means that a new object comes into existence. When the function terminates, the copy of the argument (i.e., the parameter) is destroyed. This raises two fundamental questions: First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answers may, at first, surprise you.

When a copy of an argument is made during a function call, the normal constructor is *not* called. Instead, the object's *copy constructor* is called. A copy constructor defines how a copy of an object is made. As explained in Chapter 14, you can explicitly define a copy constructor for a class that you create . However, if a class does not explicitly define a copy constructor, as is the case here, then C++ provides one by default. The default copy constructor creates a bitwise (that is, identical) copy of the object. The reason a bitwise copy is made is easy to understand if you think about it. Since a normal constructor is used to initialize some aspect of an object, it must not be called to make a copy of an already existing object. Such a call would alter the contents of the object. When passing an object to a function, you want to use the current state of the object, not its initial state.

However, when the function terminates and the copy of the object used as an argument is destroyed, the destructor *is* called. This is necessary because the object has gone out of scope. This is why the preceding program had two calls to the destructor. The first was when the parameter to **f( )** went out-of-scope. The second is when **o** inside **main( )** was destroyed when the program ended.

To summarize: When a copy of an object is created to be used as an argument to a function, the normal constructor is not called. Instead, the default copy constructor makes a bit-by-bit identical copy. However, when the copy is destroyed (usually by going out of scope when the function returns), the destructor is called.

Because the default copy constructor creates an exact duplicate of the original, it can, at times, be a source of trouble. Even though objects are passed to functions by means of the normal call-by-value parameter passing mechanism which, in theory,

protects and insulates the calling argument, it is still possible for a side effect to occur that may affect, or even damage, the object used as an argument. For example, if an object used as an argument allocates memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This will leave the original object damaged and effectively useless. To prevent this type of problem you will need to define the copy operation by creating a copy constructor for the class, as explained in Chapter 14.

Returning Objects

A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

myclass f();  // return object of type myclass

int main()
{
  myclass o;

  o = f();

  cout << o.get_i() << "\n";

  return 0;
}

myclass f()
{
  myclass x;

  x.set_i(1);
  return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it. There are ways to overcome this problem that involve overloading the assignment operator (see Chapter 15) and defining a copy constructor (see Chapter 14).

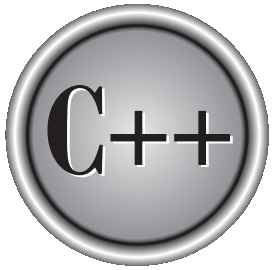
Object Assignment

Assuming that both objects are of the same type, you can assign one object to another. This causes the data of the object on the right side to be copied into the data of the object on the left. For example, this program displays **99**:

```
// Assigning objects.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i() { return i; }
};

int main()
{
  myclass ob1, ob2;

  ob1.set_i(99);
  ob2 = ob1; // assign data from ob1 to ob2

  cout << "This is ob2's i: " << ob2.get_i();

  return 0;
}
```

By default, all data from one object is assigned to the other by use of a bit-by-bit copy. However, it is possible to overload the assignment operator and define some other assignment procedure (see Chapter 15).

The Complete Reference



Chapter 13

Arrays, Pointers, References, and the Dynamic Allocation Operators

In Part One, pointers and arrays were examined as they relate to C++'s built-in types. Here, they are discussed relative to objects. This chapter also looks at a feature related to the pointer called a *reference*. The chapter concludes with an examination of C++'s dynamic allocation operators.

Arrays of Objects

In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

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

class cl {
  int i;
public:
  void set_i(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob[3];
  int i;

  for(i=0; i<3; i++) ob[i].set_i(i+1);

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

This program displays the numbers **1**, **2**, and **3** on the screen.

If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. As each element in the array is created, a value from the

list is passed to the constructor's parameter. For example, here is a slightly different version of the preceding program that uses an initialization:

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

class cl {
  int i;
public:
  cl(int j) { i=j; } // constructor
  int get_i() { return i; }
};

int main()
{
  cl ob[3] = {1, 2, 3}; // initializers
  int i;

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen.

Actually, the initialization syntax shown in the preceding program is shorthand for this longer form:

```
cl ob[3] = { cl(1), cl(2), cl(3) };
```

Here, the constructor for **cl** is invoked explicitly. Of course, the short form used in the program is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument (see Chapter 12). Thus, the short form can only be used to initialize object arrays whose constructors only require one argument.

If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example,

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

```
class cl {
  int h;
  int i;
public:
  cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
  int get_i() {return i;}
  int get_h() {return h;}
};

int main()
{
  cl ob[3] = {
    cl(1, 2), // initialize
    cl(3, 4),
    cl(5, 6)
  };

  int i;

  for(i=0; i<3; i++) {
    cout << ob[i].get_h();
    cout << ", ";
    cout << ob[i].get_i() << "\n";
  }

  return 0;
}
```

Here, **cl**'s constructor has two parameters and, therefore, requires two arguments. This means that the shorthand initialization format cannot be used and the long form, shown in the example, must be employed.

Creating Initialized vs. Uninitialized Arrays

A special case situation occurs if you intend to create both initialized and uninitialized arrays of objects. Consider the following **class.**

```
class cl {
  int i;
public:
  cl(int j) { i=j; }
```

```
  int get_i() { return i; }
};
```

Here, the constructor defined by **cl** requires one parameter. This implies that any array declared of this type must be initialized. That is, it precludes this array declaration:

```
cl a[9]; // error, constructor requires initializers
```

The reason that this statement isn't valid (as **cl** is currently defined) is that it implies that **cl** has a parameterless constructor because no initializers are specified. However, as it stands, **cl** does not have a parameterless constructor. Because there is no valid constructor that corresponds to this declaration, the compiler will report an error. To solve this problem, you need to overload the constructor, adding one that takes no parameters, as shown next. In this way, arrays that are initialized and those that are not are both allowed.

```
class cl {
  int i;
public:
  cl() { i=0; } // called for non-initialized arrays
  cl(int j) { i=j; } // called for initialized arrays
  int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:

```
cl a1[3] = {3, 5, 6}; // initialized

cl a2[34]; // uninitialized
```

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

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

```
class cl {
  int i;
public:
  cl(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob(88), *p;

  p = &ob; // get address of ob

  cout << p->get_i(); // use -> to call get_i()

  return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects. For example, this program uses a pointer to access all three elements of array **ob** after being assigned **ob**'s starting address:

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

class cl {
  int i;
public:
  cl() { i=0; }
  cl(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob[3] = {1, 2, 3};
```

```
  cl *p;
  int i;

  p = ob; // get start of array
  for(i=0; i<3; i++) {
    cout << p->get_i() << "\n";
    p++; // point to next object
  }

  return 0;
}
```


You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

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

class cl {
public:
  int i;
  cl(int j) { i=j; }
};

int main()
{
  cl ob(1);
  int *p;

  p = &ob.i; // get address of ob.i

  cout << *p; // access ob.i via p

  return 0;
}
```

Because **p** is pointing to an integer, it is declared as an integer pointer. It is irrelevant that **i** is a member of **ob** in this situation.

Type Checking C++ Pointers

There is one important thing to understand about pointers in C++: You may assign one pointer to another only if the two pointer types are compatible. For example, given:

```
int *pi;
float *pf;
```

in C++, the following assignment is illegal:

```
pi = pf; // error -- type mismatch
```

Of course, you can override any type incompatibilities using a cast, but doing so bypasses C++'s type-checking mechanism.

The this Pointer

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**. To understand **this**, first consider a program that creates a class called **pwr** that computes the result of a number raised to some power:

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

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  double get_pwr() { return val; }
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
```

```
}

int main()
{
  pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);

  cout << x.get_pwr() << " ";
  cout << y.get_pwr() << " ";
  cout << z.get_pwr() << "\n";

  return 0;
}
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **pwr( )**, the statement

```
b = base;
```

means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

```
this->b = base;
```

The **this** pointer points to the object that invoked **pwr( )**. Thus, **this –>b** refers to that object's copy of **b**. For example, if **pwr( )** had been invoked by **x** (as in **x(4.0, 2)**), then **this** in the preceding statement would have been pointing to **x**. Writing the statement without using **this** is really just shorthand.

Here is the entire **pwr( )** constructor written using the **this** pointer:

```
pwr::pwr(double base, int exp)
{
  this->b = base;
  this->e = exp;
  this->val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--)
    this->val = this->val * this->b;
}
```

Actually, no C++ programmer would write **pwr( )** as just shown because nothing is gained, and the standard form is easier. However, the **this** pointer is very important

when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it.

Remember that the **this** pointer is automatically passed to all member functions. Therefore, **get_pwr( )** could also be rewritten as shown here:

```
double get_pwr() { return this->val; }
```

In this case, if **get_pwr( )** is invoked like this:

```
y.get_pwr();
```

then **this** will point to object **y**.

Two final points about **this**. First, **friend** functions are not members of a class and, therefore, are not passed a **this** pointer. Second, **static** member functions do not have a **this** pointer.

Pointers to Derived Types

In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes. To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B *** may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type **D *** may not point to an object of type **B**. Further, although you can use a base pointer to point to a derived object, you can access only the members of the derived type that were inherited from the base. That is, you won't be able to access any members added by the derived class. (You can cast a base pointer into a derived pointer and gain full access to the entire derived class, however.)

Here is a short program that illustrates the use of a base pointer to access derived objects.

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

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
```

```
};
class derived: public base {
  int j;
public:
  void set_j(int num) { j=num; }
  int get_j() { return j; }
};

int main()
{
  base *bp;
  derived d;

  bp = &d; // base pointer points to derived object

  // access derived object using base pointer
  bp->set_i(10);
  cout << bp->get_i() << " ";

/* The following won't work. You can't access elements of
   a derived class using a base class pointer.

   bp->set_j(88); // error
   cout << bp->get_j(); // error

*/
  return 0;
}
```


As you can see, a base pointer is used to access an object of a derived class.

Although you must be careful, it is possible to cast a base pointer into a pointer of the derived type to access a member of the derived class through the base pointer. For example, this is valid C++ code:

```
// access now allowed because of cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

It is important to remember that pointer arithmetic is relative to the base type of the pointer. For this reason, when a base pointer is pointing to a derived object, incrementing the pointer does not cause it to point to the next object of the derived type. Instead, it will point to what it thinks is the next object of the base type. This,

of course, usually spells trouble. For example, this program, while syntactically correct, contains this error.

```
// This program contains an error.
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};

class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};

int main()
{
  base *bp;
  derived d[2];

  bp = d;

  d[0].set_i(1);
  d[1].set_i(2);

  cout << bp->get_i() << " ";
  bp++; // relative to base, not derived
  cout << bp->get_i(); // garbage value displayed

  return 0;
}
```

The use of base pointers to derived types is most useful when creating run-time polymorphism through the mechanism of virtual functions (see Chapter 17).

Pointers to Class Members

C++ allows you to generate a special type of pointer that "points" generically to a member of a class, not to a specific instance of that member in an object. This sort of pointer is called a *pointer to a class member* or a *pointer-to-member*, for short. A pointer to a member is not the same as a normal C++ pointer. Instead, a pointer to a member provides only an offset into an object of the member's class at which that member can be found. Since member pointers are not true pointers, the **.** and **–>** cannot be applied to them. To access a member of a class given a pointer to it, you must use the special pointer-to-member operators **.*** and **–>***. Their job is to allow you to access a member of a class given a pointer to that member.

Here is an example:

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

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val() { return val+val; }
};

int main()
{
  int cl::*data; // data member pointer
  int (cl::*func)(); // function member pointer
  cl ob1(1), ob2(2); // create objects

  data = &cl::val; // get offset of val
  func = &cl::double_val; // get offset of double_val()

  cout << "Here are values: ";
  cout << ob1.*data << " " << ob2.*data << "\n";

  cout << "Here they are doubled: ";
  cout << (ob1.*func)() << " ";
  cout << (ob2.*func)() << "\n";

  return 0;
}
```

In **main( )**, this program creates two member pointers: **data** and **func**. Note carefully the syntax of each declaration. When declaring pointers to members, you must specify the class and use the scope resolution operator. The program also creates objects of **cl** called **ob1** and **ob2**. As the program illustrates, member pointers may point to either functions or data. Next, the program obtains the addresses of **val** and **double_val( )**. As stated earlier, these "addresses" are really just offsets into an object of type **cl**, at which point **val** and **double_val( )** will be found. Next, to display the values of each object's **val**, each is accessed through **data**. Finally, the program uses **func** to call the **double_val( )** function. The extra parentheses are necessary in order to correctly associate the **.*** operator.

When you are accessing a member of an object by using an object or a reference (discussed later in this chapter), you must use the **.*** operator. However, if you are using a pointer to the object, you need to use the **–>*** operator, as illustrated in this version of the preceding program:

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

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val() { return val+val; }
};

int main()
{
  int cl::*data; // data member pointer
  int (cl::*func)(); // function member pointer
  cl ob1(1), ob2(2); // create objects
  cl *p1, *p2;

  p1 = &ob1; // access objects through a pointer
  p2 = &ob2;

  data = &cl::val; // get offset of val
  func = &cl::double_val; // get offset of double_val()

  cout << "Here are values: ";
  cout << p1->*data << " " << p2->*data << "\n";

  cout << "Here they are doubled: ";
```

```
  cout << (p1->*func)() << " ";
  cout << (p2->*func)() << "\n";

  return 0;
}
```

In this version, **p1** and **p2** are pointers to objects of type **cl**. Therefore, the **–>*** operator is used to access **val** and **double_val( ).**

Remember, pointers to members are different from pointers to specific instances of elements of an object. Consider this fragment (assume that **cl** is declared as shown in the preceding programs):

```
int cl::*d;
int *p;
cl o;

p = &o.val // this is address of a specific val

d = &cl::val // this is offset of generic val
```

Here, **p** is a pointer to an integer inside a *specific* object. However, **d** is simply an offset that indicates where **val** will be found in any object of type **cl**.

In general, pointer-to-member operators are applied in special-case situations. They are not typically used in day-to-day programming.

References

C++ contains a feature that is related to the pointer called a *reference*. A reference is essentially an implicit pointer. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference. Each is examined here.

Reference Parameters

Probably the most important use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing. As explained in Chapter 6, arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing. First, you can explicitly pass a pointer to the argument. Second,

you can use a reference parameter. For most circumstances the best way is to use a reference parameter.

To fully understand what a reference parameter is and why it is valuable, we will begin by reviewing how a call-by-reference can be generated using a pointer parameter. The following program manually creates a call-by-reference parameter using a pointer in the function called **neg( ),** which reverses the sign of the integer variable pointed to by its argument.

```
// Manually create a call-by-reference using a pointer.
#include <iostream>
using namespace std;

void neg(int *i);

int main()
{
  int x;

  x = 10;
  cout << x << " negated is ";

  neg(&x);
  cout << x << "\n";

  return 0;
}

void neg(int *i)
{
  *i = -*i;
}
```

In this program, **neg( )** takes as a parameter a pointer to the integer whose sign it will reverse. Therefore, **neg( )** must be explicitly called with the address of **x**. Further, inside **neg( )** the * operator must be used to access the variable pointed to by **i**. This is how you generate a "manual" call-by-reference in C++, and it is the only way to obtain a call-by-reference using the C subset. Fortunately, in C++ you can automate this feature by using a reference parameter.

To create a reference parameter, precede the parameter's name with an **&**. For example, here is how to declare **neg( )** with **i** declared as a reference parameter:

```
void neg(int &i);
```

For all practical purposes, this causes **i** to become another name for whatever argument **neg( )** is called with. Any operations that are applied to **i** actually affect the calling argument. In technical terms, **i** is an implicit pointer that automatically refers to the argument used in the call to **neg( )**. Once **i** has been made into a reference, it is no longer necessary (or even legal) to apply the * operator. Instead, each time **i** is used, it is implicitly a reference to the argument and any changes made to **i** affect the argument. Further, when calling **neg( ),** it is no longer necessary (or legal) to precede the argument's name with the **&** operator. Instead, the compiler does this automatically. Here is the reference version of the preceding program:

```
// Use a reference parameter.
#include <iostream>
using namespace std;

void neg(int &i); // i now a reference

int main()
{
  int x;

  x = 10;
  cout << x << " negated is ";

  neg(x); // no longer need the & operator
  cout << x << "\n";

  return 0;
}

void neg(int &i)
{
  i = -i; // i is now a reference, don't need *
}
```

To review: When you create a reference parameter, it automatically refers to (implicitly points to) the argument used to call the function. Therefore, in the preceding program, the statement

```
i = -i ;
```

actually operates on **x,** not on a copy of **x**. There is no need to apply the **&** operator to an argument. Also, inside the function, the reference parameter is used directly without

the need to apply the * operator. In general, when you assign a value to a reference, you are actually assigning that value to the variable that the reference points to.

Inside the function, it is not possible to change what the reference parameter is pointing to. That is, a statement like

```
i++:
```

inside **neg( )** increments the value of the variable used in the call. It does not cause **i** to point to some new location.

Here is another example. This program uses reference parameters to swap the values of the variables it is called with. The **swap( )** function is the classic example of call-by-reference parameter passing.

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

void swap(int &i, int &j);

int main()
{
  int a, b, c, d;

  a = 1;
  b = 2;
  c = 3;
  d = 4;

  cout << "a and b: " << a << " " << b << "\n";
  swap(a, b); // no & operator needed
  cout << "a and b: " << a << " " << b << "\n";

  cout << "c and d: " << c << " " << d << "\n";
  swap(c, d);
  cout << "c and d: " << c << " " << d << "\n";

  return 0;
}

void swap(int &i, int &j)
{
  int t;

  t = i; // no * operator needed
```

```
  i = j;
  j = t;
}
```

This program displays the following:

```
a and b: 1 2
a and b: 2 1
c and d: 3 4
c and d: 4 3
```


Passing References to Objects

In Chapter 12 it was explained that when an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called. However, when you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called. For example, try this program:

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

class cl {
  int id;
public:
  int i;
  cl(int i);
  ~cl();
  void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
{
  cout << "Constructing " << num << "\n";
  id = num;
}

cl::~cl()
{
  cout << "Destructing " << id << "\n";
}
```

```
int main()
{
  c1 o(1);

  o.i = 10;
  o.neg(o);

  cout << o.i << "\n";

  return 0;
}
```

Here is the output of this program:

```
Constructing 1
-10
Destructing 1
```

As you can see, only one call is made to **c1**'s destructor. Had **o** been passed by value, a second object would have been created inside **neg( ),** and the destructor would have been called a second time when that object was destroyed at the time **neg( )** terminated.

As the code inside **neg( )** illustrates, when you access a member of a class through a reference, you use the dot operator. The arrow operator is reserved for use with pointers only.

When passing parameters by reference, remember that changes to the object inside the function affect the calling object.

One other point: Passing all but the smallest objects by reference is faster than passing them by value. Arguments are usually passed on the stack. Thus, large objects take a considerable number of CPU cycles to push onto and pop from the stack.

Returning References

A function may return a reference. This has the rather startling effect of allowing a function to be used on the left side of an assignment statement! For example, consider this simple program:

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

char &replace(int i); // return a reference
```

```
char s[80] = "Hello There";

int main()
{

  replace(5) = 'X'; // assign X to space after Hello

  cout << s;

  return 0;
}

char &replace(int i)
{
  return s[i];
}
```


This program replaces the space between **Hello** and **There** with an **X**. That is, the program displays **HelloXthere**. Take a look at how this is accomplished. First, **replace( )** is declared as returning a reference to a character. As **replace( )** is coded, it returns a reference to the element of **s** that is specified by its argument **i.** The reference returned by **replace( )** is then used in **main( )** to assign to that element the character **X**.

One thing you must be careful about when returning references is that the object being referred to does not go out of scope after the function terminates.

Independent References

By far the most common uses for references are to pass an argument using call-by-reference and to act as a return value from a function. However, you can declare a reference that is simply a variable. This type of reference is called an *independent reference*.

When you create an independent reference, all you are creating is another name for an object. All independent references must be initialized when they are created. The reason for this is easy to understand. Aside from initialization, you cannot change what object a reference variable points to. Therefore, it must be initialized when it is declared. (In C++, initialization is a wholly separate operation from assignment.)

The following program illustrates an independent reference:

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

```
int main()
{
  int a;
  int &ref = a; // independent reference

  a = 10;
  cout << a << " " << ref << "\n";

  ref = 100;
  cout << a << " " << ref << "\n";

  int b = 19;
  ref = b; // this puts b's value into a
  cout << a << " " << ref << "\n";

  ref--; // this decrements a
         // it does not affect what ref refers to

  cout << a << " " << ref << "\n";

  return 0;
}
```

The program displays this output:

```
10 10
100 100
19 19
18 18
```

Actually, independent references are of little real value because each one is, literally, just another name for another variable. Having two names to describe the same object is likely to confuse, not organize, your program.

References to Derived Types

Similar to the situation as described for pointers earlier, a base class reference can be used to refer to an object of a derived class. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

Restrictions to References

There are a number of restrictions that apply to references. You cannot reference another reference. Put differently, you cannot obtain the address of a reference. You cannot create arrays of references. You cannot create a pointer to a reference. You cannot reference a bit-field.

A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value. Null references are prohibited.

A Matter of Style

When declaring pointer and reference variables, some C++ programmers use a unique coding style that associates the * or the **&** with the type name and not the variable. For example, here are two functionally equivalent declarations:

```
int& p; // & associated with type
int &p; // & associated with variable
```

Associating the * or **&** with the type name reflects the desire of some programmers for C++ to contain a separate pointer type. However, the trouble with associating the **&** or * with the type name rather than the variable is that, according to the formal C++ syntax, neither the **&** nor the * is distributive over a list of variables. Thus, misleading declarations are easily created. For example, the following declaration creates *one, not two*, integer pointers.

```
int* a, b;
```

Here, **b** is declared as an integer (not an integer pointer) because, as specified by the C++ syntax, when used in a declaration, the * (or **&**) is linked to the individual variable that it precedes, not to the type that it follows. The trouble with this declaration is that the visual message suggests that both **a** and **b** are pointer types, even though, in fact, only **a** is a pointer. This visual confusion not only misleads novice C++ programmers, but occasionally old pros, too.

It is important to understand that, as far as the C++ compiler is concerned, it doesn't matter whether you write **int *p** or **int* p.** Thus, if you prefer to associate the * or **&** with the type rather than the variable, feel free to do so. However, to avoid confusion, this book will continue to associate the * and the **&** with the variables that they modify rather than their types.

C++'s Dynamic Allocation Operators

C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part

of almost all real-world programs. As explained in Part One, C++ also supports dynamic memory allocation functions, called **malloc( )** and **free( ).** These are included for the sake of compatibility with C. However, for C++ code, you should use the **new** and **delete** operators because they have several advantages.

The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**. The general forms of **new** and **delete** are shown here:

p_var = new *type*;
delete *p_var*;

Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated. This exception is defined in the header **<new>**. Your program should handle this exception and take appropriate action if a failure occurs. (Exception handling is described in Chapter 19.) If this exception is not handled by your program, then your program will be terminated.

The actions of **new** on failure as just described are specified by Standard C++. The trouble is that not all compilers, especially older ones, will have implemented **new** in compliance with Standard C++. When C++ was first invented, **new** returned null on failure. Later, this was changed such that **new** caused an exception on failure. Finally, it was decided that a **new** failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, **new** has been implemented differently, at different times, by compiler manufacturers. Although all compilers will eventually implement **new** in compliance with Standard C++, currently the only way to know the precise action of **new** on failure is to check your compiler's documentation.

Since Standard C++ specifies that **new** generates an exception on failure, this is the way the code in this book is written. If your compiler handles an allocation failure differently, you will need to make the appropriate changes.

Here is a program that allocates memory to hold an integer:

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

int main()
{
  int *p;

  try {
```

```
    p = new int; // allocate space for an int
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  *p = 100;

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```


This program assigns to **p** an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements **new** such that it returns null on failure, you must change the preceding program appropriately.

The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Although **new** and **delete** perform functions similar to **malloc( )** and **free( )**, they have several advantages. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use the **sizeof** operator. Because the size is computed automatically, it eliminates any possibility for error in this regard. Second, **new** automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using **malloc( )**. Finally, both **new** and **delete** can be overloaded, allowing you to create customized allocation systems.

Although there is no formal rule that states this, it is best not to mix **new** and **delete** with **malloc( )** and **free( )** in the same program. There is no guarantee that they are mutually compatible.

Initializing Allocated Memory

You can initialize allocated memory to some known value by putting an initializer after the type name in the **new** statement. Here is the general form of **new** when an initialization is included:

p_var = new *var_type (initializer);*

Of course, the type of the initializer must be compatible with the type of data for which memory is being allocated.

This program gives the allocated integer an initial value of 87:

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

int main()
{
  int *p;

  try {
    p = new int (87); // initialize to 87
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```

Allocating Arrays

You can allocate arrays using **new** by using this general form:

p_var = new *array_type [size];*

Here, *size* specifies the number of elements in the array.

To free an array, use this form of **delete:**

delete [ ] *p_var;*

Here, the **[ ]** informs **delete** that an array is being released.

For example, the next program allocates a 10-element integer array.

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

int main()
{
  int *p, i;

  try {
    p = new int [10]; // allocate 10 integer array
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  for(i=0; i<10; i++ )
    p[i] = i;

  for(i=0; i<10; i++)
    cout << p[i] << " ";

  delete [] p; // release the array

  return 0;
}
```


Notice the **delete** statement. As just mentioned, when an array allocated by **new** is released, **delete** must be made aware that an array is being freed by using the **[ ]**. (As you will see in the next section, this is especially important when you are allocating arrays of objects.)

One restriction applies to allocating arrays: They may not be given initial values. That is, you may not specify an initializer when allocating arrays.

Allocating Objects

You can allocate objects dynamically by using **new**. When you do this, an object is created and a pointer is returned to it. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.

Here is a short program that creates a class called **balance** that links a person's name with his or her account balance. Inside **main( )**, an object of type **balance** is created dynamically.

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

class balance {
  double cur_bal;
  char name[80];
public:
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }

  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;

  try {
    p = new balance;
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->set(12387.87, "Ralph Wilson");

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
```

```
  cout << "\n";

  delete p;

  return 0;
}
```

Because **p** contains a pointer to an object, the arrow operator is used to access members of the object.

As stated, dynamically allocated objects may have constructors and destructors. Also, the constructors can be parameterized. Examine this version of the previous program:

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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  ~balance() {
    cout << "Destructing ";
    cout << name << "\n";
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
```

```
  // this version uses an initializer
  try {
    p = new balance (12387.87, "Ralph Wilson");
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

Notice that the parameters to the object's constructor are specified after the type name, just as in other sorts of initializations.

You can allocate arrays of objects, but there is one catch. Since no array allocated by **new** can have an initializer, you must make sure that if the class contains constructors, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program.

In this version of the preceding program, an array of **balance** objects is allocated, and the parameterless constructor is called.

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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  balance() {} // parameterless constructor
```

```
  ~balance() {
    cout << "Destructing ";
    cout << name << "\n";
  }
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
  int i;

  try {
    p = new balance [3]; // allocate entire array
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  // note use of dot, not arrow operators
  p[0].set(12387.87, "Ralph Wilson");
  p[1].set(144.00, "A. C. Conners");
  p[2].set(-11.23, "I. M. Overdrawn");

  for(i=0; i<3; i++) {
    p[i].get_bal(n, s);

    cout << s << "'s balance is: " << n;
    cout << "\n";
  }

  delete [] p;
  return 0;
}
```

The output from this program is shown here.

```
Ralph Wilson's balance is: 12387.9
A. C. Conners's balance is: 144
I. M. Overdrawn's balance is: -11.23
Destructing I. M. Overdrawn
Destructing A. C. Conners
Destructing Ralph Wilson
```

One reason that you need to use the **delete [ ]** form when deleting an array of dynamically allocated objects is so that the destructor can be called for each object in the array.

The nothrow Alternative

In Standard C++ it is possible to have **new** return **null** instead of throwing an exception when an allocation failure occurs. This form of **new** is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to **malloc( )** with **new**. (This is common when updating C code to C++.) This form of **new** is shown here:

p_var = new(nothrow) *type*;

Here, *p_var* is a pointer variable of *type*. The **nothrow** form of **new** works like the original version of **new** from years ago. Since it returns null on failure, it can be "dropped into" older code without having to add exception handling. However, for new code, exceptions provide a better alternative. To use the **nothrow** option, you must include the header **<new>**.

The following program shows how to use the **new(nothrow)** alternative.

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  p = new(nothrow) int[32]; // use nothrow option
  if(!p) {
    cout << "Allocation failure.\n";
```

```
    return 1;
  }

  for(i=0; i<32; i++) p[i] = i;

  for(i=0; i<32; i++) cout << p[i] << " ";

  delete [] p; // free the memory

  return 0;
}
```

As this program demonstrates, when using the **nothrow** approach, you must check the pointer returned by **new** after each allocation request.

The Placement Form of new

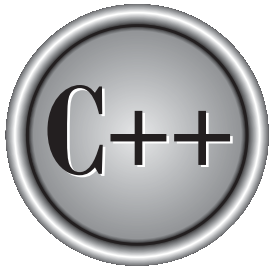
There is a special form of **new**, called the *placement form*, that can be used to specify an alternative method of allocating memory. It is primarily useful when overloading the **new** operator for special circumstances. Here is its general form:

p_var = new (*arg-list*) *type*;

Here, *arg-list* is a comma-separated list of values passed to an overloaded form of **new**.

This page intentionally left blank

The Complete Reference



Chapter 14

Function Overloading, Copy Constructors, and Default Arguments

This chapter examines function overloading, copy constructors, and default arguments. Function overloading is one of the defining aspects of the C++ programming language. Not only does it provide support for compile-time polymorphism, it also adds flexibility and convenience. Some of the most commonly overloaded functions are constructors. Perhaps the most important form of an overloaded constructor is the copy constructor. Closely related to function overloading are default arguments. Default arguments can sometimes provide an alternative to function overloading.

Function Overloading

Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation. For example, this program overloads **myfunc( )** by using different types of parameters.

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

int myfunc(int i); // these differ in types of parameters
double myfunc(double i);

int main()
{

  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(5.4); // calls myfunc(double i)

  return 0;
}

double myfunc(double i)
{
  return i;
}

int myfunc(int i)
{
  return i;
}
```

The next program overloads **myfunc( )** using a different number of parameters:

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

int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);

int main()
{
  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(4, 5); // calls myfunc(int i, int j)

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt to overload **myfunc( )**:

```
int myfunc(int i);   // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

Remember, to the compiler ***p** is the same as **p[ ]**. Therefore, although the two prototypes appear to differ in the types of their parameter, in actuality they do not.

Overloading Constructors

Constructors can be overloaded; in fact, overloaded constructors are very common. There are three main reasons why you will want to overload a constructor: to gain flexibility, to allow both initialized and uninitialized objects to be created, and to define copy constructors. In this section, the first two of these are examined. The following section describes the copy constructor.

Overloading a Constructor to Gain Flexibility

Many times you will create a class for which there are two or more possible ways to construct an object. In these cases, you will want to provide an overloaded constructor for each way. This is a self-enforcing rule because if you attempt to create an object for which there is no matching constructor, a compile-time error results.

By providing a constructor for each way that a user of your class may plausibly want to construct an object, you increase the flexibility of your class. The user is free to choose the best way to construct an object given the specific circumstance. Consider this program that creates a class called **date**, which holds a calendar date. Notice that the constructor is overloaded two ways:

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

class date {
  int day, month, year;
public:
  date(char *d);
  date(int m, int d, int y);
  void show_date();
};

// Initialize using string.
date::date(char *d)
{
  sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
}

// Initialize using integers.
date::date(int m, int d, int y)
{
```

```
  day = d;
  month = m;
  year = y;
}

void date::show_date()
{
  cout << month << "/" << day;
  cout << "/" << year << "\n";
}

int main()
{
  date ob1(12, 4, 2003), ob2("10/22/2003");

  ob1.show_date();
  ob2.show_date();

  return 0;
}
```


In this program, you can initialize an object of type **date**, either by specifying the date using three integers to represent the month, day, and year, or by using a string that contains the date in this general form:

mm/dd/yyyy

Since both are common ways to represent a date, it makes sense that **date** allow both when constructing an object.

As the **date** class illustrates, perhaps the most common reason to overload a constructor is to allow an object to be created by using the most appropriate and natural means for each particular circumstance. For example, in the following **main( )**, the user is prompted for the date, which is input to array **s**. This string can then be used directly to create **d**. There is no need for it to be converted to any other form. However, if **date( )** were not overloaded to accept the string form, you would have to manually convert it into three integers.

```
int main()
{
  char s[80];
```

```
  cout << "Enter new date: ";
  cin >> s;

  date d(s);
  d.show_date();

  return 0;
}
```

In another situation, initializing an object of type **date** by using three integers may be more convenient. For example, if the date is generated by some sort of computational method, then creating a **date** object using **date(int, int, int)** is the most natural and appropriate constructor to employ. The point here is that by overloading **date**'s constructor, you have made it more flexible and easier to use. This increased flexibility and ease of use are especially important if you are creating class libraries that will be used by other programmers.

Allowing Both Initialized and Uninitialized Objects

Another common reason constructors are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created. This is especially important if you want to be able to create dynamic arrays of objects of some class, since it is not possible to initialize a dynamically allocated array. To allow uninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

For example, the following program declares two arrays of type **powers**; one is initialized and the other is not. It also dynamically allocates an array.

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

class powers {
  int x;
public:
  // overload constructor two ways
  powers() { x = 0; } // no initializer
  powers(int n) { x = n; } // initializer

  int getx() { return x; }
  void setx(int i) { x = i; }
```

```
};

int main()
{
  powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
  powers ofThree[5]; // uninitialized
  powers *p;
  int i;

  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << ofTwo[i].getx() << " ";
  }
  cout << "\n\n";

  // set powers of three
  ofThree[0].setx(1);
  ofThree[1].setx(3);
  ofThree[2].setx(9);
  ofThree[3].setx(27);
  ofThree[4].setx(81);

  // show powers of three
  cout << "Powers of three: ";
  for(i=0; i<5; i++) {
    cout << ofThree[i].getx() << " ";
  }
  cout << "\n\n";

  // dynamically allocate an array
  try {
    p = new powers[5]; // no initialization
  } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      return 1;
  }

  // initialize dynamic array with powers of two
  for(i=0; i<5; i++) {
    p[i].setx(ofTwo[i].getx());
```

```
  }

  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << p[i].getx() << " ";
  }
  cout << "\n\n";

  delete [] p;
  return 0;
}
```

In this example, both constructors are necessary. The default constructor is used to construct the uninitialized **ofThree** array and the dynamically allocated array. The parameterized constructor is called to create the objects for the **ofTwo** array.

Copy Constructors

One of the more important forms of an overloaded constructor is the *copy constructor*. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another.

Let's begin by restating the problem that the copy constructor is designed to solve. By default, when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created. For example, assume a class called *MyClass* that allocates memory for each object when it is created, and an object *A* of that class. This means that *A* has already allocated its memory. Further, assume that *A* is used to initialize *B*, as shown here:

```
MyClass B = A;
```

If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed!

The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary

objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances.

To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy. The most common general form of a copy constructor is

```
classname (const classname &o) {
   // body of constructor
}
```

Here, *o* is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing x
func(y);       // y passed as a parameter
y = func();    // y receiving a temporary, return object
```

Following is an example where an explicit copy constructor is needed. This program creates a very limited "safe" integer array type that prevents array boundaries from being overrun. (Chapter 15 shows a better way to create a safe array that uses overloaded operators.) Storage for each array is allocated by the use of **new**, and a pointer to the memory is maintained within each array object.

```
/* This program creates a "safe" array class.  Since space
   for the array is allocated using new, a copy constructor
   is provided to allocate memory when one array object is
   used to initialize another.
*/
```

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

class array {
  int *p;
  int size;
public:
  array(int sz) {
    try {
      p = new int[sz];
    } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      exit(EXIT_FAILURE);
    }
    size = sz;
  }
  ~array() { delete [] p; }

  // copy constructor
  array(const array &a);

  void put(int i, int j) {
    if(i>=0 && i<size) p[i] = j;
  }
  int get(int i) {
    return p[i];
  }
};

// Copy Constructor
array::array(const array &a) {
  int i;

  try {
    p = new int[a.size];
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    exit(EXIT_FAILURE);
  }
  for(i=0; i<a.size; i++) p[i] = a.p[i];
}
```

```
int main()
{
  array num(10);
  int i;

  for(i=0; i<10; i++) num.put(i, i);
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // create another array and initialize with num
  array x(num); // invokes copy constructor
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```


Let's look closely at what happens when **num** is used to initialize **x** in the statement

```
array x(num); // invokes copy constructor
```

The copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that contain the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num** sharing the same memory for their arrays. (That is, **num.p** and **x.p** would have indeed pointed to the same location.)

Remember that the copy constructor is called only for initializations. For example, this sequence does not call the copy constructor defined in the preceding program:

```
array a(10);
// ...
array b(10);

b = a; // does not call copy constructor
```

In this case, **b = a** performs the assignment operation. If **=** is not overloaded (as it is not here), a bitwise copy will be made. Therefore, in some cases, you may need to overload the **=** operator as well as create a copy constructor to avoid certain types of problems (see Chapter 15).

Finding the Address of an Overloaded Function

As explained in Chapter 5, you can obtain the address of a function. One reason to do so is to assign the address of the function to a pointer and then call that function through that pointer. If the function is not overloaded, this process is straightforward. However, for overloaded functions, the process requires a little more subtlety. To understand why, first consider this statement, which assigns the address of some function called **myfunc( )** to a pointer called **p**:

```
p = myfunc;
```

If **myfunc( )** is not overloaded, there is one and only one function called **myfunc( ),** and the compiler has no difficulty assigning its address to **p**. However, if **myfunc( )** is overloaded, how does the compiler know which version's address to assign to **p**? The answer is that it depends upon how **p** is declared. For example, consider this program:

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

int myfunc(int a);
int myfunc(int a, int b);

int main()
{
  int (*fp)(int a); // pointer to int f(int)

  fp = myfunc; // points to myfunc(int)

  cout << fp(5);

  return 0;
}

int myfunc(int a)
{
  return a;
}

int myfunc(int a, int b)
{
  return a*b;
}
```

Here, there are two versions of **myfunc( )**. Both return **int**, but one takes a single integer argument; the other requires two integer arguments. In the program, **fp** is declared as a pointer to a function that returns an integer and that takes one integer argument. When **fp** is assigned the address of **myfunc( )**, C++ uses this information to select the **myfunc(int a)** version of **myfunc( )**. Had **fp** been declared like this:

```
int (*fp)(int a, int b);
```

then **fp** would have been assigned the address of the **myfunc(int a, int b)** version of **myfunc( )**.

In general, when you assign the address of an overloaded function to a function pointer, it is the declaration of the pointer that determines which function's address is obtained. Further, the declaration of the function pointer must exactly match one and only one of the overloaded function's declarations.

The overload Anachronism

When C++ was created, the keyword **overload** was required to create an overloaded function. It is obsolete and no longer used or supported. Indeed, it is not even a reserved word in Standard C++. However, because you might encounter older programs, and for its historical interest, it is a good idea to know how **overload** was used. Here is its general form:

overload *func-name*;

Here, *func-name* is the name of the function that you will be overloading. This statement must precede the overloaded declarations. For example, this tells an old-style compiler that you will be overloading a function called **test( )**:

```
overload test;
```

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. For example, this declares **myfunc( )** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
  // ...
}
```

Now, **myfunc( )** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc();        // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.

One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason.

A simple illustration of how useful a default function argument can be is shown by the **clrscr( )** function in the following program. The **clrscr( )** function clears the screen by outputting a series of linefeeds (not the most efficient way, but sufficient for this example). Because a very common video mode displays 25 lines of text, the default argument of 25 is provided. However, because some video modes display more or less than 25 lines, you can override the default argument by specifying one explicitly.

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

void clrscr(int size=25);

int main()
{
  register int i;

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(); // clears 25 lines

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(10); // clears 10 lines

  return 0;
}

void clrscr(int size)
```

```
{
   for(; size; size--) cout << endl;
}
```

As this program illustrates, when the default value is appropriate to the situation, no argument need be specified when **clrscr( )** is called. However, it is still possible to override the default and give **size** a different value when needed.

A default argument can also be used as a flag telling the function to reuse a previous argument. To illustrate this usage, a function called **iputs( )** is developed here that automatically indents a string by a specified amount. To begin, here is a version of this function that does not use a default argument:

```
void iputs(char *str, int indent)
{
  if(indent < 0) indent = 0;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

This version of **iputs( )** is called with the string to output as the first argument and the amount to indent as the second. Although there is nothing wrong with writing **iputs( )** this way, you can improve its usability by providing a default argument for the **indent** parameter that tells **iputs( )** to indent to the previously specified level. It is quite common to display a block of text with each line indented the same amount. In this situation, instead of having to supply the same **indent** argument over and over, you can give **indent** a default value that tells **iputs( )** to indent to the level of the previous call. This approach is illustrated in the following program:

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

/* Default indent to -1.  This value tells the function
   to reuse the previous value. */
void iputs(char *str, int indent = -1);

int main()
{
  iputs("Hello there", 10);
  iputs("This will be indented 10 spaces by default");
```

```
  iputs("This will be indented 5 spaces", 5);
  iputs("This is not indented", 0);

  return 0;
}

void iputs(char *str, int indent)
{
  static i = 0; // holds previous indent value

  if(indent >= 0)
    i = indent;
  else  // reuse old indent value
    indent = i;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

This program displays this output:

```
          Hello there
          This will be indented 10 spaces by default
     This will be indented 5 spaces
This is not indented
```

When you are creating functions that have default arguments, it is important to remember that the default values must be specified only once, and this must be the first time the function is declared within the file. In the preceding example, the default argument was specified in **iputs( )**'s prototype. If you try to specify new (or even the same) default values in **iputs( )**'s definition, the compiler will display an error and not compile your program. Even though default arguments for the same function cannot be redefined, you can specify different default arguments for each version of an overloaded function.

All parameters that take default values must appear to the right of those that do not. For example, it is incorrect to define **iputs( )** like this:

```
// wrong!
void iputs(int indent = -1, char *str);
```

Once you begin to define parameters that take default values, you cannot specify a nondefaulting parameter. That is, a declaration like this is also wrong and will not compile:

```
int myfunc(float f, char *str, int i=10, int j);
```

Because **i** has been given a default value, **j** must be given one too.

You can also use default parameters in an object's constructor. For example, the **cube** class shown here maintains the integer dimensions of a cube. Its constructor defaults all dimensions to zero if no other arguments are supplied, as shown here:

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

class cube {
  int x, y, z;
public:
  cube(int i=0, int j=0, int k=0) {
    x=i;
    y=j;
    z=k;
  }

  int volume() {
    return x*y*z;
  }
};

int main()
{
  cube a(2,3,4), b;

  cout << a.volume() << endl;
  cout << b.volume();

  return 0;
}
```

There are two advantages to including default arguments, when appropriate, in a constructor. First, they prevent you from having to provide an overloaded constructor that takes no parameters. For example, if the parameters to **cube( )** were not given

defaults, the second constructor shown here would be needed to handle the declaration of **b** (which specified no arguments).

```
cube() {x=0; y=0; z=0}
```

Second, defaulting common initial values is more convenient than specifying them each time an object is declared.

Default Arguments vs. Overloading

In some situations, default arguments can be used as a shorthand form of function overloading. The **cube** class's constructor just shown is one example. Let's look at another. Imagine that you want to create two customized versions of the standard **strcat( )** function. The first version will operate like **strcat( )** and concatenate the entire contents of one string to the end of another. The second version takes a third argument that specifies the number of characters to concatenate. That is, the second version will only concatenate a specified number of characters from one string to the end of another. Thus, assuming that you call your customized functions **mystrcat( )**, they will have the following prototypes:

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

The first version will copy **len** characters from **s2** to the end of **s1**. The second version will copy the entire string pointed to by **s2** onto the end of the string pointed to by **s1** and operates like **strcat( )**.

While it would not be wrong to implement two versions of **mystrcat( )** to create the two versions that you desire, there is an easier way. Using a default argument, you can create only one version of **mystrcat( )** that performs both functions. The following program demonstrates this.

```
// A customized version of strcat().
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);

int main()
{
  char str1[80] = "This is a test";
  char str2[80] = "0123456789";
```

```
  mystrcat(str1, str2, 5); // concatenate 5 chars
  cout << str1 << '\n';

  strcpy(str1, "This is a test"); // reset str1

  mystrcat(str1, str2); // concatenate entire string
  cout << str1 << '\n';

  return 0;
}

// A custom version of strcat().
void mystrcat(char *s1, char *s2, int len)
{
  // find end of s1
  while(*s1) s1++;

  if(len == -1) len = strlen(s2);

  while(*s2 && len) {
    *s1 = *s2; // copy chars
    s1++;
    s2++;
    len--;
  }

  *s1 = '\0'; // null terminate s1
}
```


Here, **mystrcat( )** concatenates up to **len** characters from the string pointed to by **s2** onto the end of the string pointed to by **s1**. However, if **len** is –1, as it will be when it is allowed to default, **mystrcat( )** concatenates the entire string pointed to by **s2** onto **s1**. (Thus, when **len** is –1, the function operates like the standard **strcat( )** function.) By using a default argument for **len**, it is possible to combine both operations into one function. In this way, default arguments sometimes provide an alternative to function overloading.

Using Default Arguments Correctly

Although default arguments can be a very powerful tool when used correctly, they can also be misused. The point of default arguments is to allow a function to perform its job in an efficient, easy-to-use manner while still allowing considerable flexibility. Toward

this end, all default arguments should reflect the way a function is generally used, or a reasonable alternate usage. When there is no single value that can be meaningfully associated with a parameter, there is no reason to declare a default argument. In fact, declaring default arguments when there is insufficient basis for doing so destructures your code, because they are liable to mislead and confuse anyone reading your program.

One other important guideline you should follow when using default arguments is this: No default argument should cause a harmful or destructive action. That is, the accidental use of a default argument should not cause a catastrophe.

Function Overloading and Ambiguity

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be *ambiguous*. Ambiguous statements are errors, and programs containing ambiguity will not compile.

By far the main cause of ambiguity involves C++'s automatic type conversions. As you know, C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function. For example, consider this fragment:

```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

As the comment indicates, this is not an error because C++ automatically converts the character **c** into its **double** equivalent. In C++, very few type conversions of this sort are actually disallowed. Although automatic type conversions are convenient, they are also a prime cause of ambiguity. For example, consider the following program:

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

float myfunc(float i);
double myfunc(double i);

int main()
{
  cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
  cout << myfunc(10); // ambiguous

  return 0;
}
```

```
float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
  return -i;
}
```

Here, **myfunc( )** is overloaded so that it can take arguments of either type **float** or type **double**. In the unambiguous line, **myfunc(double)** is called because, unless explicitly specified as **float**, all floating-point constants in C++ are automatically of type **double**. Hence, that call is unambiguous. However, when **myfunc( )** is called by using the integer 10, ambiguity is introduced because the compiler has no way of knowing whether it should be converted to a **float** or to a **double**. This causes an error message to be displayed, and the program will not compile.

As the preceding example illustrates, it is not the overloading of **myfunc( )** relative to **double** and **float** that causes the ambiguity. Rather, it is the specific call to **myfunc( )** using an indeterminate type of argument that causes the confusion. Put differently, the error is not caused by the overloading of **myfunc( )**, but by the specific invocation.

Here is another example of ambiguity caused by C++'s automatic type conversions:

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

char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
  cout << myfunc('c'); // this calls myfunc(char)
  cout << myfunc(88) << " "; // ambiguous

  return 0;
}

char myfunc(unsigned char ch)
{
  return ch-1;
```

```
}

char myfunc(char ch)
{
  return ch+1;
}
```

In C++, **unsigned char** and **char** are *not* inherently ambiguous. However, when **myfunc( )** is called by using the integer 88, the compiler does not know which function to call. That is, should 88 be converted into a **char** or an **unsigned char**?

Another way you can cause ambiguity is by using default arguments in overloaded functions. To see how, examine this program:

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

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
  cout << myfunc(4, 5) << " "; // unambiguous
  cout << myfunc(10); // ambiguous

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

Here, in the first call to **myfunc( ),** two arguments are specified; therefore, no ambiguity is introduced and **myfunc(int i, int j)** is called. However, when the second call to **myfunc( )** is made, ambiguity occurs because the compiler does not know whether to call the version of **myfunc( )** that takes one argument or to apply the default to the version that takes two arguments.

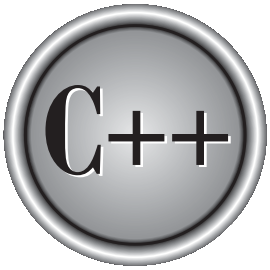
Some types of overloaded functions are simply inherently ambiguous even if, at first, they may not seem so. For example, consider this program.

```
// This program contains an error.
#include <iostream>
using namespace std;

void f(int x);
void f(int &x); // error

int main()
{
  int a=10;

  f(a); // error, which f()?

  return 0;
}

void f(int x)
{
  cout << "In f(int)\n";
}

void f(int &x)
{
  cout << "In f(int &)\n";
}
```

As the comments in the program describe, two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter. In this situation, the compiler has no way of knowing which version of the function is intended when it is called. Remember, there is no syntactical difference in the way that an argument is specified when it will be received by a reference parameter or by a value parameter.

This page intentionally left blank

The Complete Reference



Chapter 15

Operator Overloading

Closely related to function overloading is operator overloading. In C++, you can overload most operators so that they perform special operations relative to classes that you create. For example, a class that maintains a stack might overload **+** to perform a push operation and **– –** to perform a pop. When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded.

The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.

You overload operators by creating operator functions. An *operator function* defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions. Therefore, each will be examined separately, beginning with member operator functions.

Creating a Member Operator Function

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
  // operations
}
```

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The **#** is a placeholder. When you create an operator function, substitute the operator for the **#**. For example, if you are overloading the **/** operator, use **operator/.** When you are overloading a unary operator, *arg-list* will be empty. When you are overloading binary operators, *arg-list* will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.)

Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the **+** operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+( )**:

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
};

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1.show(); // displays 10 20
  ob2.show(); // displays 5 30

  ob1 = ob1 + ob2;
  ob1.show(); // displays 15 50

  return 0;
}
```

As you can see, **operator+( )** has only one parameter even though it overloads the binary **+** operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that **operator+( )** takes only one parameter is that the operand on the left side of the **+** is passed implicitly to the function through the **this** pointer. The operand on the right is passed in the parameter **op2**. The fact that the left operand is passed using **this** also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function.

As mentioned, it is common for an overloaded operator function to return an object of the class it operates upon. By doing so, it allows the operator to be used in larger expressions. For example, if the **operator+( )** function returned some other type, this expression would not have been valid:

```
ob1 = ob1 + ob2;
```

In order for the sum of **ob1** and **ob2** to be assigned to **ob1**, the outcome of that operation must be an object of type **loc**.

Further, having **operator+( )** return an object of type **loc** makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In this situation, **ob1+ob2** generates a temporary object that ceases to exist after the call to **show( )** terminates.

It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates.

One last point about the **operator+( )** function: It does not modify either operand. Because the traditional use of the **+** operator does not modify either operand, it makes sense for the overloaded version not to do so either. (For example, 5+7 yields 12, but neither 5 nor 7 is changed.) Although you are free to perform any operation you want inside an operator function, it is usually best to stay within the context of the normal use of the operator.

The next program adds three additional overloaded operators to the **loc** class: the **–,** the **=,** and the unary **++.** Pay special attention to how these functions are defined.

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

class loc {
  int longitude, latitude;
```

```
public:
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2)
{
  loc temp;

  // notice order of operands
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Overload asignment for loc.
```

```
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // i.e., return object that generated call
}

// Overload prefix ++ for loc.
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);

  ob1.show();
  ob2.show();

  ++ob1;
  ob1.show(); // displays 11 21

  ob2 = ++ob1;
  ob1.show(); // displays 12 22
  ob2.show(); // displays 12 22

  ob1 = ob2 = ob3; // multiple assignment
  ob1.show(); // displays 90 90
  ob2.show(); // displays 90 90

  return 0;
}
```

First, examine the **operator–( )** function. Notice the order of the operands in the subtraction. In keeping with the meaning of subtraction, the operand on the right side of the minus sign is subtracted from the operand on the left. Because it is the object on the left that generates the call to the **operator–( )** function, **op2**'s data must be subtracted

from the data pointed to by **this**. It is important to remember which operand generates the call to the function.

In C++, if the **=** is not overloaded, a default assignment operation is created automatically for any class you define. The default assignment is simply a member- by-member, bitwise copy. By overloading the **=**, you can define explicitly what the assignment does relative to a class. In this example, the overloaded **=** does exactly the same thing as the default, but in other situations, it could perform other operations. Notice that the **operator=( )** function returns ***this**, which is the object that generated the call. This arrangement is necessary if you want to be able to use multiple assignment operations such as this:

```
ob1 = ob2 = ob3; // multiple assignment
```

Now, look at the definition of **operator++( ).** As you can see, it takes no parameters. Since **++** is a unary operator, its only operand is implicitly passed by using the **this** pointer.

Notice that both **operator=( )** and **operator++( )** alter the value of an operand. In the case of assignment, the operand on the left (the one generating the call to the **operator=( )** function) is assigned a new value. In the case of the **++**, the operand is incremented. As stated previously, although you are free to make these operators do anything you please, it is almost always wisest to stay consistent with their original meanings.

Creating Prefix and Postfix Forms of the Increment and Decrement Operators

In the preceding program, only the prefix form of the increment operator was overloaded. However, Standard C++ allows you to explicitly create separate prefix and postfix versions of the increment or decrement operators. To accomplish this, you must define two versions of the **operator++( )** function. One is defined as shown in the foregoing program. The other is declared like this:

```
loc operator++(int x);
```

If the **++** precedes its operand, the **operator++( )** function is called. If the **++** follows its operand, the **operator++(int x)** is called and **x** has the value zero.

The preceding example can be generalized. Here are the general forms for the prefix and postfix **++** and **– –** operator functions.

```
// Prefix increment
type operator++( ) {
  // body of prefix operator
}
```

```
// Postfix increment
type operator++(int x) {
 // body of postfix operator
}

// Prefix decrement
type operator--() {
  // body of prefix operator
}

// Postfix decrement
type operator--(int x) {
 // body of postfix operator
}
```

Note *You should be careful when working with older C++ programs where the increment and decrement operators are concerned. In older versions of C++, it was not possible to specify separate prefix and postfix versions of an overloaded* **++** *or* **--**. *The prefix form was used for both.*

Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as **+=, -=,** and the like. For example, this function overloads **+=** relative to **loc**:

```
loc loc::operator+=(loc op2)
{
  longitude = op2.longitude + longitude;
  latitude = op2.latitude + latitude;

  return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

Operator Overloading Restrictions

There are some restrictions that apply to operator overloading. You cannot alter the precedence of an operator. You cannot change the number of operands that an operator takes. (You can choose to ignore an operand, however.) Except for the function call

operator (described later), operator functions cannot have default arguments. Finally, these operators cannot be overloaded:

```
. :: .* ?
```

As stated, technically you are free to perform any activity inside an operator function. For example, if you want to overload the **+** operator in such a way that it writes **I like C++** 10 times to a disk file, you can do so. However, when you stray significantly from the normal meaning of an operator, you run the risk of dangerously destructuring your program. When someone reading your program sees a statement like **Ob1+Ob2**, he or she expects something resembling addition to be taking place—not a disk access, for example. Therefore, before decoupling an overloaded operator from its normal meaning, be sure that you have sufficient reason to do so. One good example where decoupling is successful is found in the way C++ overloads the **<<** and **>>** operators for I/O. Although the I/O operations have no relationship to bit shifting, these operators provide a visual "clue" as to their meaning relative to both I/O and bit shifting, and this decoupling works. In general, however, it is best to stay within the context of the expected meaning of an operator when overloading it.

Except for the **=** operator, operator functions are inherited by a derived class. However, a derived class is free to overload any operator (including those overloaded by the base class) it chooses relative to itself.

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

In this program, the **operator+( )** function is made into a friend:

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

class loc {
  int longitude, latitude;
public:
```

```
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, loc op2); // now a friend
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2.longitude;
  temp.latitude = op1.latitude + op2.latitude;

  return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2)
{
  loc temp;

  // notice order of operands
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;

  return temp;
}

// Overload assignment for loc.
```

```
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // i.e., return object that generated call
}

// Overload ++ for loc.
loc loc::operator++()
{
  longitude++;
  latitude++;

  return *this;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30);

  ob1 = ob1 + ob2;
  ob1.show();

  return 0;
}
```

There are some restrictions that apply to friend operator functions. First, you may not overload the **=**, **( )**, **[ ]**, or **–>** operators by using a friend function. Second, as explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Using a Friend to Overload ++ or ––

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. This is because friend functions do not have **this** pointers. Assuming that you stay true to the original meaning of the **++** and – – operators, these operations imply the modification of the operand they operate upon. However, if you overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed

a **this** pointer to the operand, but rather a copy of the operand, no changes made to that parameter affect the operand that generated the call. However, you can remedy this situation by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call. For example, this program uses friend functions to overload the prefix versions of **++** and **– –** operators relative to the **loc** class:

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator=(loc op2);
  friend loc operator++(loc &op);
  friend loc operator--(loc &op);
};

// Overload assignment for loc.
loc loc::operator=(loc op2)
{
  longitude = op2.longitude;
  latitude = op2.latitude;

  return *this; // i.e., return object that generated call
}

// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
```

```
  op.longitude++;
  op.latitude++;

  return op;
}

// Make op-- a friend; use reference.
loc operator--(loc &op)
{
  op.longitude--;
  op.latitude--;

  return op;
}

int main()
{
  loc ob1(10, 20), ob2;

  ob1.show();
  ++ob1;
  ob1.show(); // displays 11 21

  ob2 =  ++ob1;
  ob2.show(); // displays 12 22

  --ob2;
  ob2.show(); // displays 11 21

  return 0;
}
```


If you want to overload the postfix versions of the increment and decrement operators using a friend, simply specify a second, dummy integer parameter. For example, this shows the prototype for the **friend,** postfix version of the increment operator relative to **loc.**

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```

Friend Operator Functions Add Flexibility

In many cases, whether you overload an operator by using a friend or a member function makes no functional difference. In those cases, it is usually best to overload by using member functions. However, there is one situation in which overloading by using a friend increases the flexibility of an overloaded operator. Let's examine this case now.

As you know, when you overload a binary operator by using a member function, the object on the left side of the operator generates the call to the operator function. Further, a pointer to that object is passed in the **this** pointer. Now, assume some class that defines a member **operator+( )** function that adds an object of the class to an integer. Given an object of that class called **Ob**, the following expression is valid:

```
Ob + 100 // valid
```

In this case, **Ob** generates the call to the overloaded **+** function, and the addition is performed. But what happens if the expression is written like this?

```
100 + Ob // invalid
```

In this case, it is the integer that appears on the left. Since an integer is a built-in type, no operation between an integer and an object of **Ob**'s type is defined. Therefore, the compiler will not compile this expression. As you can imagine, in some applications, having to always position the object on the left could be a significant burden and cause of frustration.

The solution to the preceding problem is to overload addition using a friend, not a member, function. When this is done, both arguments are explicitly passed to the operator function. Therefore, to allow both *object+integer* and *integer+object*, simply overload the function twice—one version for each situation. Thus, when you overload an operator by using two **friend** functions, the object may appear on either the left or right side of the operator.

This program illustrates how **friend** functions are used to define an operation that involves an object and built-in type:

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

class loc {
  int longitude, latitude;
public:
```

```
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, int op2);
  friend loc operator+(int op1, loc op2);
};

// + is overloaded for loc + int.
loc operator+(loc op1, int op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2;
  temp.latitude = op1.latitude + op2;

  return temp;
}
// + is overloaded for int + loc.
loc operator+(int op1, loc op2)
{
  loc temp;

  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

  ob1.show();
```

```
  ob2.show();
  ob3.show();

  ob1 = ob2 + 10; // both of these
  ob3 = 10 + ob2; // are valid

  ob1.show();
  ob3.show();

  return 0;
}
```

Overloading new and delete

It is possible to overload **new** and **delete**. You might choose to do this if you want to use some special allocation method. For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted. Whatever the reason, it is a very simple matter to overload these operators.

The skeletons for the functions that overload **new** and **delete** are shown here:

```
// Allocate an object.
void *operator new(size_t size)
{
  /* Perform allocation.  Throw bad_alloc on failure.
     Constructor called automatically. */
  return pointer_to_memory;
}

// Delete an object.
void operator delete(void *p)
{
  /* Free memory pointed to by p.
     Destructor called automatically. */
}
```

The type **size_t** is a defined type capable of containing the largest single piece of memory that can be allocated. (**size_t** is essentially an unsigned integer.) The parameter **size** will contain the number of bytes needed to hold the object being allocated. This is the amount of memory that your version of **new** must allocate. The overloaded **new** function must return a pointer to the memory that it allocates, or throw a **bad_alloc** exception if an allocation error occurs. Beyond these constraints, the overloaded **new** function can do anything else you require. When you allocate an

object using **new** (whether your own version or not), the object's constructor is automatically called.

The **delete** function receives a pointer to the region of memory to be freed. It then releases the previously allocated memory back to the system. When an object is deleted, its destructor is automatically called.

The **new** and **delete** operators may be overloaded globally so that all uses of these operators call your custom versions. They may also be overloaded relative to one or more classes. Lets begin with an example of overloading **new** and **delete** relative to a class. For the sake of simplicity, no new allocation scheme will be used. Instead, the overloaded operators will simply invoke the standard library functions **malloc( )** and **free( ).** (In your own application, you may, of course, implement any alternative allocation scheme you like.)

To overload the **new** and **delete** operators for a class, simply make the overloaded operator functions class members. For example, here the **new** and **delete** operators are overloaded for the **loc** class**:**


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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);
};

// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
  void *p;
```

```
  cout << "In overloaded new.\n";
  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
  cout << "In overloaded delete.\n";
  free(p);
}

int main()
{
  loc *p1, *p2;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
  }

  p1->show();
  p2->show();

  delete p1;
  delete p2;

  return 0;
}
```

Output from this program is shown here.

```
In overloaded new.
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

When **new** and **delete** are for a specific class, the use of these operators on any other type of data causes the original **new** or **delete** to be employed. The overloaded operators are only applied to the types for which they are defined. This means that if you add this line to the **main( ),** the default **new** will be executed:

```
int *f = new float; // uses default new
```

You can overload **new** and **delete** globally by overloading these operators outside of any class declaration. When **new** and **delete** are overloaded globally, C++'s default **new** and **delete** are ignored and the new operators are used for all allocation requests. Of course, if you have defined any versions of **new** and **delete** relative to one or more classes, then the class-specific versions are used when allocating objects of the class for which they are defined. In other words, when **new** or **delete** are encountered, the compiler first checks to see whether they are defined relative to the class they are operating on. If so, those specific versions are used. If not, C++ uses the globally defined **new** and **delete**. If these have been overloaded, the overloaded versions are used.

To see an example of overloading **new** and **delete** globally, examine this program:

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

class loc {
int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }
```

```
  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
};

// Global new
void *operator new(size_t size)
{
  void *p;

  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Global delete
void operator delete(void *p)
{
  free(p);
}

int main()
{
  loc *p1, *p2;
  float *f;

  try {
    p1 = new loc (10, 20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
```

```
  }

  try {
    f = new float; // uses overloaded new, too
  } catch (bad_alloc xa) {
    cout << "Allocation error for f.\n";
    return 1;;
  }

  *f = 10.10F;
  cout << *f << "\n";

  p1->show();
  p2->show();

  delete p1;
  delete p2;
  delete f;

  return 0;
}
```

Run this program to prove to yourself that the built-in **new** and **delete** operators have indeed been overloaded.

Overloading new and delete for Arrays

If you want to be able to allocate arrays of objects using your own allocation system, you will need to overload **new** and **delete** a second time. To allocate and free arrays, you must use these forms of **new** and **delete**.

```
// Allocate an array of objects.
void *operator new[](size_t size)
{
  /* Perform allocation.  Throw bad_alloc on failure.
     Constructor for each element called automatically. */
  return pointer_to_memory;
}

// Delete an array of objects.
void operator delete[](void *p)
```

```
{
  /* Free memory pointed to by p.
     Destructor for each element called automatically.
  */
}
```

When allocating an array, the constructor for each object in the array is automatically called. When freeing an array, each object's destructor is automatically called. You do not have to provide explicit code to accomplish these actions.

The following program allocates and frees an object and an array of objects of type **loc.**

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

class loc {
  int longitude, latitude;
public:
  loc() {longitude = latitude = 0;}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  void *operator new(size_t size);
  void operator delete(void *p);

  void *operator new[](size_t size);
  void operator delete[](void *p);
};

// new overloaded relative to loc.
void *loc::operator new(size_t size)
{
```

```
void *p;

  cout << "In overloaded new.\n";
  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
  cout << "In overloaded delete.\n";
  free(p);
}

// new overloaded for loc arrays.
void *loc::operator new[](size_t size)
{
  void *p;

  cout << "Using overload new[].\n";
  p =  malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// delete overloaded for loc arrays.
void loc::operator delete[](void *p)
{
  cout << "Freeing array using overloaded delete[]\n";
  free(p);
}

int main()
{
  loc *p1, *p2;
```

```
  int i;

  try {
    p1 = new loc (10, 20); // allocate an object
  } catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n";
    return 1;;
  }

  try {
    p2 = new loc [10]; // allocate an array
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
  }

  p1->show();

  for(i=0; i<10; i++)
    p2[i].show();

  delete p1; // free an object
  delete [] p2; // free an array

  return 0;
}
```

Overloading the nothrow Version of new and delete

You can also create overloaded **nothrow** versions of **new** and **delete**. To do so, use these skeletons.

```
// Nothrow version of new.
void *operator new(size_t size, const nothrow_t &n)
{
  // Perform allocation.
  if(success) return pointer_to_memory;
  else return 0;
}

// Nothrow version of new for arrays.
```

```
void *operator new[](size_t size, const nothrow_t &n)
{
  // Perform allocation.
  if(success) return pointer_to_memory;
  else return 0;
}

void operator delete(void *p, const nothrow_t &n)
{
  // free memory
}

void operator delete[](void *p, const nothrow_t &n)
{
  // free memory
}
```

The type **nothrow_t** is defined in **<new>**. This is the type of the **nothrow** object. The **nothrow_t** parameter is unused.

Overloading Some Special Operators

C++ defines array subscripting, function calling, and class member access as operations. The operators that perform these functions are the **[ ]**, **( )**, and **–>**, respectively. These rather exotic operators may be overloaded in C++, opening up some very interesting uses.

One important restriction applies to overloading these three operators: They must be nonstatic member functions. They cannot be **friend**s.

Overloading [ ]

In C++, the **[ ]** is considered a binary operator when you are overloading it. Therefore, the general form of a member **operator[ ]( )** function is as shown here:

```
type class-name::operator[](int i)
{
  //...
}
```

Technically, the parameter does not have to be of type **int**, but an **operator[ ]( )** function is typically used to provide array subscripting, and as such, an integer value is generally used.

Given an object called **O**, the expression

```
O[3]
```

translates into this call to the **operator[ ]( )** function:

```
O.operator[](3)
```

That is, the value of the expression within the subscripting operators is passed to the **operator[ ]( )** function in its explicit parameter. The **this** pointer will point to **O**, the object that generated the call.

In the following program, **atype** declares an array of three integers. Its constructor initializes each member of the array to the specified values. The overloaded **operator[ ]( )** function returns the value of the array as indexed by the value of its parameter.

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

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int operator[](int i) { return a[i]; }
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // displays 2

  return 0;
}
```

You can design the **operator[ ]( )** function in such a way that the **[ ]** can be used on both the left and right sides of an assignment statement. To do this, simply specify the

return value of **operator[ ]( )** as a reference. The following program makes this change and shows its use:

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

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i) { return a[i]; }
};

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // displays 2
  cout << " ";

  ob[1] = 25; // [] on left of =

  cout << ob[1]; // now displays 25

  return 0;
}
```

Because **operator[ ]( )** now returns a reference to the array element indexed by **i,** it can be used on the left side of an assignment to modify an element of the array. (Of course, it may still be used on the right side as well.)

One advantage of being able to overload the **[ ]** operator is that it allows a means of implementing safe array indexing in C++. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded **[ ]** subscripting operator, then you can intercept an out-of-range index. For example, this program adds a range check to the preceding program and proves that it works:

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;

class atype {
  int a[3];
public:
  atype(int i, int j, int k) {
    a[0] = i;
    a[1] = j;
    a[2] = k;
  }
  int &operator[](int i);
};

// Provide range checking for atype.
int &atype::operator[](int i)
{
  if(i<0 || i> 2) {
    cout << "Boundary Error\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype ob(1, 2, 3);

  cout << ob[1]; // displays 2
  cout << " ";

  ob[1] = 25; // [] appears on left
  cout << ob[1]; // displays 25

  ob[3] = 44; // generates runtime error, 3 out-of-range

  return 0;
}
```

In this program, when the statement

```
ob[3] = 44;
```

executes, the boundary error is intercepted by **operator[]( ),** and the program is terminated before any damage can be done. (In actual practice, some sort of error-handling function would be called to deal with the out-of-range condition; the program would not have to terminate.)

Overloading ( )

When you overload the **( )** function call operator, you are not, per se, creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters. Let's begin with an example. Given the overloaded operator function declaration

```
double operator()(int a, float f, char *s);
```

and an object **O** of its class, then the statement

```
O(10, 23.34, "hi");
```

translates into this call to the **operator( )** function.

```
O.operator()(10, 23.34, "hi");
```

In general, when you overload the **( )** operator, you define the parameters that you want to pass to that function. When you use the **( )** operator in your program, the arguments you specify are copied to those parameters. As always, the object that generates the call (**O** in this example) is pointed to by the **this** pointer.

Here is an example of overloading **( )** for the **loc** class. It assigns the value of its two arguments to the longitude and latitude of the object to which it is applied.

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

class loc {
  int longitude, latitude;
public:
```

```
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator()(int i, int j);
};

// Overload ( ) for loc.
loc loc::operator()(int i, int j)
{
  longitude = i;
  latitude = j;

  return *this;
}

// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;
  return temp;
}

int main()
{
  loc ob1(10, 20), ob2(1, 1);

  ob1.show();
  ob1(7, 8); // can be executed by itself
  ob1.show();
```

```
  ob1 = ob2 + ob1(10, 10); // can be used in expressions
  ob1.show();

  return 0;
}
```

The output produced by the program is shown here.

```
10 20
7 8
11 11
```

Remember, when overloading **( ),** you can use any type of parameters and return any type of value. These types will be dictated by the demands of your programs. You can also specify default arguments.

Overloading –>

The **–>** pointer operator, also called the *class member access* operator, is considered a unary operator when overloading. Its general usage is shown here:

object->element;

Here, *object* is the object that activates the call. The **operator–>( )** function must return a pointer to an object of the class that **operator–>( )** operates upon. The *element* must be some member accessible within the object.

The following program illustrates overloading the **–>** by showing the equivalence between **ob.i** and **ob–>i** when **operator–>( )** returns the **this** pointer:

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

class myclass {
public:
  int i;
  myclass *operator->() {return this;}
};

int main()
{
  myclass ob;
```

```
  ob->i = 10; // same as ob.i

  cout << ob.i << " " << ob->i;

  return 0;
}
```

An **operator–>( )** function must be a member of the class upon which it works.

Overloading the Comma Operator

You can overload C++'s comma operator. The comma is a binary operator, and like all overloaded operators, you can make an overloaded comma perform any operation you want. However, if you want the overloaded comma to perform in a fashion similar to its normal operation, then your version must discard the values of all operands except the rightmost. The rightmost value becomes the result of the comma operation. This is the way the comma works by default in C++.

Here is a program that illustrates the effect of overloading the comma operator.

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

class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  loc operator+(loc op2);
  loc operator,(loc op2);
```

```
};

// overload comma for loc
loc loc::operator,(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude;
  temp.latitude = op2.latitude;
  cout << op2.longitude << " " << op2.latitude << "\n";

  return temp;
}

// Overload + for loc
loc loc::operator+(loc op2)
{
  loc temp;

  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);

  ob1.show();
  ob2.show();
  ob3.show();
  cout << "\n";

  ob1 = (ob1, ob2+ob2, ob3);

  ob1.show(); // displays 1 1, the value of ob3

  return 0;
}
```

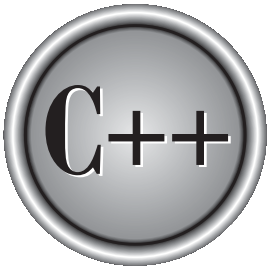
This program displays the following output:

```
10 20
5 30
1 1

10 60
1 1
1 1
```

Notice that although the values of the left-hand operands are discarded, each expression is still evaluated by the compiler so that any desired side effects will be performed.

Remember, the left-hand operand is passed via **this**, and its value is discarded by the **operator,( )** function. The value of the right-hand operation is returned by the function. This causes the overloaded comma to behave similarly to its default operation. If you want the overloaded comma to do something else, you will have to change these two features.

The Complete Reference



Chapter 16

Inheritance

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

In keeping with standard C++ terminology, a class that is inherited is referred to as a *base class*. The class that does the inheriting is called the *derived class*. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.

C++'s support of inheritance is both rich and flexible. Inheritance was introduced in Chapter 11. It is examined in detail here.

Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name {
  // body of class
};
```

The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be either **public**, **private**, or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. Let's examine the ramifications of using **public** or **private** access. (The **protected** specifier is examined in the next section.)

When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class. For example, as illustrated in this program, objects of type **derived** can directly access the public members of **base**:

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

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
```

```
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // access member of base
  ob.show(); // access member of base

  ob.showk(); // uses member of derived class

  return 0;
}
```

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class. For example, the following program will not even compile because both **set( )** and **show( )** are now private elements of **derived**:

```
// This program won't compile.
#include <iostream>
using namespace std;

class base {
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n";}
};

// Public elements of base are private in derived.
class derived : private base {
  int k;
```

```
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // error, can't access set()
  ob.show(); // error, can't access show()

  return 0;
}
```

Remember *When a base class' access specifier is **private**, public and protected members of the base become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.*

Inheritance and protected Members

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

As explained in the preceding section, a private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. Here is an example:

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

class base {
```

```
protected:
  int i, j; // private to base, but accessible by derived
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  // derived may access base's i and j
  void setk() { k=i*j; }

  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob;

  ob.set(2, 3); // OK, known to derived
  ob.show(); // OK, known to derived

  ob.setk();
  ob.showk();

  return 0;
}
```

In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected**, **derived**'s function **setk( )** may access them. If **i** and **j** had been declared as **private** by **base**, then **derived** would not have access to them, and the program would not compile.

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and **derived2** does indeed have access to **i** and **j**.

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

class base {
```

```
protected:
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// i and j inherited as protected.
class derived1 : public base {
  int k;
public:
  void setk() { k = i*j; } // legal
  void showk() { cout << k << "\n"; }
};

// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
  int m;
public:
  void setm() { m = i-j; } // legal
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(2, 3);
  ob1.show();
  ob1.setk();
  ob1.showk();

  ob2.set(3, 4);
  ob2.show();
  ob2.setk();
  ob2.setm();
  ob2.showk();
  ob2.showm();

  return 0;
}
```

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **i** and **j** would still be accessible by **derived1**.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error:

```
// This program won't compile.
#include <iostream>
using namespace std;

class base {
protected:
  int i, j;
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

// Now, all elements of base are private in derived1.
class derived1 : private base {
  int k;
public:
  // this is legal because i and j are private to derived1
  void setk() { k = i*j; } // OK
  void showk() { cout << k << "\n"; }
};

// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
  int m;
public:
  // illegal because i and j are private to derived1
  void setm() { m = i-j; } // Error
  void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(1, 2); // error, can't use set()
```

```
  ob1.show(); // error, can't use show()

  ob2.set(3, 4); // error, can't use set()
  ob2.show(); // error, can't use show()

  return 0;
}
```

Note *Even though **base** is inherited as **private** by **derived1**, **derived1** still has access to **base**'s **public** and **protected** elements. However, it cannot pass along this privilege.*

Protected Base-Class Inheritance

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class. For example,

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

class base {
protected:
  int i, j; // private to base, but accessible by derived
public:
  void setij(int a, int b) { i=a; j=b; }
  void showij() { cout << i << " " << j << "\n"; }
};

// Inherit base as protected.
class derived : protected base{
  int k;
public:
  // derived may access base's i and j and setij().
  void setk() { setij(10, 12); k = i*j; }

  // may access showij() here
  void showall() { cout << k << " "; showij(); }
};

int main()
```

```
{
  derived ob;

//  ob.setij(2, 3); // illegal, setij() is
//                     protected member of derived

  ob.setk(); // OK, public member of derived
  ob.showall(); // OK, public member of derived

// ob.showij(); // illegal, showij() is protected
//                 member of derived

  return 0;
}
```

As you can see by reading the comments, even though **setij( )** and **showij( )** are public members of **base**, they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be accessible inside **main( )**.

Inheriting Multiple Base Classes

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// An example of multiple base classes.

#include <iostream>
using namespace std;

class base1 {
protected:
  int x;
public:
  void showx() { cout << x << "\n"; }
};

class base2 {
protected:
  int y;
public:
```

```
  void showy() {cout << y << "\n";}
};

// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
  void set(int i, int j) { x=i; y=j; }
};

int main()
{
  derived ob;

  ob.set(10, 20); // provided by derived
  ob.showx(); // from base1
  ob.showy(); // from base2

  return 0;
}
```

As the example illustrates, to inherit more than one base class, use a comma-separated list. Further, be sure to use an access-specifier for each base inherited.

Constructors, Destructors, and Inheritance

There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base-class and derived-class constructors and destructors called? Second, how can parameters be passed to base-class constructors? This section examines these two important topics.

When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence. To begin, examine this short program:

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

```
class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived: public base {
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

  // do nothing but construct and destruct ob

  return 0;
}
```


As the comment in **main( )** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**. When executed, this program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As you can see, first **base**'s constructor is executed followed by **derived**'s. Next (because **ob** is immediately destroyed in this program), **derived**'s destructor is called, followed by **base**'s.

The results of the foregoing experiment can be generalized. When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor. Put differently, constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation.

If you think about it, it makes sense that constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any

initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed.

In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

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

class base {
public:
  base() { cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived1 : public base {
public:
  derived1() { cout << "Constructing derived1\n"; }
  ~derived1() { cout << "Destructing derived1\n"; }
};

class derived2: public derived1 {
public:
  derived2() { cout << "Constructing derived2\n"; }
  ~derived2() { cout << "Destructing derived2\n"; }
};

int main()
{
  derived2 ob;

  // construct and destruct ob

  return 0;
}
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes. For example, this program

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

class base1 {
public:
  base1() { cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
public:
  base2() { cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  derived() { cout << "Constructing derived\n"; }
  ~derived() { cout << "Destructing derived\n"; }
};

int main()
{
  derived ob;

  // construct and destruct ob

  return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left. This means that had **base2** been specified before **base1** in **derived**'s list, as shown here:

```
class derived: public base2, public base1 {
```

then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructors that require arguments. In cases where only the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax (see Chapter 12). However, how do you pass arguments to a constructor in a base class? The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                // ...
                                baseN(arg-list)
{
  // body of derived constructor
}
```

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes. Consider this program:

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

class base {
protected:
  int i;
public:
  base(int x) { i=x; cout << "Constructing base\n"; }
  ~base() { cout << "Destructing base\n"; }
};

class derived: public base {
  int j;
public:
  // derived uses x; y is passed along to base.
  derived(int x, int y): base(y)
    { j=x; cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // displays 4 3

  return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived( )** uses only **x; y** is passed along to **base( ).** In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.

Here is an example that uses multiple base classes:

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

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base1\n"; }
};

class derived: public base1, public base2 {
  int j;
public:
  derived(int x, int y, int z): base1(y), base2(z)
    { j=x; cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << j << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4, 5);

  ob.show(); // displays 4 3 5

  return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class

requires it. In this situation, the arguments passed to the derived class are simply passed along to the base. For example, in this program, the derived class' constructor takes no arguments, but **base1( )** and **base2( )** do:

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

class base1 {
protected:
  int i;
public:
  base1(int x) { i=x; cout << "Constructing base1\n"; }
  ~base1() { cout << "Destructing base1\n"; }
};

class base2 {
protected:
  int k;
public:
  base2(int x) { k=x; cout << "Constructing base2\n"; }
  ~base2() { cout << "Destructing base2\n"; }
};

class derived: public base1, public base2 {
public:
  /* Derived constructor uses no parameter,
     but still must be declared as taking them to
     pass them along to base classes.
  */

  derived(int x, int y): base1(x), base2(y)
    { cout << "Constructing derived\n"; }

  ~derived() { cout << "Destructing derived\n"; }
  void show() { cout << i << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4);

  ob.show(); // displays 3 4
```

```
  return 0;
}
```

A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {
  int j;
public:
  // derived uses both x and y and then passes them to base.
  derived(int x, int y): base(x, y)
    { j = x*y; cout << "Constructing derived\n"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

Granting Access

When a base class is inherited as **private**, all public and protected members of that class become private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification. For example, you might want to grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**. In Standard C++, you have two ways to accomplish this. First, you can use a **using** statement, which is the preferred way. The **using** statement is designed primarily to support namespaces and is discussed in Chapter 23. The second way to restore an inherited member's access specification is to employ an *access declaration* within the derived class. Access declarations are currently supported by Standard C++, but they are deprecated. This means that they should not be used for new code. Since there are still many, many existing programs that use access declarations, they will be examined here.

An access declaration takes this general form:

base-class::member;

The access declaration is put under the appropriate access heading in the derived class' declaration. Notice that no type declaration is required (or, indeed, allowed) in an access declaration.

To see how an access declaration works, let's begin with this short fragment:

```
class base {
public:
  int j; // public in base
};

// Inherit base as private.
class derived: private base {
public:

  // here is access declaration
  base::j; // make j public again
  .
  .
  .
};
```

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including

```
base::j;
```

as the access declaration under **derived**'s **public** heading, **j** is restored to its public status.

You can use an access declaration to restore the access rights of public and protected members. However, you cannot use an access declaration to raise or lower a member's access status. For example, a member declared as private in a base class cannot be made public by a derived class. (If C++ allowed this to occur, it would destroy its encapsulation mechanism!)

The following program illustrates the access declaration; notice how it uses access declarations to restore **j**, **seti( )**, and **geti( )** to **public** status.

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

class base {
  int i; // private to base
```

```
public:
  int j, k;
  void seti(int x) { i = x; }
  int geti() { return i; }
};

// Inherit base as private.
class derived: private base {
public:
  /* The next three statements override
     base's inheritance as private and restore j,
     seti(), and geti() to public access. */
  base::j; // make j public again - but not k
  base::seti; // make seti() public
  base::geti; // make geti() public

// base::i; // illegal, you cannot elevate access

  int a; // public
};

int main()
{
  derived ob;

//ob.i = 10; // illegal because i is private in derived

  ob.j = 20; // legal because j is made public in derived
//ob.k = 30; // illegal because k is private in derived

  ob.a = 40; // legal because a is public in derived
  ob.seti(10);

  cout << ob.geti() << " " << ob.j << " " << ob.a;

  return 0;
}
```

Access declarations are supported in C++ to accommodate those situations in which most of an inherited class is intended to be made private, but a few members are to retain their public or protected status.

While Standard C++ still supports access declarations, they are deprecated. This means that they are allowed for now, but they might not be supported in the future. Instead, the standard suggests achieving the same effect by applying the ***using*** *keyword.*

Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base.
class derived1 :  public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
```

```
  derived3 ob;

  ob.i = 10; // this is ambiguous, which i???
  ob.j = 20;
  ob.k = 30;

  // i ambiguous here, too
  ob.sum = ob.i + ob.j + ob.k;

  // also ambiguous, which i?
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like

```
ob.i = 10;
```

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base.
```

```
class derived1 :  public base {
public:
  int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This means that there are two copies of base
   in derived3! */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.derived1::i = 10; // scope resolved, use derived1's i
  ob.j = 20;
  ob.k = 30;

  // scope resolved
  ob.sum = ob.derived1::i + ob.j + ob.k;

  // also resolved here
  cout << ob.derived1::i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As you can see, because the **::** was applied, the program has manually selected **derived1**'s version of **base**. However, this solution raises a deeper issue: What if only one copy of **base** is actually required? Is there some way to prevent two copies from

being included in **derived3**? The answer, as you probably have guessed, is yes. This solution is achieved using **virtual** base classes.

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited. For example, here is another version of the example program in which **derived3** contains only one copy of **base**:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
  int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // now unambiguous
```

```
  ob.j = 20;
  ob.k = 30;

  // unambiguous
  ob.sum = ob.i + ob.j + ob.k;

  // unambiguous
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited **class**' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there is only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous.

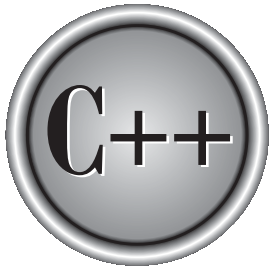
One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is still present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;

myclass.i = 88;
```

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

This page intentionally left blank

The Complete Reference



Chapter 17

Virtual Functions and Polymorphism

Polymorphism is supported by C++ both at compile time and at run time. As discussed in earlier chapters, compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions, and these are the topics of this chapter.

Virtual Functions

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. As discussed in Chapter 13, a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

To begin, examine this short example:

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
```

```
};

class derived2 : public base {
public:
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()

  return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside **base**, the virtual function **vfunc( )** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc( )** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc( )** is redefined relative to that class. Inside **main( )**, four variables are declared:

Name	Type
p	base class pointer
b	object of base
d1	object of derived1
d2	object of derived2

Next, **p** is assigned the address of **b,** and **vfunc( )** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc( )** is executed. Next, **p** is set to the address of **d1**, and again **vfunc( )** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc( )** to be executed. Finally, **p** is assigned the address of **d2**, and **p–>vfunc( )** causes the version of **vfunc( )** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc( )** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.

Although you can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved. For example, assuming the preceding example, this is syntactically valid:

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading. However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist. Perhaps the most important is that the prototype for a redefined virtual function must match exactly the prototype specified in the base class. This differs from overloading a normal function, in which return types and the number and type of parameters may differ. (In fact, when you overload a function, either the number or the type of the parameters *must* differ! It is through these differences that C++ can select the correct version of an overloaded function.) However, when a virtual function is redefined, all aspects of its prototype must be the same. If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost. Another important restriction is that virtual functions must be

nonstatic members of the classes of which they are part. They cannot be **friend**s. Finally, constructor functions cannot be virtual, but destructor functions can.

Because of the restrictions and differences between function overloading and virtual function redefinition, the term *overriding* is used to describe virtual function redefinition by a derived class.

Calling a Virtual Function Through a Base Class Reference

In the preceding example, a virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference. As explained in Chapter 13, a reference is an implicit pointer. Thus, a base-class reference can be used to refer to an object of the base class or any object derived from that base. When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.

The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter. For example, consider the following variation on the preceding program.

```
/* Here, a base class reference is used to access
   a virtual function. */
#include <iostream>
using namespace std;

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
public:
```

```
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};

// Use a base class reference parameter.
void f(base &r) {
  r.vfunc();
}

int main()
{
  base b;
  derived1 d1;
  derived2 d2;

  f(b); // pass a base object to f()
  f(d1); // pass a derived1 object to f()
  f(d2); // pass a derived2 object to f()

  return 0;
}
```

This program produces the same output as its preceding version. In this example, the function **f( )** defines a reference parameter of type **base**. Inside **main( )**, the function is called using objects of type **base**, **derived1**, and **derived2**. Inside **f( )**, the specific version of **vfunc( )** that is called is determined by the type of object being referenced when the function is called.

For the sake of simplicity, the rest of the examples in this chapter will call virtual functions through base-class pointers, but the effects are same for base-class references.

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden. Put differently, no matter how many times a virtual function is inherited, it remains virtual. For example, consider this program:

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

```
class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

/* derived2 inherits virtual function vfunc()
   from derived1. */
class derived2 : public derived1 {
public:
  // vfunc() is still virtual
  void vfunc() {
    cout << "This is derived2's vfunc().\n";
  }
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // access derived2's vfunc()
```

```
  return 0;
}
```

As expected, the preceding program displays this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

In this case, **derived2** inherits **derived1** rather than **base**, but **vfunc( )** is still virtual.

Virtual Functions Are Hierarchical

As explained, when a function is declared as **virtual** by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which **derived2** does not override **vfunc( )**:

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
```

```
public:
// vfunc() not overridden by derived2, base's is used
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // use base's vfunc()

  return 0;
}
```


The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Because **derived2** does not override **vfunc( )**, the function defined by **base** is used when **vfunc( )** is referenced relative to objects of type **derived2**.

The preceding program illustrates a special case of a more general rule. Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical. This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used. For example, in the following program, **derived2** is derived from **derived1**, which is derived from **base**. However, **derived2** does not override **vfunc( )**. This means that, relative to **derived2**,

the closest version of **vfunc( )** is in **derived1**. Therefore, it is **derived1::vfunc( )** that is used when an object of **derived2** attempts to call **vfunc( )**.

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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public derived1 {
public:
/* vfunc() not overridden by derived2.
   In this case, since derived2 is derived from
   derived1, derived1's vfunc() is used.
*/
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()
```

```
  // point to derived2
  p = &d2;
  p->vfunc(); // use derived1's vfunc()

  return 0;
}
```

The program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived1's vfunc().
```


Pure Virtual Functions

As the examples in the preceding section illustrate, when a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

A *pure virtual function* is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual *type func-name(parameter-list)* = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

The following program contains a simple example of a pure virtual function. The base class, **number**, contains an integer called **val**, the function **setval( )**, and the pure virtual function **show( )**. The derived classes **hextype**, **dectype**, and **octtype** inherit **number** and redefine **show( )** so that it outputs the value of **val** in each respective number base (that is, hexadecimal, decimal, or octal).

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

class number {
```

```
protected:
  int val;
public:
  void setval(int i) { val = i; }

  // show() is a pure virtual function
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};

class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};

class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};

int main()
{
  dectype d;
  hextype h;
  octtype o;

  d.setval(20);
  d.show(); // displays 20 - decimal

  h.setval(20);
  h.show(); // displays 14 - hexadecimal
```

```
  o.setval(20);
  o.show();  // displays 24 - octal

  return 0;
}
```

Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, **number** simply provides the common interface for the derived types to use. There is no reason to define **show( )** inside **number** since the base of the number is undefined. Of course, you can always create a placeholder definition of a virtual function. However, making **show( )** pure also ensures that all derived classes will indeed redefine it to meet their own needs.

Keep in mind that when a virtual function is declared as pure, all derived classes must override it. If a derived class fails to do this, a compile-time error will result.

Abstract Classes

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes.

Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Using Virtual Functions

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, you create a class hierarchy that moves from general to specific (base to derived). Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base

class you create and define everything you can that relates to the general case. The derived class fills in the specific details.

Following is a simple example that illustrates the value of the "one interface, multiple methods" philosophy. A class hierarchy is created that performs conversions from one system of units to another. (For example, liters to gallons.) The base class **convert** declares two variables, **val1** and **val2**, which hold the initial and converted values, respectively. It also defines the functions **getinit( )** and **getconv( )**, which return the initial value and the converted value. These elements of **convert** are fixed and applicable to all derived classes that will inherit **convert**. However, the function that will actually perform the conversion, **compute( )**, is a pure virtual function that must be defined by the classes derived from **convert**. The specific nature of **compute( )** will be determined by what type of conversion is taking place.

```
// Virtual function practical example.
#include <iostream>
using namespace std;

class convert {
protected:
  double val1;  // initial value
  double val2;  // converted value
public:
  convert(double i) {
    val1 = i;
  }
  double getconv() { return val2; }
  double getinit() { return val1; }

  virtual void compute() = 0;
};

// Liters to gallons.
class l_to_g : public convert {
public:
  l_to_g(double i) : convert(i) { }
  void compute() {
    val2 = val1 / 3.7854;
  }
};

// Fahrenheit to Celsius
class f_to_c : public convert {
```

```
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};

int main()
{
  convert *p;  // pointer to base class

  l_to_g lgob(4);
  f_to_c fcob(70);

  // use virtual function mechanism to convert
  p = &lgob;
  cout << p->getinit() << " liters is ";
  p->compute();
  cout << p->getconv() << " gallons\n";  // l_to_g

  p = &fcob;
  cout << p->getinit() << " in Fahrenheit is ";
  p->compute();
  cout << p->getconv() << " Celsius\n";  // f_to_c

  return 0;
}
```

The preceding program creates two derived classes from **convert**, called **l_to_g** and **f_to_c**. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides **compute( )** in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between **l_to_g** and **f_to_c**, the interface remains constant.

One of the benefits of derived classes and virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
class f_to_m : public convert {
public:
  f_to_m(double i) : convert(i) { }
```

```
  void compute() {
    val2 = val1 / 3.28;
  }
};
```

An important use of abstract classes and virtual functions is in *class libraries*. You can create a generic, extensible class library that will be used by other programmers. Another programmer will inherit your general class, which defines the interface and all elements common to all classes derived from it, and will add those functions specific to the derived class. By creating class libraries, you are able to create and control the interface of a general class while still letting other programmers adapt it to their specific needs.

One final point: The base class **convert** is an example of an abstract class. The virtual function **compute( )** is not defined within **convert** because no meaningful definition can be provided. The class **convert** simply does not contain sufficient information for **compute( )** to be defined. It is only when **convert** is inherited by a derived class that a complete type is created.

Early vs. Late Binding

Before concluding this chapter on virtual functions and run-time polymorphism, there are two terms that need to be defined because they are used frequently in discussions of C++ and object-oriented programming: *early binding* and *late binding*.

Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

The opposite of early binding is *late binding*. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.