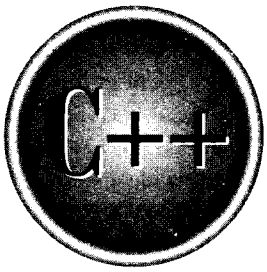
The Complete Reference



Chapter 20

The C++ I/O System Basics

C++ supports two complete I/O systems: the one inherited from C and the object-oriented I/O system defined by C++ (hereafter called simply the C++ I/O system). The C-based I/O system was discussed in Part One. Here we will begin to examine the C++ I/O system. Like C-based I/O, C++'s I/O system is fully integrated. The different aspects of C++'s I/O system, such as console I/O and disk I/O, are actually just different perspectives on the same mechanism. This chapter discusses the foundations of the C++ I/O system. Although the examples in this chapter use "console" I/O, the information is applicable to other devices, including disk files (discussed in Chapter 21).

Since the I/O system inherited from C is extremely rich, flexible, and powerful, you might be wondering why C++ defines yet another system. The answer is that C's I/O system knows nothing about objects. Therefore, for C++ to provide complete support for object-oriented programming, it was necessary to create an I/O system that could operate on user-defined objects. In addition to support for objects, there are several benefits to using C++'s I/O system even in programs that don't make extensive (or any) use of user-defined objects. Frankly, for all new code, you should use the C++ I/O system. The C I/O is supported by C++ only for compatibility.

This chapter explains how to format data, how to overload the << and >> I/O operators so they can be used with classes that you create, and how to create special I/O functions called manipulators that can make your programs more efficient.

Old vs. Modern C++ I/O

There are currently two versions of the C++ object-oriented I/O library in use: the older one that is based upon the original specifications for C++ and the newer one defined by Standard C++. The old I/O library is supported by the header file **<iostream.h>**. The new I/O library is supported by the header **<iostream>**. For the most part the two libraries appear the same to the programmer. This is because the new I/O library is, in essence, simply an updated and improved version of the old one. In fact, the vast majority of differences between the two occur beneath the surface, in the way that the libraries are implemented—not in how they are used.

From the programmer's perspective, there are two main differences between the old and new C++ I/O libraries. First, the new I/O library contains a few additional features and defines some new data types. Thus, the new I/O library is essentially a superset of the old one. Nearly all programs originally written for the old library will compile without substantive changes when the new library is used. Second, the old-style I/O library was in the global namespace. The new-style library is in the **std** namespace. (Recall that the **std** namespace is used by all of the Standard C++ libraries.) Since the old-style I/O library is now obsolete, this book describes only the new I/O library, but most of the information is applicable to the old I/O library as well.

C++ Streams

Like the C-based I/O system, the C++ I/O system operates through streams. Streams were discussed in detail in Chapter 9; that discussion will not be repeated here. However, to summarize: A *stream* is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device. For example, you can use the same function that writes to a file to write to the printer or to the screen. The advantage to this approach is that you need learn only one I/O system.

The C++ Stream Classes

As mentioned, Standard C++ provides support for its I/O system in **<iostream>**. In this header, a rather complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of template classes. As explained in Chapter 18, a template class defines the form of a class without fully specifying the data upon which it will operate. Once a template class has been defined, specific instances of it can be created. As it relates to the I/O library, Standard C++ creates two specializations of the I/O template classes: one for 8-bit characters and another for wide characters. This book will use only the 8-bit character classes since they are by far the most common. But the same techniques apply to both.

The C++ I/O system is built upon two related but different template class hierarchies. The first is derived from the low-level I/O class called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly. The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O. (A base class for **basic_ios** is called **ios_base**, which defines several nontemplate traits used by **basic_ios**.) **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively.

As explained, the I/O library creates two specializations of the template class hierarchies just described: one for 8-bit characters and one for wide characters. Here is a list of the mapping of template class names to their character and wide-character versions.

Template Class	Character-based Class	Wide-Character-based Class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

The character-based names will be used throughout the remainder of this book, since they are the names that you will normally use in your programs. They are also the same names that were used by the old I/O library. This is why the old and the new I/O library are compatible at the source code level.

One last point: The **ios** class contains many member functions and variables that control or monitor the fundamental operation of a stream. It will be referred to frequently. Just remember that if you include **<iostream>** in your program, you will have access to this important class.

C++'s Predefined Streams

When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**.

By default, the standard streams are used to communicate with the console. However, in environments that support I/O redirection (such as DOS, Unix, OS/2, and Windows), the standard streams can be redirected to other devices or files. For the sake of simplicity, the examples in this chapter assume that no I/O redirection has occurred.

Standard C++ also defines these four additional streams: **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

Formatted I/O

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. There are two related but conceptually different ways that you can format data. First, you can directly access members of the **ios** class. Specifically, you can set various format status flags defined inside the **ios** class or call various **ios** member functions. Second, you can use special functions called *manipulators* that can be included as part of an I/O expression.

We will begin the discussion of formatted I/O by using the **ios** member functions and flags.

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined. (Technically, these values are defined within **ios_base**, which, as explained earlier, is a base class for **ios**.)

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

These values are used to set or clear the format flags. If you are using an older compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer.

When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.

When the **left** flag is set, output is left justified. When **right** is set, output is right justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags are set, output is right justified by default.

By default, numeric values are output in decimal. However, it is possible to change the number base. Setting the **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.

Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.

By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.

Setting **showpos** causes a leading plus sign to be displayed before positive values.

Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.

When **unitbuf** is set, the buffer is flushed after each insertion operation.

When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.

Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the Format Flags

To set a flag, use the **setf( )** function. This function is a member of **ios**. Its most common form is shown here:

fmtflags setf(fmtflags *flags*);

This function returns the previous settings of the format flags and turns on those flags specified by *flags*. For example, to turn on the **showpos** flag, you can use this statement:

```
stream.setf(ios::showpos);
```

Here, *stream* is the stream you wish to affect. Notice the use of **ios::** to qualify **showpos**. Since **showpos** is an enumerated constant defined by the **ios** class, it must be qualified by **ios** when it is used.

The following program displays the value 100 with the **showpos** and **showpoint** flags turned on.

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

int main()
{
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);

  cout << 100.0; // displays +100.000

  return 0;
}
```

It is important to understand that **setf( )** is a member function of the **ios** class and affects streams created by that class. Therefore, any call to **setf( )** is done relative to a specific stream. There is no concept of calling **setf( )** by itself. Put differently, there is no concept in C++ of global format status. Each stream maintains its own format status information individually.

Although there is nothing technically wrong with the preceding program, there is a more efficient way to write it. Instead of making multiple calls to **setf( )**, you can simply OR together the values of the flags you want set. For example, this single call accomplishes the same thing:

```
// You can OR together two or more flags,
cout.setf(ios::showpoint | ios::showpos);
```

Remember *Because the format flags are defined within the* ***ios*** *class, you must access their values by using* ***ios*** *and the scope resolution operator. For example,* ***showbase*** *by itself will not be recognized. You must specify* ***ios::showbase****.*

Clearing Format Flags

The complement of **setf( )** is **unsetf( )**. This member function of **ios** is used to clear one or more format flags. Its general form is

void unsetf(fmtflags *flags*);

The flags specified by *flags* are cleared. (All other flags are unaffected.)

The following program illustrates **unsetf( )**. It first sets both the **uppercase** and **scientific** flags. It then outputs 100.12 in scientific notation. In this case, the "E" used

in the scientific notation is in uppercase. Next, it clears the **uppercase** flag and again outputs 100.12 in scientific notation, using a lowercase "e."

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

int main()
{
  cout.setf(ios::uppercase | ios::scientific);

  cout << 100.12;  // displays 1.001200E+02

  cout.unsetf(ios::uppercase); // clear uppercase

  cout << " \n" << 100.12; // displays 1.001200e+02

  return 0;
}
```

An Overloaded Form of setf( )

There is an overloaded form of **setf( )** that takes this general form:

fmtflags setf(fmtflags *flags1*, fmtflags *flags2*);

In this version, only the flags specified by *flags2* are affected. They are first cleared and then set according to the flags specified by *flags1*. Note that even if *flags1* contains other flags, only those specified by *flags2* will be affected. The previous flags setting is returned. For example,

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

int main( )
{
  cout.setf(ios::showpoint | ios::showpos, ios::showpoint);

  cout << 100.0; // displays 100.000, not +100.000

  return 0;
}
```

Here, **showpoint** is set, but not **showpos**, since it is not specified in the second parameter.

Perhaps the most common use of the two-parameter form of **setf( )** is when setting the number base, justification, and format flags. As explained, references to the **oct**, **dec**, and **hex** fields can collectively be referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**. Since the flags that comprise these groupings are mutually exclusive, you may need to turn off one flag when setting another. For example, the following program sets output to hexadecimal. To output in hexadecimal, some implementations require that the other number base flags be turned off in addition to turning on the **hex** flag. This is most easily accomplished using the two-parameter form of **setf( )**.

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

int main()
{
  cout.setf(ios::hex, ios::basefield);

  cout << 100; // this displays 64

  return 0;
}
```

Here, the **basefield** flags (i.,e., **dec**, **oct**, and **hex**) are first cleared and then the **hex** flag is set.

Remember, only the flags specified in *flags2* can be affected by flags specified by *flags1*. For example, in this program, the first attempt to set the **showpos** flag fails.

```
// This program will not work.
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpos, ios::hex); // error, showpos not set

  cout << 100 << '\n'; // displays 100, not +100

  cout.setf(ios::showpos, ios::showpos); // this is correct
```

```
  cout << 100; // now displays +100

  return 0;
}
```

Keep in mind that most of the time you will want to use **unsetf( )** to clear flags and the single parameter version of **setf( )** (described earlier) to set flags. The **setf(fmtflags, fmtflags)** version of **setf( )** is most often used in specialized situations, such as setting the number base. Another good use may involve a situation in which you are using a flag template that specifies the state of all format flags but wish to alter only one or two. In this case, you could specify the template in *flags1* and use *flags2* to specify which of those flags will be affected.

Examining the Formatting Flags

There will be times when you only want to know the current format settings but not alter any. To accomplish this goal, **ios** includes the member function **flags( )**, which simply returns the current setting of each format flag. Its prototype is shown here:

```
fmtflags flags( );
```

The following program uses **flags( )** to display the setting of the format flags relative to **cout**. Pay special attention to the **showflags( )** function. You might find it useful in programs you write.

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

void showflags() ;

int main()
{
  // show default condition of format flags
  showflags();

  cout.setf(ios::right | ios::showpoint | ios::fixed);

  showflags();

  return 0;
}
```

```
// This function displays the status of the format flags.
void showflags()
{
  ios::fmtflags f;
  long i;

  f = (long) cout.flags(); // get flag settings

  // check each flag
  for(i=0x4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";

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

Sample output from the program is shown here. (The precise output will vary from compiler to compiler.)

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 1 0 0 0 1 0 1 0 0 1 0 0 0 1
```

Setting All Flags

The **flags( )** function has a second form that allows you to set all format flags associated with a stream. The prototype for this version of **flags( )** is shown here:

fmtflags flags(fmtflags *f*);

When you use this version, the bit pattern found in *f* is used to set the format flags associated with the stream. Thus, all format flags are affected. The function returns the previous settings.

The next program illustrates this version of **flags( )**. It first constructs a flag mask that turns on **showpos**, **showbase**, **oct**, and **right**. All other flags are off. It then uses **flags( )** to set the format flags associated with **cout** to these settings. The function **showflags( )** verifies that the flags are set as indicated. (It is the same function used in the previous program.)

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

```
void showflags();

int main()
{
  // show default condition of format flags
  showflags();

  // showpos, showbase, oct, right are on, others off
  ios::fmtflags f = ios::showpos | ios::showbase | ios::oct | ios::right;
  cout.flags(f);  // set all flags

  showflags();

  return 0;
}
```

Using width( ), precision( ), and fill( )

In addition to the formatting flags, there are three member functions defined by **ios** that set these format parameters: the field width, the precision, and the fill character. The functions that do these things are **width( )**, **precision( )**, and **fill( )**, respectively. Each is examined in turn.

By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width( )** function. Its prototype is shown here:

streamsize width(streamsize *w*);

Here, *w* becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The **streamsize** type is defined as some form of integer by the compiler.

After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width. If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.

When outputting floating-point values, you can determine the number of digits of precision by using the **precision( )** function. Its prototype is shown here:

streamsize precision(streamsize *p*);

Here, the precision is set to *p*, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If it is not, then the default precision will be used.

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill( )** function. Its prototype is

```
char fill(char ch);
```

After a call to **fill( )**, *ch* becomes the new fill character, and the old one is returned.

Here is a program that illustrates these functions:

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

int main()
{
  cout.precision(4) ;
  cout.width(10);

  cout << 10.12345 << "\n";  // displays 10.12

  cout.fill('*');

  cout.width(10);
  cout << 10.12345 << "\n"; // displays *****10.12

  // field width applies to strings, too
  cout.width(10);
  cout << "Hi!" << "\n"; // displays *******Hi!
  cout.width(10);
  cout.setf(ios::left); // left justify
  cout << 10.12345; // displays 10.12*****

  return 0;
}
```

This program's output is shown here:

```
     10.12
*****10.12
*******Hi!
10.12*****
```

There are overloaded forms of **width( )**, **precision( )**, and **fill( )** that obtain but do not change the current setting. These forms are shown here:

```
char fill( );
streamsize width( );
streamsize precision( );
```

Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. The standard manipulators are shown in Table 20-1. As you can see by examining the table, many of the I/O manipulators parallel member functions of the **ios** class. Many of the manipulators were added recently to C++ and will not be supported by older compilers.

Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolapha** flag.	Input/Output
dec	Turns on **dec** flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on **fixed** flag.	Output
flush	Flush a stream.	Output
hex	Turns on **hex** flag.	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
nobooalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output

Table 20-1. *The C++ Manipulators*

Manipulator	Purpose	Input/Output
noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

Table 20-1. *The C++ Manipulators* (continued)

To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program.

Here is an example that uses some manipulators:

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

int main()
{
  cout << hex << 100 << endl;

  cout << setfill('?') << setw(10) << 2343.0;

  return 0;
}
```

This displays

```
64
??????2343
```

Notice how the manipulators occur within a larger I/O expression. Also notice that when a manipulator does not take an argument, such as **endl( )** in the example, it is not followed by parentheses. This is because it is the address of the function that is passed to the overloaded **<<** operator.

As a comparison, here is a functionally equivalent version of the preceding program that uses **ios** member functions to achieve the same results:

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

int main()
{
  cout.setf(ios::hex, ios::basefield);
  cout << 100 << "\n";  // 100 in hex

  cout.fill('?');
  cout.width(10);
  cout << 2343.0;

  return 0;
}
```

As the examples suggest, the main advantage of using manipulators instead of the **ios** member functions is that they often allow more compact code to be written.

You can use the **setiosflags( )** manipulator to directly set the various format flags related to a stream. For example, this program uses **setiosflags( )** to set the **showbase** and **showpos** flags:

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

int main()
{
  cout << setiosflags(ios::showpos);
  cout << setiosflags(ios::showbase);
  cout << 123 << " " << hex << 123;

  return 0;
}
```

The manipulator **setiosflags( )** performs the same function as the member function **setf( )**.

One of the more interesting manipulators is **boolapha**. It allows true and false values to be input and output using the words "true" and "false" rather than numbers. For example,

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

int main()
{
  bool b;

  b = true;
  cout << b << " " << boolalpha << b << endl;

  cout << "Enter a Boolean value: ";
  cin >> boolalpha >> b;
  cout << "Here is what you entered:  " << b;

  return 0;
}
```

Here is a sample run.

```
1 true
Enter a Boolean value: false
Here is what you entered:  false
```

Overloading << and >>

As you know, the << and the >> operators are overloaded in C++ to perform I/O operations on C++'s built-in types. You can also overload these operators so that they perform I/O operations on types that you create.

In the language of C++, the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream. The functions that overload the insertion and extraction operators are generally called *inserters* and *extractors*, respectively.

Creating Your Own Inserters

It is quite simple to create an inserter for a class that you create. All inserter functions have this general form:

```
ostream &operator<<(ostream &stream, class_type obj)
{
  // body of inserter
  return stream;
}
```

Notice that the function returns a reference to a stream of type **ostream**. (Remember, **ostream** is a class derived from **ios** that supports output.) Further, the first parameter to the function is a reference to the output stream. The second parameter is the object being inserted. (The second parameter may also be a reference to the object being inserted.) The last thing the inserter must do before exiting is return *stream*. This allows the inserter to be used in a larger I/O expression.

Within an inserter function, you may put any type of procedures or operations that you want. That is, precisely what an inserter does is completely up to you. However, for the inserter to be in keeping with good programming practices, you should limit its operations to outputting information to a stream. For example, having an inserter compute pi to 30 decimal places as a side effect to an insertion operation is probably not a very good idea!

To demonstrate a custom inserter, one will be created for objects of type **phonebook**, shown here.

```
class phonebook {
public:
```

```
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};
```

This class holds a person's name and telephone number. Here is one way to create an inserter function for objects of type **phonebook**.

```
// Display name and phone number
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}
```

Here is a short program that illustrates the **phonebook** inserter function:

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

class phonebook {
public:
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
```

```
  }
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}

int main()
{
  phonebook a("Ted", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tom", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

The program produces this output:

```
Ted (111) 555-1234
Alice (312) 555-5768
Tom (212) 555-9991
```

In the preceding program, notice that the **phonebook** inserter is not a member of **phonebook**. Although this may seem weird at first, the reason is easy to understand. When an operator function of any type is a member of a class, the left operand (passed implicitly through **this**) is the object that generates the call to the operator function. Further, this object is an *object of the class* for which the operator function is a member. There is no way to change this. If an overloaded operator function is a member of a class, the left operand must be an object of that class. However, when you overload inserters, the left operand is a *stream* and the right operand is an object of the class. Therefore, overloaded inserters cannot be members of the class for which they are overloaded. The variables **name**, **areacode**, **prefix**, and **num** are public in the preceding program so that they can be accessed by the inserter.

The fact that inserters cannot be members of the class for which they are defined seems to be a serious flaw in C++. Since overloaded inserters are not members, how can they access the private elements of a class? In the foregoing program, all members were made public. However, encapsulation is an essential component of object-oriented programming. Requiring that all data that will be output be public conflicts with this principle. Fortunately, there is a solution to this dilemma: Make the inserter a **friend** of the class. This preserves the requirement that the first argument to the overloaded inserter be a stream and still grants the function access to the private members of the class for which it is overloaded. Here is the same program modified to make the inserter into a **friend** function:

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

class phonebook {
  // now private
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}
```

```
int main()
{
  phonebook a("Ted", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tom", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

When you define the body of an inserter function, remember to keep it as general as possible. For example, the inserter shown in the preceding example can be used with any stream because the body of the function directs its output to **stream**, which is the stream that invoked the inserter. While it would not be technically wrong to have written

```
stream << o.name << " ";
```

as

```
cout << o.name << " ";
```

this would have the effect of hard-coding **cout** as the output stream. The original version will work with any stream, including those linked to disk files. Although in some situations, especially where special output devices are involved, you may want to hard-code the output stream, in most cases you will not. In general, the more flexible your inserters are, the more valuable they are.

Note *The inserter for the* ***phonebook*** *class works fine unless the value of* ***num*** *is something like 0034, in which case the preceding zeroes will not be displayed. To fix this, you can either make* ***num*** *into a string or you can set the fill character to zero and use the* ***width( )*** *format function to generate the leading zeroes. The solution is left to the reader as an exercise.*

Before moving on to extractors, let's look at one more example of an inserter function. An inserter need not be limited to handling only text. An inserter can be used to output data in any form that makes sense. For example, an inserter for some class that is part of a CAD system may output plotter instructions. Another inserter might generate graphics images. An inserter for a Windows-based program could display a dialog box. To sample the flavor of outputting things other than text, examine the following program, which draws boxes on the screen. (Because C++ does not define

a graphics library, the program uses characters to draw a box, but feel free to substitute graphics if your system supports them.)

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

class box {
  int x, y;
public:
  box(int i, int j) { x=i; y=j; }
  friend ostream &operator<<(ostream &stream, box o);
};

// Output a box.
ostream &operator<<(ostream &stream, box o)
{
  register int i, j;

  for(i=0; i<o.x; i++)
    stream << "*";

  stream << "\n";

  for(j=1; j<o.y-1; j++) {
    for(i=0; i<o.x; i++)
      if(i==0 || i==o.x-1) stream << "*";
      else stream << " ";
    stream << "\n";
  }

  for(i=0; i<o.x; i++)
     stream << "*";
  stream << "\n";

  return stream;
}

int main()
{
  box a(14, 6), b(30, 7), c(40, 5);
```

```
  cout << "Here are some boxes:\n";
  cout << a << b << c;

  return 0;
}
```

The program displays the following:

```
Here are some boxes:
**************
*             *
*             *
*             *
*             *
**************
******************************
*                            *
*                            *
*                            *
*                            *
*                            *
******************************
****************************************
*                                      *
*                                      *
*                                      *
****************************************
```

Creating Your Own Extractors

Extractors are the complement of inserters. The general form of an extractor function is

```
istream &operator>>(istream &stream, class_type &obj)
{
 // body of extractor
 return stream;
}
```

Extractors return a reference to a stream of type **istream**, which is an input stream. The first parameter must also be a reference to a stream of type **istream**. Notice that

the second parameter must be a reference to an object of the class for which the extractor is overloaded. This is so the object can be modified by the input (extraction) operation.

Continuing with the **phonebook** class, here is one way to write an extraction function:

```
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Enter name: ";
  stream >> o.name;
  cout << "Enter area code: ";
  stream >> o.areacode;
  cout << "Enter prefix: ";
  stream >> o.prefix;
  cout << "Enter number: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}
```

Notice that although this is an input function, it performs output by prompting the user. The point is that although the main purpose of an extractor is input, it can perform any operations necessary to achieve that end. However, as with inserters, it is best to keep the actions performed by an extractor directly related to input. If you don't, you run the risk of losing much in terms of structure and clarity.

Here is a program that illustrates the **phonebook** extractor:

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

class phonebook {
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook() { };
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
```

```
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}

// Input name and telephone number.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Enter name: ";
  stream >> o.name;
  cout << "Enter area code: ";
  stream >> o.areacode;
  cout << "Enter prefix: ";
  stream >> o.prefix;
  cout << "Enter number: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}

int main()
{
  phonebook a;

  cin >> a;

  cout << a;

  return 0;
}
```

Actually, the extractor for **phonebook** is less than perfect because the **cout** statements are needed only if the input stream is connected to an interactive device such as the console (that is, when the input stream is **cin**). If the extractor is used on a stream connected to a disk file, for example, then the **cout** statements would not be applicable. For fun, you might want to try suppressing the **cout** statements except when the input stream refers to **cin**. For example, you might use **if** statements such as the one shown here.

```
if(stream == cin) cout << "Enter name: ";
```

Now, the prompt will take place only when the output device is most likely the screen.

Creating Your Own Manipulator Functions

In addition to overloading the insertion and extraction operators, you can further customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons. First, you can consolidate a sequence of several separate I/O operations into one manipulator. For example, it is not uncommon to have situations in which the same sequence of I/O operations occurs frequently within a program. In these cases you can use a custom manipulator to perform these actions, thus simplifying your source code and preventing accidental errors. A custom manipulator can also be important when you need to perform I/O operations on a nonstandard device. For example, you might use a manipulator to send control codes to a special type of printer or to an optical recognition system.

Custom manipulators are a feature of C++ that supports OOP, but also can benefit programs that aren't object oriented. As you will see, custom manipulators can help make any I/O-intensive program clearer and more efficient.

As you know, there are two basic types of manipulators: those that operate on input streams and those that operate on output streams. In addition to these two broad categories, there is a secondary division: those manipulators that take an argument and those that don't. Frankly, the procedures necessary to create a parameterized manipulator vary widely from compiler to compiler, and even between two different versions of the same compiler. For this reason, you must consult the documentation to your compiler for instructions on creating parameterized manipulators. However, the creation of parameterless manipulators is straightforward and the same for all compilers. It is described here.

All parameterless manipulator output functions have this skeleton:

```
ostream &manip-name(ostream &stream)
{
  // your code here
  return stream;
}
```

Here, *manip-name* is the name of the manipulator. Notice that a reference to a stream of type **ostream** is returned. This is necessary if a manipulator is used as part of a larger I/O expression. It is important to note that even though the manipulator has as its single argument a reference to the stream upon which it is operating, no argument is used when the manipulator is inserted in an output operation.

As a simple first example, the following program creates a manipulator called **sethex( )**, which turns on the **showbase** flag and sets output to hexadecimal.

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

// A simple output manipulator.
ostream &sethex(ostream &stream)
{
  stream.setf(ios::showbase);
  stream.setf(ios::hex, ios::basefield);

  return stream;
}

int main()
{
  cout << 256 << " " << sethex << 256;

  return 0;
}
```

This program displays **256 0x100**. As you can see, **sethex** is used as part of an I/O expression in the same way as any of the built-in manipulators.

Custom manipulators need not be complex to be useful. For example, the simple manipulators **la( )** and **ra( )** display a left and right arrow for emphasis, as shown here:

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

// Right Arrow
ostream &ra(ostream &stream)
{
  stream << "-------> ";
  return stream;
```

```
}

// Left Arrow
ostream &la(ostream &stream)
{
  stream << " <-------";
  return stream;
}

int main()
{
  cout << "High balance " << ra << 1233.23 << "\n";
  cout << "Over draft " << ra << 567.66 << la;

  return 0;
}
```

This program displays:

```
High balance -------> 1233.23
Over draft -------> 567.66 <-------
```

If used frequently, these simple manipulators save you from some tedious typing.

Using an output manipulator is particularly useful for sending special codes to a device. For example, a printer may be able to accept various codes that change the type size or font, or that position the print head in a special location. If these adjustments are going to be made frequently, they are perfect candidates for a manipulator.

All parameterless input manipulator functions have this skeleton:

```
istream &manip-name(istream &stream)
{
  // your code here
  return stream;
}
```

An input manipulator receives a reference to the stream for which it was invoked. This stream must be returned by the manipulator.

The following program creates the **getpass( )** input manipulator, which rings the bell and then prompts for a password:

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

// A simple input manipulator.
istream &getpass(istream &stream)
{
  cout << '\a';  // sound bell
  cout << "Enter password: ";

  return stream;
}

int main()
{
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while(strcmp(pw, "password"));

  cout << "Logon complete\n";

  return 0;
}
```

Remember that it is crucial that your manipulator return **stream**. If it does not, your manipulator cannot be used in a series of input or output operations.