Module 11

The C++ I/O System

Table of Contents

Since the beginning of this book you have been using the C++ I/O system, but you have been doing so without much formal explanation. Since the I/O system is based upon a hierarchy of classes, it was not possible to present its theory and details without first discussing classes and inheritance. Now it is time to examine the C++ I/O system in detail. The C++ I/O system is quite large, and it won't be possible to discuss here every class, function, or feature, but this module will introduce you to the most important and commonly used parts. Specifically, it shows how to overload the << and >> operators so that you can input or output objects of classes that you design. It describes how to format output and how to use I/O manipulators. The module ends by discussing file I/O.

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.

CRITICAL SKILL 11.1: C++ Streams

The most fundamental point to understand about the C++ I/O system is that it operates on streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the same I/O functions and operators can operate on virtually any type of device. For example, the same method that you use to write to the screen can be used to write to a disk or to the printer.

In its most common form, a stream is a logical interface to a file. As C++ defines the term "file," it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on. Although files differ in form and capabilities, all streams are the same. The advantage to this approach is that to you, the programmer, one hardware device will look much like any other. The stream provides a consistent interface.

A stream is linked to a file through an open operation. A stream is disassociated from a file through a close operation.

There are two types of streams: text and binary. A text stream is used with characters. When a text stream is being used, some character translations may take place. For example, when the newline character is output, it may be converted into a carriage return–linefeed sequence. For this reason, there might not be a one-to-one correspondence between what is sent to the stream and what is written to the file. A binary stream can be used with any type of data. No character translations will occur, and there is a one-to-one correspondence between what is sent to the stream and what is actually contained in the file.

One more concept to understand is that of the current location. The current location (also referred to as the current position) is the location in a file where the next file access will occur. For example, if a file is 100 bytes long and half the file has been read, the next read operation will occur at byte 50, which is the current location.

To summarize: In C++, I/O is performed through a logical interface called a stream. All streams have similar properties, and every stream is operated upon by the same I/O functions, no matter what type of file it is associated with. A file is the actual physical entity that contains

The C++ I/O System

the data. Even though files differ, streams do not. (Of course, some devices may not support all operations, such as random-access operations, so their associated streams will not support these operations either.)

The C++ Predefined Streams

C++ contains several predefined streams that are automatically opened when your C++ program begins execution. They are cin, cout, cerr, and clog. As you know, cin is the stream associated with standard input, and cout is the stream associated with standard output. The cerr stream is linked to standard output, and so is clog. The difference between these two streams is that clog is buffered, but cerr is not. This means that any output sent to cerr is immediately output, but output to clog is written only when a buffer is full. Typically, cerr and clog are streams to which program debugging or error information is written. C++ also opens wide (16-bit) character versions of the standard streams called wcin, wcout, wcerr, and wclog. These streams exist to support languages, such as Chinese, that require large character sets. We won't be using them in this book. By default, the C++ standard streams are linked to the console, but they can be redirected to other devices or files by your program. They can also be redirected by the operating system.

CRITICAL SKILL 11.2: The C++ Stream Classes

As you learned in Module 1, C++ provides support for its I/O system in <iostream>.Inthis 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 you will learn in Module 12, a template 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 the template class can be created. As it relates to the I/O library, Standard C++ creates two specific versions of these template classes: one for 8-bit characters and another for wide characters. These specific versions act like any other classes, and no familiarity with templates is required to fully utilize the C++ I/O system.

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 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 specific versions of the I/O class hierarchies: one for 8-bit characters and one for wide characters. This book discusses only the 8-bit character classes since they are by far the most frequently used. Here is a list of the mapping of template class names to their character-based versions.

Template Class Name	Equivalent Character-Based Class Name
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

The character-based names will be used throughout the remainder of this book, since they are the names that you will 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.

1. What is a stream? What is a file?
2. What stream is connected to standard output?
3. C++ I/O is supported by a sophisticated set of class hierarchies. True or false?

CRITICAL SKILL 11.3: Overloading the I/O Operators

In the preceding modules, when a program needed to output or input the data associated with a class, member functions were created whose only purpose was to output or input the class' data. While there is nothing, in itself, wrong with this approach, C++ allows a much better way of performing I/O operations on classes: by overloading the << and the >> I/O operators.

In the language of C++, the << operator is referred to as the insertion operator because it inserts data into a stream. Likewise, the >> operator is called the extraction operator because it extracts data from a

stream. The operator functions that overload the insertion and extraction operators are generally called inserters and extractors, respectively.

In <iostream>, the insertion and extraction operators are overloaded for all of the C++ built-in types. Here you will see how to define these operators relative to classes that you create.

Creating Inserters

As a simple first example, let's create an inserter for the version of the ThreeD class shown here:

```
class ThreeD {
public:
  int x, y, z; // 3-D coordinates
  ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};
```


To create an inserter function for an object of type ThreeD, overload the << for it. Here is one way to do this:

```
// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream;  // return the stream
}
```

Let's look closely at this function, because many of its features are common to all inserter functions. First, notice that it is declared as returning a reference to an object of type ostream. This declaration is necessary so that several inserters of this type can be combined in a compound I/O expression. Next, the function has two parameters. The first is the reference to the stream that occurs on the left side of the << operator. The second parameter is the object that occurs on the right side. (This parameter can also be a reference to the object, if you like.) Inside the function, the three values contained in an object of type ThreeD are output, and stream is returned.

Here is a short program that demonstrates the inserter:

```
// Demonstrate a custom inserter.

#include <iostream>
using namespace std;

class ThreeD {
public:
  int x, y, z; // 3-D coordinates
  ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};

// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj) <-- An inserter for Three D
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream;  // return the stream
}
int main()
{
  ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);
  cout << a << b << c; <-- Use ThreeD inserter to output coordinates.

  return 0;
}
```

This program displays the following output:

```
1, 2, 3
3, 4, 5
5, 6, 7
```

If you eliminate the code that is specific to the ThreeD class, you are left with the skeleton for an inserter function, as shown here:

```
ostream &operator<<(ostream &stream, class_type obj)
{
  // class specific code goes here
  return stream;  // return the stream
}
```

Of course, it is permissible for obj to be passed by reference.

Within wide boundaries, what an inserter function actually does is up to you. However, good programming practice dictates that your inserter should produce reasonable output. Just make sure that you return stream.

Using Friend Functions to Overload Inserters

In the preceding program, the overloaded inserter function is not a member of ThreeD. In fact, neither inserter nor extractor functions can be members of a class. The reason is that when an operator function is a member of a class, the left operand (implicitly passed using the this pointer) is an object of that class. There is no way to change this. However, when inserters are overloaded, the left operand is a stream, and the right operand is an object of the class being output. Therefore, overloaded inserters must be nonmember functions.

The fact that inserters must not be members of the class they are defined to operate on raises a serious question: How can an overloaded inserter access the private elements of a class? In the preceding program, the variables x, y,and z were made public so that the inserter could access them. But hiding data is an important part of OOP, and forcing all data to be public is a serious inconsistency. However, there is a solution: an inserter can be a friend of a class. As a friend of the class for which it is defined, it has access to private data. Here, the ThreeD class and sample program are reworked, with the overloaded inserter declared as a friend:

// Use a friend to overload <<.

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

class ThreeD {
  int x, y, z; // 3-D coordinates - now private
public:
  ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
  friend ostream &operator<<(ostream &stream, ThreeD obj);
};

// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream;  // return the stream
}

int main()
{
  ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

  cout << a << b << c;

  return 0;
}
```

ThreeD inserter is now a friend and has access to private data.

Notice that the variables x, y, and z are now private to ThreeD, but can still be directly accessed by the inserter. Making inserters (and extractors) friends of the classes for which they are defined preserves the encapsulation principle of OOP.

Overloading Extractors

To overload an extractor, use the same general approach that you use when overloading an inserter. For example, the following extractor inputs 3-D coordinates into an object of type ThreeD. Notice that it also prompts the user.

```
// Get three-dimensional values - ThreeD extractor.
istream &operator>>(istream &stream, ThreeD &obj)
{
  cout << "Enter X,Y,Z values: ";
  stream >> obj.x >> obj.y >> obj.z;
  return stream;
}
```

An extractor must return a reference to an object of type istream. Also, the first parameter must be a reference to an object of type istream. This is the stream that occurs on the left side of the >>. The second parameter is a reference to the variable that will be receiving input. Because it is a reference, the second parameter can be modified when information is input.

The skeleton of an extractor is shown here:

```
istream &operator>>(istream &stream, object_type &obj)
{
  // put your extractor code here
  return stream;
}
```

The following program demonstrates the extractor for objects of type ThreeD:

```
// Demonstrate a custom extractor.

#include <iostream>
using namespace std;

class ThreeD {
  int x, y, z; // 3-D coordinates
public:
  ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
  friend ostream &operator<<(ostream &stream, ThreeD obj);
  friend istream &operator>>(istream &stream, ThreeD &obj);
} ;

// Display X, Y, Z coordinates - ThreeD inserter.
ostream &operator<<(ostream &stream, ThreeD obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream; // return the stream
}

// Get three dimensional values - ThreeD extractor.
istream &operator>>(istream &stream, ThreeD &obj) <-- Extractor for ThreeD
{
  cout << "Enter X,Y,Z values: ";
  stream >> obj.x >> obj.y >> obj.z;
  return stream;
}

int main()
{
  ThreeD a(1, 2, 3);

  cout << a;

  cin >> a;
  cout << a;

  return 0;
}
```

A sample run is shown here:

```
1, 2, 3
Enter X, Y, Z values: 5 6 7
5, 6, 7
```

Like inserters, extractor functions cannot be members of the class they are designed to operate upon. They can be friends or simply independent functions.

Except for the fact that you must return a reference to an object of type istream, you can do anything you like inside an extractor function. However, for the sake of structure and clarity, it is best to use extractors only for input operations.

1. What is an inserter?

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

2. What is an extractor?

3. Why are friend functions often used for inserter or extractor functions?

Formatted I/O

Up to this point, the format for inputting or outputting information has been left to the defaults provided by the C++ I/O system. However, you can precisely control the format of your data in either of two ways. The first uses member functions of the ios class. The second uses a

special type of function called a manipulator. We will begin by looking at formatting using the ios member functions.

CRITICAL SKILL 11.4: Formatting with the ios Member Functions

Each stream has associated with it a set of format flags that control the way information is formatted by a stream. 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 is a base class for ios.)

These values are used to set or clear the format flags. Some older compilers 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 whitespace characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When skipws is cleared, whitespace 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 is 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 and Clearing 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 showbase flag, you can use this statement:

stream.setf(ios::showbase);

Here, stream is the stream you want to affect. Notice the use of ios:: to qualify showbase. Because showbase is an enumerated constant defined by the ios class, it must be qualified by ios when it is referred to. This principle applies to all of the format flags.

The following program uses setf( ) to turn on both the showpos and scientific flags:

```
// Use setf().

#include <iostream>
using namespace std;

int main()
{
  // Turn on showpos and scientific flags.
  cout.setf(ios::showpos);
  cout.setf(ios::scientific);

  cout << 123 << " " << 123.23 << " ";

  return 0;
}
```

Set format flag using **setf( )**.

The output produced by this program is shown here:

+123 +1.232300e+002

You can OR together as many flags as you like in a single call. For example, by ORing together scientific and showpos, as shown next, you can change the program so that only one call is made to setf( ):

cout.setf(ios::scientific | ios::showpos);

To turn off a flag, use the unsetf( ) function, whose prototype is shown here: void unsetf(fmtflags flags); The flags specified by flags are cleared. (All other flags are unaffected.)

Sometimes it is useful to know the current flag settings. You can retrieve the current flag values using the flags( ) function, whose prototype is shown here: fmtflags flags( );

This function returns the current value of the flags relative to the invoking stream. The following form of flags( ) sets the flag values to those specified by flags and returns the previous flag values: fmtflags flags(fmtflags flags); The following program demonstrates flags( ) and unsetf( ):

```
// Demonstrate flags() and unsetf().

#include <iostream>
using namespace std;

int main()
{
  ios::fmtflags f;

  f = cout.flags(); <-- Get the format flags.

  if(f & ios::showpos)
    cout << "showpos is set for cout.\n";
  else
    cout << "showpos is cleared for cout.\n";

  cout << "\nSetting showpos for cout.\n";
  cout.setf(ios::showpos); <-- Set the showpos flag.

  f = cout.flags();
  if(f & ios::showpos)
    cout << "showpos is set for cout.\n";
  else
    cout << "showpos is cleared for cout.\n";

  cout << "\nClearing showpos for cout.\n";
  cout.unsetf(ios::showpos); <-- Clear the showpos flag.

  f = cout.flags();

  if(f & ios::showpos)
    cout << "showpos is set for cout.\n";
  else
    cout << "showpos is cleared for cout.\n";

  return 0;
}
```

The program produces this output:

```
showpos is cleared for cout.
Setting showpos for cout.
showpos is set for cout.
Clearing showpos for cout.
showpos is cleared for cout.
```

In the program, notice that the type fmtflags is preceded by ios:: when f is declared. This is necessary since fmtflags is a type defined by ios. In general, whenever you use the name of a type or enumerated constant that is defined by a class, you must qualify it with the name of the class.

Setting the Field Width, Precision, and Fill Character

In addition to the formatting flags, there are three member functions defined by ios that set these additional format values: the field width, the precision, and the fill character. The functions that set these values 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, then the field will be overrun. No values are truncated.

When outputting floating-point values in scientific notation, you can determine the number of digits to be displayed after the decimal point 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 you don't set it, the default precision is 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 demonstrates these three functions:

```
// Demonstrate width(), precision(), and fill().

#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpos);
  cout.setf(ios::scientific);
  cout << 123 << " " << 123.23 << "\n";

  cout.precision(2); // two digits after decimal point
  cout.width(10);    // in a field of 10 characters
  cout << 123 << " ";
  cout.width(10);    // set width to 10
  cout << 123.23 << "\n";

  cout.fill('#');  // fill using #
  cout.width(10);  // in a field of 10 characters
  cout << 123 << " ";
  cout.width(10);  // set width to 10
  cout << 123.23;

  return 0;
}
```

Set the precision.

Set field width.

Set the fill character.

The program displays this output:

```
+123 +1.232300e+002
      +123 +1.23e+002
######+123 +1.23e+002
```

As mentioned, in some implementations, it is necessary to reset the field width before each output operation. This is why width( ) is called repeatedly in the preceding program. 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( );

1. What does boolalpha do?
2. What does setf( ) do?
3. What function is used to set the fill character?

CRITICAL SKILL 10.5: Using I/O Manipulators

The C++ I/O system includes a second way in which you can alter the format parameters of a stream. This method uses special functions, called manipulators, that can be included in an I/O expression. The standard manipulators are shown in Table 11-1. To use those manipulators that take arguments, you must include <iomanip> in your program.

Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolalpha** flag	Input/Output
dec	Turns on **dec** flag	Input/Output
endl	Outputs a newline character and flushes the stream	Output
ends	Outputs a null	Output
fixed	Turns on **fixed** flag	Output
flush	Flushes a stream	Output
hex	Turns on **hex** flag	Input/Output
internal	Turns on **internal** flag	Output
left	Turns on **left** flag	Output
noboolalpha	Turns off **boolalpha** flag	Input/Output
noshowbase	Turns off **showbase** flag	Output
noshowpoint	Turns off **showpoint** flag	Output
noshowpos	Turns off **showpos** flag	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)	Turns off the flags specified in f	Input/Output
right	Turns on **right** flag	Output
scientific	Turns on **scientific** flag	Output
setbase(int base)	Sets the number base to base	Input/Output
setfill(int ch)	Sets the fill character to ch	Output
setiosflags(fmtflags f)	Turns on the flags specified in f	Input/Output
setprecision (int p)	Sets the number of digits of precision	Output
setw(int w)	Sets the field width to w	Output
showbase	Turns on **showbase** flag	Output
showpoint	Turns on **showpoint** flag	Output

Table 11-1 The C++ I/O Manipulators

A

Manipulator	Purpose	Input/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	Skips leading whitespace	Input

Table 11-1 The C++ I/O Manipulators (continued)

manipulator is used as part of a larger I/O expression. Here is a sample program that uses manipulators to control the format of its output:

```
// Demonstrate an I/O manipulator.

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

int main()
{
  cout << setprecision(2) << 1000.243 << endl;
  cout << setw(20) << "Hello there.";

  return 0;
}
```

Use I/O manipulators.

It produces this output:

```
1e+003
        Hello there.
```

Notice how the manipulators occur in the chain of I/O operations. Also, notice that when a manipulator does not take an argument, such as endl in the example, it is not followed by parentheses.

The following program uses setiosflags( ) to set the scientific and showpos flags:

```
// Use setiosflags().

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

```
int main()
{
  cout << setiosflags(ios::showpos) <<          <-- Use setiosflags( ).
          setiosflags(ios::scientific) <<
          123 << " " << 123.23;

  return 0;
}
```

The program shown next uses ws to skip any leading whitespace when inputting a string into s:

```
// Skip leading whitespace.

#include <iostream>
using namespace std;

int main()
{
  char s[80];

  cin >> ws >> s; <-- Use ws.
  cout << s;

  return 0;
}
```

CRITICAL SKILL 11.6: Creating Your Own Manipulator Functions

You can create your own manipulator functions. There are two types of manipulator functions: those that take arguments and those that don't. The creation of parameterized manipulators requires the use of techniques beyond the scope of this book. However, the creation of parameterless manipulators is quite easy and 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. It is important to understand that even though the manipulator has as its single argument a pointer to the stream upon which

it is operating, no argument is specified when the manipulator is used in an output expression.

The following program creates a manipulator called setup( ) that turns on left justification, sets the field width to 10, and specifies that the dollar sign will be the fill character.

```
// Create an output manipulator.

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

ostream &setup(ostream &stream) <-- A custom output manipulator
{
  stream.setf(ios::left);
  stream << setw(10) << setfill('$');
  return stream;
}

int main()
{
  cout << 10 << " " << setup << 10;

  return 0;
}
```

Custom manipulators are useful for two reasons. First, you might need to perform an I/O operation on a device for which none of the predefined manipulators applies—a plotter, for example. In this case, creating your own manipulators will make it more convenient when outputting to the device. Second, you may find that you are repeating the same sequence of operations many times. You can consolidate these operations into a single manipulator, as the foregoing program illustrates.

All parameterless input manipulator functions have this skeleton:

```
istream &manip_name(istream &stream)
{
  // your code here

  return stream;
}
```

For example, the following program creates the prompt( ) manipulator. It displays a prompting message and then configures input to accept hexadecimal.

```
// Create an input manipulator.

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

istream &prompt(istream &stream)  <-- A custom input manipulator
{
  cin >> hex;
  cout << "Enter number using hex format: ";

  return stream;
}

int main()
{
  int i;

  cin >> prompt >> i;
  cout << i;

  return 0;
}
```

Remember that it is crucial that your manipulator return stream. If this is not done, then your manipulator cannot be used in a chain of input or output operations.

1. What does endl do?
2. What does ws do?
3. Is an I/O manipulator used as part of a larger I/O expression?

File I/O

You can use the C++ I/O system to perform file I/O. To perform file I/O, you must include the header <fstream> in your program. It defines several important classes and values.

CRITICAL SKILL 11.7: Opening and Closing a File

In C++, a file is opened by linking it to a stream. As you know, there are three types of streams: input, output, and input/output. To open an input stream, you must declare the stream to be of class ifstream.

To open an output stream, it must be declared as class ofstream. A stream that will be performing both input and output operations must be declared as class fstream. For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output:

```
ifstream in;  // input
ofstream out; // output
fstream both; // input and output
```

Once you have created a stream, one way to associate it with a file is by using open( ).This function is a member of each of the three stream classes. The prototype for each is shown here:

```
void ifstream::open(const char *filename,
                    ios::openmode mode = ios::in);
void ofstream::open(const char *filename,
                    ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename,
                    ios::openmode mode = ios::in | ios::out);
```

Here, filename is the name of the file; it can include a path specifier. The value of mode determines how the file is opened. It must be one or more of the values defined by openmode, which is an enumeration defined by ios (through its base class ios_base). The values are shown here:

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

You can combine two or more of these values by ORing them together. Including ios::app causes all output to that file to be appended to the end. This value can be used only with files capable of output. Including ios::ate causes a seek to the end of the file to occur when the file is opened. Although ios::ate causes an initial seek to end-of-file, I/O operations can still occur anywhere within the file.

The ios::in value specifies that the file is capable of input. The ios::out value specifies that the file is capable of output.

The ios::binary value causes a file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations may take place, such as carriage return–linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur. Understand that any file, whether it contains formatted text or raw data, can be opened in either binary or text mode. The only difference is whether character translations take place.

The ios::trunc value causes the contents of a preexisting file by the same name to be destroyed, and the file to be truncated to zero length. When creating an output stream using ofstream, any preexisting file by that name is automatically truncated.

The following fragment opens a text file for output:

```
ofstream mystream; mystream.open("test");
```

Since the mode parameter to open( ) defaults to a value appropriate to the type of stream being opened, there is often no need to specify its value in the preceding example. (Some compilers do not default the mode parameter for fstream::open( ) to in | out, so you might need to specify this explicitly.)

If open( ) fails, the stream will evaluate to false when used in a Boolean expression. You can make use of this fact to confirm that the open operation succeeded by using a statement like this:

```
if(!mystream) {

cout << "Cannot open file.\n";

// handle error }
```

In general, you should always check the result of a call to open( ) before attempting to access the file. You can also check to see if you have successfully opened a file by using the is_open( ) function, which is a member of fstream, ifstream, and ofstream. It has this prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if mystream is currently open:

```
if(!mystream.is_open()) {

cout << "File is not open.\n";

// ...
```

Although it is entirely proper to use the open( ) function for opening a file, most of the time you will not do so because the ifstream, ofstream, and fstream classes have constructors that automatically open the file. The constructors have the same parameters and defaults as the open( ) function. Therefore, the most common way you will see a file opened is shown in this example:

```
ifstream mystream("myfile"); // open file for input
```

If, for some reason, the file cannot be opened, the value of the associated stream variable will evaluate to false. To close a file, use the member function close( ). For example, to close the file linked to a stream called mystream, you would use this statement:

```
mystream.close();
```

The close( ) function takes no parameters and returns no value.

CRITICAL SKILL 11.8: Reading and Writing Text Files

The easiest way to read from or write to a text file is to use the << and >> operators. For example, this program writes an integer, a floating-point value, and a string to a file called test:

```
// Write to file.

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

int main()
{
  ofstream out("test"); <-- Create and open a file called "test" for text output.
  if(!out) {
    cout << "Cannot open file.\n";
    return 1;
   }

  out << 10 << " " << 123.23 << "\n"; <-- Output to the file.
  out << "This is a short text file.";

  out.close(); <-- Close the file.
  return 0;
}
```

The following program reads an integer, a float, a character, and a string from the file created by the previous program:

```
// Read from file.

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

int main()
{
  char ch;
  int i;
  float f;
  char str[80];

  ifstream in("test"); <-- Open a file for text input.
  if(!in) {
    cout << "Cannot open file.\n";
    return 1;
  }

  in >> i;
  in >> f;
  in >> ch;
  in >> str;          <-- Read from the file.

  cout << i << " " << f << " " << ch << "\n";
  cout << str;
  in.close(); <-- Close the file.
  return 0;
}
```

Keep in mind that when the >> operator is used for reading text files, certain character translations occur. For example, whitespace characters are omitted. If you want to prevent any character translations, you must open a file for binary access. Also remember that when >> is used to read a string, input stops when the first whitespace character is encountered.

Ask the Expert

Q: As you explained in Module 1, C++ is a superset of C. I know that C defines an I/O system of its own. Is the C I/O system available to C++ programmers? If so, should it be used in C++ programs?

A: The answer to the first question is yes. The C I/O system is available to C++ programmers. The answer to the second question is a qualified no. The C I/O system is not object-oriented. Thus, you will nearly always find the C++ I/O system more compatible with C++ programs. However, the C I/O system is still widely used and is quite streamlined, carrying little overhead. Thus, for some highly specialized programs, the C I/O system might be a good choice. Information on the C I/O system can be found in my book C++: The Complete Reference (Osborne/McGraw-Hill).

1. What class creates an input file?
2. What function opens a file?
3. Can you read and write to a file using << and >>?

CRITICAL SKILL 11.9: Unformatted and Binary I/O

While reading and writing formatted text files is very easy, it is not always the most efficient way to handle files. Also, there will be times when you need to store unformatted (raw) binary data, not text. The functions that allow you to do this are described here.

When performing binary operations on a file, be sure to open it using the ios::binary mode specifier. Although the unformatted file functions will work on files opened for text mode, some character translations may occur. Character translations negate the purpose of binary file operations.

In general, there are two ways to write and read unformatted binary data to or from a file. First, you can write a byte using the member function put( ), and read a byte using the member function get( ). The second way uses the block I/O functions: read( ) and write( ). Each is examined here.

Using get( ) and put( )

The get( ) function has many forms, but the most commonly used version is shown next, along with that of put( ):

istream &get(char &ch); ostream &put(char ch);

The get( ) function reads a single character from the associated stream and puts that value in ch. It returns a reference to the stream. This value will be null if the end of the file is reached. The put( ) function writes ch to the stream and returns a reference to the stream.

The following program will display the contents of any file on the screen. It uses the get( ) function:

```
// Display a file using get().

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

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Usage: PR <filename>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Cannot open file.\n";
    return 1;
 }

  while(in) { // in will be false when eof is reached
    in.get(ch);
    if(in) cout << ch;
  }
  in.close();

  return 0;
}
```

Open the file for binary operations.

Read data until the end of the file is reached.

Look closely at the while loop. When in reaches the end of the file, it will be false, causing the while loop to stop.

There is actually a more compact way to code the loop that reads and displays a file, as shown here:

```
while(in.get(ch)) cout << ch;
```

This form works because get( ) returns the stream in, and in will be false when the end of the file is encountered. This program uses put( ) to write a string to a file.

```
// Use put() to write to a file.

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

int main()
{
  char *p = "hello there\n";

  ofstream out("test", ios::out | ios::binary);
  if(!out) {
    cout << "Cannot open file.\n";
    return 1;
   }

  // Write characters until the null-terminator is reached.
  while(*p) out.put(*p++);

  out.close();

  return 0;
}
```

Write a string to a file using **put( )**.
No character translations will occur.

After this program executes, the file test will contain the string “hello there” followed by a newline character. No character translations will have taken place.

Reading and Writing Blocks of Data

To read and write blocks of binary data, use the read( ) and write( ) member functions. Their prototypes are shown here:

istream &read(char *buf, streamsize num); ostream &write(const char *buf, streamsize num);

The read( ) function reads num bytes from the associated stream and puts them in the buffer pointed to by buf. The write( ) function writes num bytes to the associated stream from the buffer pointed to by buf. As mentioned earlier, streamsize is some form of integer defined by the C++ library. It is capable of holding the largest number of bytes that can be transferred in any one I/O operation.

The following program writes and then reads an array of integers:

```
// Use read() and write().

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

int main()
{
  int n[5] = {1, 2, 3, 4, 5};
  register int i;

  ofstream out("test", ios::out | ios::binary);
  if(!out) {
    cout << "Cannot open file.\n";
    return 1;
   }

  out.write((char *) &n, sizeof n); <-------- Write a block of data.

  out.close();

  for(i=0; i<5; i++) // clear array
    n[i] = 0;

  ifstream in("test", ios::in | ios::binary);
  if(!in) {
    cout << "Cannot open file.\n";
    return 1;
  }
  in.read((char *) &n, sizeof n);  <-------- Read a block of data.

  for(i=0; i<5; i++) // show values read from file
    cout << n[i] << " ";

  in.close();

  return 0;
}
```

Note that the type casts inside the calls to read( ) and write( ) are necessary when operating on a buffer that is not defined as a character array.

If the end of the file is reached before num characters have been read, then read( ) simply stops, and the buffer will contain as many characters as were available. You can find out how many characters have been read using another member function, called gcount( ), which has this prototype:

streamsize gcount( );

gcount( ) returns the number of characters read by the last input operation.

1. To read or write binary data, you open a file using what mode specifier?
2. What does get( ) do? What does put( ) do?
3. What function reads a block of data?

CRITICAL SKILL 11.10: More I/O Functions

The C++ I/O system defines other I/O related functions, several of which you will find useful. They are discussed here.

More Versions of get( )

In addition to the form shown earlier, the get( ) function is overloaded in several different ways. The prototypes for the three most commonly used overloaded forms are shown here:

istream &get(char *buf, streamsize num); istream &get(char *buf, streamsize num, char delim); int get( );

The first form reads characters into the array pointed to by buf until either num–1 characters have been read, a newline is found, or the end of the file has been encountered. The array pointed to by buf will be null-terminated by get( ). If the newline character is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.

The second form reads characters into the array pointed to by buf until either num–1 characters have been read, the character specified by delim has been found, or the end of the file has been encountered. The array pointed to by buf will be null-terminated by get( ). If the delimiter character is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.

The third overloaded form of get( ) returns the next character from the stream. It returns EOF (a value that indicates end-of-file) if the end of the file is encountered. EOF is defined by <iostream>.

One good use for get( ) is to read a string that contains spaces. As you know, when you use >> to read a string, it stops reading when the first whitespace character is encountered. This makes >> useless for reading a string containing spaces. However, you can overcome this problem by using get(buf, num), as illustrated in this program:

```
// Use get() to read a string that contains spaces.

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

int main()
{
  char str[80];

  cout << "Enter your name: ";
  cin.get(str, 79); <-- Use get( ) to read a string that contains whitespace.

  cout << str << '\n';

  return 0;
}
```

Here, the delimiter to get( ) is allowed to default to a newline. This makes get( ) act much like the standard gets( ) function.

getline( )

Another function that performs input is getline( ). It is a member of each input stream class. Its prototypes are shown here:

istream &getline(char *buf, streamsize num); istream &getline(char *buf, streamsize num, char delim);

The first form reads characters into the array pointed to by buf until either num–1 characters have been read, a newline character has been found, or the end of the file has been encountered.

The array pointed to by buf will be null-terminated by getline( ). If the newline character is encountered in the input stream, it is extracted, but is not put into buf.

The second form reads characters into the array pointed to by buf until either num–1 characters have been read, the character specified by delim has been found, or the end of the file has been encountered. The array pointed to by buf will be null-terminated by getline( ). If the delimiter character is encountered in the input stream, it is extracted, but is not put into buf.

As you can see, the two versions of getline( ) are virtually identical to the get(buf, num) and get(buf, num, delim) versions of get( ). Both read characters from input and put them into the array pointed to by buf until either num–1 characters have been read or until the delimiter character is encountered. The difference between get( ) and getline( ) is that getline( ) reads and removes the delimiter from the input stream; get( ) does not.

Detecting EOF

You can detect when the end of the file is reached by using the member function eof( ), which has this prototype:

bool eof( );

It returns true when the end of the file has been reached; otherwise it returns false.

peek( ) and putback( )

You can obtain the next character in the input stream without removing it from that stream by using peek( ). It has this prototype:

int peek( );

peek( ) returns the next character in the stream, or EOF if the end of the file is encountered. The character is contained in the low-order byte of the return value. You can return the last character read from a stream to that stream by using putback( ). Its prototype is shown here:

istream &putback(char c);

where c is the last character read.

flush( )

When output is performed, data is not immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to disk before the buffer is full by calling flush( ). Its prototype is shown here:

ostream &flush( );

Calls to flush( ) might be warranted when a program is going to be used in adverse environments (in situations where power outages occur frequently, for example).

NOTE: Closing a file or terminating a program also flushes all buffers.

Project 11-1 A File Comparison Utility

This project develops a simple, yet useful file comparison utility. It works

by opening both files to be compared and then reading and comparing each corresponding set of bytes. If a mismatch is found, the files differ. If the end of each file is reached at the same time and if no mismatches have been found, then the files are the same.

Step by Step

1. Create a file called CompFiles.cpp.

2. Begin by adding these lines to CompFiles.cpp:

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

int main(int argc, char *argv[])
{
  register int i;

  unsigned char buf1[1024], buf2[1024];

  if(argc!=3) {
    cout << "Usage: compfiles <file1> <file2>\n";
    return 1;
  }
```

Notice that the names of the files to compare are specified on the command line.

3. Add the code that opens the files for binary input operations, as shown here:

```
ifstream f1(argv[1], ios::in | ios::binary);
if(!f1) {
  cout << "Cannot open first file.\n";
  return 1;
}

ifstream f2(argv[2], ios::in | ios::binary);
if(!f2) {
  cout << "Cannot open second file.\n";
  return 1;
 }
```

The files are opened for binary operations to prevent the character translations that might occur in text mode.

4. Add the code that actually compares the files, as shown next:

```
cout << "Comparing files...\n";

do {
  f1.read((char *) buf1, sizeof buf1);
  f2.read((char *) buf2, sizeof buf2);

  if(f1.gcount() != f2.gcount()) {
    cout << "Files are of differing sizes.\n";
    f1.close();
```

```
      f2.close();
      return 0;
    }

    // compare contents of buffers
    for(i=0; i<f1.gcount(); i++)
      if(buf1[i] != buf2[i]) {
        cout << "Files differ.\n";
        f1.close();
        f2.close();
        return 0;
      }

  } while(!f1.eof() && !f2.eof());

  cout << "Files are the same.\n";
```

This code reads one buffer at a time from each of the files using the read( ) function. It then compares the contents of the buffers. If the contents differ, the files are closed, the "Files differ." message is displayed, and the program terminates. Otherwise, buffers continue to be read and compared until the end of one (or both) files is reached. Because less than a full buffer may be read at the end of a file, the program uses the gcount( ) function to determine precisely how many characters are in the buffers. If one of the files is shorter than the other, the values returned by gcount( ) will differ when the end of one of the files is reached. In this case, the message "Files are of differing sizes." will be displayed. Finally, if the files are the same, then when the end of one file is reached, the other will also have been reached. This is confirmed by calling eof( ) on each stream. If the files compare equal in all regards, then they are reported as equal.

5. Finish the program by closing the files, as shown here:

```
f1.close();
f2.close();
return 0; }
```

6. The entire FileComp.cpp program is shown here:

```
/*
   Project 11-1

   Create a file comparison utility.
*/

#include <iostream>

#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  register int i;

  unsigned char buf1[1024], buf2[1024];
```

```
  if(argc!=3) {
    cout << "Usage: compfiles <file1> <file2>\n";
    return 1;
  }

  ifstream f1(argv[1], ios::in | ios::binary);
  if(!f1) {
    cout << "Cannot open first file.\n";
    return 1;
  }
  ifstream f2(argv[2], ios::in | ios::binary);
  if(!f2) {
    cout << "Cannot open second file.\n";
    return 1;
   }

  cout << "Comparing files...\n";

  do {
    f1.read((char *) buf1, sizeof buf1);
    f2.read((char *) buf2, sizeof buf2);

    if(f1.gcount() != f2.gcount()) {
      cout << "Files are of differing sizes.\n";
      f1.close();
      f2.close();
      return 0;
    }

    // compare contents of buffers
    for(i=0; i<f1.gcount(); i++)
      if(buf1[i] != buf2[i]) {
        cout << "Files differ.\n";
        f1.close();
        f2.close();
        return 0;
      }

  } while(!f1.eof() && !f2.eof());

  cout << "Files are the same.\n";

  f1.close();
  f2.close();

  return 0;
}
```

7. To try CompFiles, first copy CompFiles.cpp to a file called temp.txt. Then, try this command line:
   CompFiles CompFiles.temp txt
   The program will report that the files are the same. Next, compare CompFiles.cpp to a different file, such as one of the other program files from this module. You will see that CompFiles reports that the files differ.
8. On your own, try enhancing CompFiles with various options. For example, add an option that ignores the case of letters. Another idea is to have CompFiles display the position within the file where the files differ.

CRITICAL SKILL 11.11: Random Access

So far, files have been read or written sequentially, but you can also access a file in random order. In C++'s I/O system, you perform random access using the seekg( ) and seekp( ) functions. Their most common forms are shown here:

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

Here,

Value	Meaning
ios::beg	Beginning of file
ios::cur	Current location
ios::end	End of file

off_type is an integer type defined by ios that is capable of containing the largest valid value that offset can have. seekdir is an enumeration that has these values:

The C++ I/O system manages two pointers associated with a file. One is the get pointer, which specifies where in the file the next input operation will occur. The other is the put pointer, which specifies where in the file the next output operation will occur. Each time an input or an output operation takes place, the appropriate pointer is automatically advanced. Using the seekg( ) and seekp( ) functions, it is possible to move this pointer and access the file in a non-sequential fashion.
The seekg( ) function moves the associated file's current get pointer offset number of bytes from the specified origin. The seekp( ) function moves the associated file's current put pointer offset number of bytes from the specified origin.
Generally, random access I/O should be performed only on those files opened for binary operations. The character translations that may occur on text files could cause a position request to be out of sync with the actual contents of the file.

The following program demonstrates the seekp( ) function. It allows you to specify a filename on the command line, followed by the specific byte that you want to change in the file. The program then writes an X at the specified location. Notice that the file must be opened for read/write operations.

```
// Demonstrate random access.

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Usage: CHANGE <filename> <byte>\n";
    return 1;
  }

  fstream out(argv[1], ios::in | ios::out | ios::binary);
  if(!out) {
    cout << "Cannot open file.\n";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);

  out.put('X');
  out.close();

  return 0;
}
```

Seek to a specific byte within the file.
This moves the put pointer.

The next program uses seekg( ). It displays the contents of a file, beginning with the location you specify on the command line.

```
// Display a file from a given starting point.

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=3) {
    cout << "Usage: NAME <filename> <starting location>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Cannot open file.\n";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg); <-- This moves the get pointer.

  while(in.get(ch))
    cout << ch;

  return 0;
}
```

You can determine the current position of each file pointer using these functions:
pos_type tellg( ); pos_type tellp( );

Here, pos_type is a type defined by ios that is capable of holding the largest value that either function can return. There are overloaded versions of seekg( ) and seekp( ) that move the file pointers to the location specified by the return values of tellg( ) and tellp( ). Their prototypes are shown here:

istream &seekg(pos_type position); ostream &seekp(pos_type position);

1. What function detects the end of the file?
2. What does getline( ) do?
3. What functions handle random access position requests?

CRITICAL SKILL 11.12: Checking I/O Status

The C++ I/O system maintains status information about the outcome of each I/O operation. The current status of an I/O stream is described in an object of type iostate, which is an enumeration defined by ios that includes these members.

Name	Meaning
ios::goodbit	No error bits set
ios::eofbit	1 when end-of-file is encountered; 0 otherwise
ios::failbit	1 when a (possibly) nonfatal I/O error has occurred; 0 otherwise
ios::badbit	1 when a fatal I/O error has occurred; 0 otherwise

There are two ways in which you can obtain I/O status information. First, you can call the rdstate( ) function. It has this prototype:

iostate rdstate( );

It returns the current status of the error flags. As you can probably guess from looking at the preceding list of flags, rdstate( ) returns goodbit when no error has occurred. Otherwise, an error flag is turned on.

The other way you can determine if an error has occurred is by using one or more of these ios member functions:

bool bad( ); bool eof( );

bool fail( ); bool good( );

The eof( ) function was discussed earlier. The bad( ) function returns true if badbit is set. The fail( ) function returns true if failbit is set. The good( ) function returns true if there are no errors. Otherwise they return false.

Once an error has occurred, it may need to be cleared before your program continues. To do this, use the ios member function clear( ), whose prototype is shown here:

void clear(iostate flags = ios::goodbit);

If flags is goodbit (as it is by default), all error flags are cleared. Otherwise, set flags to the settings you desire.

Before moving on, you might want to experiment with using these status-reporting functions to add extended error-checking to the preceding file examples.

Module 11 Mastery Check

1. What are the four predefined streams called?

2. Does C++ define both 8-bit and wide-character streams?

3. Show the general form for overloading an inserter.

4. What does ios::scientific do?

5. What does width( ) do?

6. An I/O manipulator is used within an I/O expression. True or false?

7. Show how to open a file for reading text input.

8. Show how to open a file for writing text output.

9. What does ios::binary do?

10. When the end of the file is reached, the stream variable will evaluate as false. True or false?

11. Assuming a file is associated with an input stream called strm, show how to read to the end of the file.

12. Write a program that copies a file. Allow the user to specify the name of the input and output file on the command line. Make sure that your program can copy both text and binary files.

13. Write a program that merges two text files. Have the user specify the names of the two files on the command line in the order they should appear in the output file. Also, have the user specify the name of the output file. Thus, if the program is called merge, then the following command line will merge the files MyFile1.txt and MyFile2.txt into Target.txt:

```
merge MyFile1.txt MyFile2.txt Target.txt
```

14. Show how the seekg( ) statement will seek to the 300th byte in a stream called MyStrm.