Unit 4

Index

Chapter 1
The C++ I/O System Basics

Unit-4
Chapter 1
The C++ I/O System Basics

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive or a network connection to main memory, this is called *input* operation and if bytes flow from main memory to a device like a display screen, a printer, a disk drive or a network connection, this is called *output* operation.

1.1 C++ Streams:

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.

1.2 The C++ Stream Classes:

Standard C++ provides support for its I/O system in <iostream>. In this header complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of template classes. 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.

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 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. 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

Note:
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.

1.3 C++ Pre-Defined 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

cin: The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs.
cout: The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs.
cerr: The predefined object cerr is an instance of ostream class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object cerr is un-buffered and each stream insertion to cerr causes its output to appear immediately.
clog: The predefined object clog is an instance of ostream class. The clog object is said to be attached to the standard error device, which is also a display screen but the object clog is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

1.4 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. 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.

Formatting using ios Members:

Each stream is associated with 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.

adjustfield	Basefield	boolalpha	Dec
Fixed	Floatfield	hex	Internal
Left	Oct	right	Scientific
showbase	Showpoint	showpos	Skipws
Unitbuf	Uppercase		

1.5 Setting the Format Flags:

To set a flag *setf( )* function is used. This function is a member of ios. Its syntax is shown below:
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);
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;
}
```

cout.setf(ios::showpoint | ios::showpos); can be used instead of multiple setf( ) function.
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.

1.6 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;
}
```

1.7 An Overloaded form of setf( ):

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

fmtflags setf(fmtflags flags1, fmtflags flags2);

Here 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. 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.

1.8 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 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;
}
```

1.9 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 the Table. Here 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.

Manipulators	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
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
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
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

1.10 Overloading << and >>:

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. So 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.

```
#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;
}
```

Creating your own Extractor:

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. 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.

```
#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;
}
```

1. setw():

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  int a=200,b=300;
  cout<<setw(5)<<a<<setw(5)<<b<<endl;
  cout<<setw(6)<<a<<setw(6)<<b<<endl;
  cout<<setw(7)<<a<<setw(7)<<b<<endl;
  return 0;
}
```

Output:

```
 200  300
  200   300
   200    300
```

2.setfill():

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  int a=200,b=300;
  cout<<setfill('*');
  cout<<setw(5)<<a<<setw(5)<<b<<endl;
  cout<<setw(6)<<a<<setw(6)<<b<<endl;
  cout<<setw(7)<<a<<setw(7)<<b<<endl;
  cout<<setw(8)<<a<<setw(8)<<b<<endl;
  return 0;
}
```

Output:

```
**200**300
***200***300
****200****300
*****200*****300
```

3.setbase():

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  int value;
  cout<<"enter number:";
  cin>>value;
  cout<<"Decimal base:
"<<setbase(10)<<value<<endl;
  cout<<"Hexadecimal base:
"<<setbase(16)<<value<<endl;
  cout<<"Octal base:
"<<setbase(8)<<value<<endl;
  return 0;
}
```

Output:

```
Enter number:15
Decimal base: 15
Hexadecimal base: f
Octal base: 17
```

4.dec,hex,oct:

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  int value;
  cout<<"Enter number:";
  cin>>value;
  cout<<"Decimal base: "<<dec<<value<<endl;
  cout<<"Hexadecimal base:
"<<hex<<value<<endl;
  cout<<"Octal base: "<<oct<<value<<endl;
  return 0;
}
```

Output:

```
Enter number:15
Decimal base: 15
Hexadecimal base: f
Octal base: 17
```

5.setprecision():

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  float a=5, b=3, c;
  c=a/b;
  cout<<setprecision(1)<<c<<endl;
  cout<<setprecision(2)<<c<<endl;
  cout<<setprecision(3)<<c<<endl;
  cout<<setprecision(4)<<c<<endl;
  cout<<setprecision(5)<<c<<endl;
  return 0;
}
```

Output:

```
2
1.7
1.67
1.667
1.6667
```

6.ends:

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  int no=231;
  cout<<'\"'<<"number="<<no<<ends;
  cout<<'\"'<<endl;
  return 0;
}
```

Output:

```
"number=231"
```

7.endl:

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

int main()
{
  int no=231;
  cout<<'\"'<<"number="<<no<<endl;
  cout<<'\"'<<endl;
  return 0;
}
```

Output:

```
"number=231
"
```

8.ws:

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  char name[10];
  cout<<"Enter a line of text:";
  cin>>ws;
  cin>>name;
  cout<<"Typed text is: "<<name<<endl;
  return 0;
}
```

Output:

```
Enter a line of text:Welcome to REVA
Typed text is: Welcome
```

9.flush():

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout<<"My name is Jack"<<endl;
  cout<<"Greetings to you"<<endl;
  cout.flush();
}
```

Output:

```
My name is Jack
Greetings to you
```

More programs on Manipulators:

1.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout<<setw(10)<<"Hello"<<endl;
  cout<<setw(15)<<setprecision(2)<<2.5555;
  cout<<setiosflags(ios::hex);
  cout<<endl<<"Hexadecimal of 84 is:"
                    <<84<<endl;
  return 0;
}
```

Output:

```
Hello
        2.6
Hexadecimal of 84 is: 84
```

2.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  int x=84;
  cout<<"Hexadecimal number:
"<<hex<<x<<endl;
  cout<<"Octal number: "<<oct<<x<<endl;
  return 0;
}
```

Output:

```
Hexadecimal number: 54
Octal number: 124
```

3.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout<<"Demo of endl";
  endl(cout);
  cout<<"It splits a line";
  endl(cout);
  return 0;
}
```

Output:

```
Demo of endl
It splits a line
```

4.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  char text[20];
  cout<<"enter text: ";
  cin.getline(text,20);
  cout<<"Text entered: "<<text;
  flush(cout);
  cout<<endl;
  return 0;
}
```

Output:

```
Enter text: Hi how are you?
Text entered: Hi how are you?
```

5.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout<<hex<<100<<endl;
  cout<<setfill('?')<<setw(10)<<2343.0<<endl;
  return 0;
}
```

Output:

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

6.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout.setf(ios::hex,ios::basefield);
  cout<<100<<endl;
  cout.width(10);
  cout.fill('?');
  cout<<2343.0<<endl;
  return 0;
}
```

Output:

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

7.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout<<setiosflags(ios::showpos);
  cout<<setiosflags(ios::showbase);
  cout<<123<<" "<<hex<<123<<endl;
  return 0;
}
```

Output:

```
+123 0x7b
```

8.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  bool b;
  b=true;
  cout<<b<<" "<<boolalpha<<b<<endl;
  cout<<"Enter boolean value: ";
  cin>>b;
  cout<<"Here is what you entered:
"<<b<<endl;
  return 0;
}
```

Output:

```
1 true
Enter boolean value: 1
Here is what you entered: true
Or
1 true
Enter boolean value: 0
Here is what you entered: false
```

9.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
  cout.precision(4);
  cout.width(10);
  cout<<10.12345<<endl;

  cout.fill('*');
  cout.width(10);
  cout<<10.12345<<endl;

  cout.width(10);
  cout.setf(ios::left);
  cout<<10.12345<<endl;
  return 0;
}
```

Output:

```
    10.12
*****10.12
10.12*****
```

Index

Chapter 2
The C++ File I/O

Chapter 2
C++ File I/O

2.1 <fstream> and the File Classes:

To perform file I/O the header <fstream> should be included in your program. It defines several classes ifstream, ofstream and fstream. These classes are derived from istream, ostream and iostream respectively. The istream, ostream and iostream are derived from ios, so ifstream, ofstream and fstream also have access to all operations defined by ios.
Another class used by the file system is filebuf, which provides low-level facilities to manage a file stream. Usually filebuf is not used directly but it is part of the other file classes.

2.2 Opening and Closing a File:

In C++ you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream. There are three types of streams: input, output, and input/output.
To create an input stream, you must declare the stream to be of class ifstream. To create an output stream, you must declare it as class ofstream. Streams 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 io; // input and output
```

Once you have created a stream the file can be opened 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 following values defined by openmode, which is an enumeration defined by ios (through its base class ios_base).

```
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.

A 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 is 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 normal output file.

```
ofstream out;
out.open("test", ios::out);
```

However you will see open( ) called as shown because the mode parameter provides default values for each type of stream. As their prototypes show for ifstream, mode defaults to ios::in; for ofstream, it is ios::out | ios::trunc; and for fstream, it is ios::in | ios::out. Therefore, the preceding statement will usually look like this:

```
out.open("test"); // defaults to output and normal file
```

If open( ) fails, the stream will evaluate to false when used in a Boolean expression.

Before using a file, you should test to make sure that the open operation succeeded. You can do so by using a statement like this:

```
if(!mystream) {
cout << "Cannot open file.\n";
// handle error
}
```

Although it is entirely proper to open a file by using the open( ) function, 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, you will most commonly see a file opened as shown here:

```
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. Therefore, whether you use a constructor to open the file or an explicit call to open( ) you will want to confirm that the file has actually been opened by testing the value of the stream.

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";
// ...
```

To close a file, use the member function close( ). For example, to close the file linked to a stream called mystream, use this statement:

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

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

2.3 Reading and Writing Text Files:

It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using cin and cout, substitute a stream that is linked to a file. For example, this program creates a short inventory file that contains each item's name and its cost:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ofstream out("INVNTRY"); // output, normal file
if(!out) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
out << "Radios " << 39.95 << endl;
out << "Toasters " << 19.95 << endl;
out << "Mixers " << 24.80 << endl;
out.close();
return 0;
}
```

The following program reads the inventory file created by the previous program and displays its contents on the screen:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream in("INVNTRY"); // input
if(!in) {
cout << "Cannot open INVENTORY file.\n";
```

```
return 1;
}
char item[20];
float cost;
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in.close();
return 0;
}
```

In a way, reading and writing files by using >> and << is like using the C-based functions fprintf( ) and fscanf( ). All information is stored in the file in the same format as it would be displayed on the screen.

Following is another example of disk I/O. This program reads strings entered at the keyboard and writes them to disk. The program stops when the user enters an exclamation point. To use the program, specify the name of the output file on the command line.

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: output <filename>\n";
return 1;
}
ofstream out(argv[1]); // output, normal file
if(!out) {
cout << "Cannot open output file.\n";
return 1;
}
char str[80];
cout << "Write strings to disk. Enter ! to stop.\n";
do {
cout << ": ";
cin >> str;
out << str << endl;
} while (*str != '!');
out.close();
return 0;
}
```

When reading text files using the >> operator, keep in mind that certain character translations will occur. For example, white-space characters are omitted. If you want to prevent any character translations, you must open a file for binary access and use the functions discussed in the next section.

When inputting, if end-of-file is encountered, the stream linked to that file will evaluate as false. (The next section illustrates this fact.)

2.4 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.

Characters vs. Bytes:

Before beginning our examination of unformatted I/O, it is important to clarify an important concept. For many years, I/O in C and C++ was thought of as byte oriented.

This is because a char is equivalent to a byte and the only types of streams available were char streams. However, with the advent of wide characters (of type wchar_t) and their attendant streams, we can no longer say that C++ I/O is byte oriented. Instead, we must say that it is character oriented. Of course, char streams are still byte oriented and we can continue to think in terms of bytes, especially when operating on nontextual data. But the equivalency between a byte and a character can no longer be taken for granted.

As explained in Chapter 20, all of the streams used in this book are char streams since they are by far the most common. They also make unformatted file handling easier because a char stream establishes a one-to-one correspondence between bytes and characters, which is a benefit when reading or writing blocks of binary data.

2.5 put( ) and get( ):

One way that you may read and write unformatted data is by using the member functions get( ) and put( ). These functions operate on characters. That is, get( ) will read a character and put( ) will write a character. Of course if you have opened the file for binary operations and are operating on a char (rather than a wchar_t stream) then these functions read and write bytes of data.

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

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

The get( ) function reads a single character from the invoking stream and puts that value in ch. It returns a reference to the stream. The put( ) function writes ch to the stream and returns a reference to the stream.

The following program displays the contents of any file, whether it contains text or binary data, on the screen. It uses the get( ) function.

```
#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.";
return 1;
}
while(in) { // in will be false when eof is reached
in.get(ch);
if(in) cout << ch;
}
return 0;
}
```

As stated in the preceding section, when the end-of-file is reached, the stream associated with the file becomes false. Therefore, 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 works because get( ) returns a reference to the stream in, and in will be false when the end of the file is encountered.

The next program uses put( ) to write all characters from zero to 255 to a file called CHARS. As you probably know, the ASCII characters occupy only about half the available values that can be held by a char. The other values are generally called the extended character set and include such things as foreign language and mathematical symbols.

(Not all systems support the extended character set, but most do.)

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
int i;
ofstream out("CHARS", ios::out | ios::binary);
if(!out) {
```

```
cout << "Cannot open output file.\n";
return 1;
}
// write all characters to disk
for(i=0; i<256; i++) out.put((char) i);
out.close();
return 0;
}
```

You might find it interesting to examine the contents of the CHARS file to see what extended characters your computer has available.

2.6 read( ) and write( ):

Another way to read and write blocks of binary data is to use C++'s read( ) and write( ) functions. Their prototypes are

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

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

The next program writes a structure to disk and then reads it back in:

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
struct status {
char name[80];
double balance;
unsigned long account_num;
};
int main()
{
struct status acc;
strcpy(acc.name, "Ralph Trantor");
acc.balance = 1123.23;
acc.account_num = 34235678;
// write data
ofstream outbal("balance", ios::out | ios::binary);
if(!outbal) {
cout << "Cannot open file.\n";
```

```
return 1;
}
outbal.write((char *) &acc, sizeof(struct status));
outbal.close();
// now, read back;
ifstream inbal("balance", ios::in | ios::binary);
if(!inbal) {
cout << "Cannot open file.\n";
return 1;
}
inbal.read((char *) &acc, sizeof(struct status));
cout << acc.name << endl;
cout << "Account # " << acc.account_num;
cout.precision(2);
cout.setf(ios::fixed);
cout << endl << "Balance: $" << acc.balance;
inbal.close();
return 0;
}
```

As you can see, only a single call to read( ) or write( ) is necessary to read or write the entire structure. Each individual field need not be read or written separately. As this example illustrates, the buffer can be any type of object.

If the end of the file is reached before num characters have been read, then read( ) simply stops, and the buffer contains as many characters as were available.

2.7 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 is that getline( ) reads and removes the delimiter from the input stream; get( ) does not.

Here is a program that demonstrates the getline( ) function. It reads the contents of a text file one line at a time and displays it on the screen.

```
// Read and display a text file line by line.
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: Display <filename>\n";
return 1;
}
ifstream in(argv[1]); // input
if(!in) {
cout << "Cannot open input file.\n";
return 1;
}
char str[255];
while(in) {
in.getline(str, 255); // delim defaults to '\n'
if(in) cout << str << endl;
}
in.close();
return 0;
}
```

2.8 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.

```
int main()
{
ofstream os(“Example.txt”);
os<<”a”<<endl;
os<<”b”<<endl;
```

```
os.close();

ifstream is("Example.txt");
char c;
while(is.get(c))
   cout<<c;
if(is.eof())
   cout<<"EOF reached"<<endl;
else
   cout<<"Error reading the file"<<endl;
is.close();
}
```

2.9 Random Access:

In C++'s I/O system, you perform random access by using the seekg( ) and seekp( ) functions. Their most common forms are

istream &seekg(off_type offset, seekdir origin);

ostream &seekp(off_type offset, seekdir origin);

Here, 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 defined by ios that determines how the seek will take place.

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 output operation takes place, the appropriate pointer is automatically sequentially advanced. However, using the seekg( ) and seekp( ) functions allows you to access the file in a nonsequential fashion.

The seekg( ) function moves the associated file's current get pointer offset number of characters from the specified origin, which must be one of these three values:

ios::beg Beginning-of-file

ios::cur Current location

ios::end End-of-file

The seekp( ) function moves the associated file's current put pointer offset number of characters from the specified origin, which must be one of the values just shown. 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.

Obtaining the Current File Position:

You can determine the current position of each file pointer by 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. You can use the values returned by tellg( ) and tellp( ) as arguments to the following forms of seekg( ) and seekp( ), respectively.

```
istream &seekg(pos_type pos);
ostream &seekp(pos_type pos);
```

These functions allow you to save the current file location, perform other file operations, and then reset the file location to its previously saved location.

```
#include <iostream>
#include<fstream>
using namespace std;
struct record
{
   char code[6];
   char name[20];
   int i;

}r;
int main()
{
  fstream
file("Temp.dat",ios::trunc|ios::in|ios::out|
ios::binary);
  if(!file)
  {
     cout<<"Cannot open file";
     return 1;
  }
  cout<<"Enter character code, name and
integer value:";
  cin.getline(r.code,6);
  cin.getline(r.name,20);
  cin>>r.i;
  file.write((char *)&r, sizeof(r));

cout<<"\n"<<file.tellg()<<"\n"<<file.tellp();

  file.seekg(3);

cout<<"\n"<<file.tellg()<<"\n"<<file.tellp();

  file.seekp(5);

cout<<"\n"<<file.tellg()<<"\n"<<file.tellp();
  return 0;
}
```

Output:

```
Enter character code, name and integer
value: abc  jack  45
32
32
3
3
5
```

2.10 I/O Status:

The C++ I/O system maintains status information about the outcome of each I/O operation. The current state of the I/O system is held in an object of type iostate, which is an enumeration defined by ios that includes the following 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 nonfatal I/O has occurred 0 otherwise
ios::badbit	1 when a fatal I/O occurs 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 following program illustrates rdstate( ). It displays the contents of a text file.

If an error occurs, the program reports it, using checkstatus( ).

```
#include <iostream>
#include <fstream>
using namespace std;
void checkstatus(ifstream &in);
int main(int argc, char *argv[])
{
if(argc!=2) {
cout << "Usage: Display <filename>\n";
return 1;
}
ifstream in(argv[1]);
if(!in) {
cout << "Cannot open input file.\n";
return 1;
}
char c;
while(in.get(c)) {
if(in) cout << c;
checkstatus(in);
}
checkstatus(in); // check final status
```

```
in.close();
return 0;
}
void checkstatus(ifstream &in)
{
ios::iostate i;
i = in.rdstate();
if(i & ios::eofbit)
cout << "EOF encountered\n";
else if(i & ios::failbit)
cout << "Non-Fatal I/O error\n";
else if(i & ios::badbit)
cout << "Fatal I/O error\n";
}
```

This program will always report one "error." After the while loop ends, the final call to checkstatus( ) reports, as expected, that an EOF has been encountered. You might find the checkstatus( ) function useful in programs that you write.

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

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

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

Once an error has occurred, it may need to be cleared before your program continues.

To do this, use the clear( ) function, which has this prototype:

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

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

1.

get( ):

```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
  char str[10];
  ofstream a_file("Example.txt");
  a_file<<"This text will be inside Example.txt";
  a_file.close();

  ifstream b_file("Example.txt");
  b_file>>str;
  cout<<str<<endl;
  cin.get();
  return 0;
}
```

2.

getline( ):

```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
  string line;
  ifstream myfile("myTest.txt");
  if(myfile.is_open())
  {
     while(!myfile.eof())
     {
        getline(myfile,line);
        cout<<line<<endl;
     }
     myfile.close();
  }
  else
  cout<<"Cannot open file";
  return 0;
}
```

3.

read( ) and write( ):

```
#include<iostream>
#include<fstream>
using namespace std;
class Student
{
   char name[20];
   int mark;
   public:
      void GetStudentData();
      void ShowStudentData();
};
void Student ::  GetStudentData()
{
   cout << "Enter Student Name:" << endl;
   cin >> name;
   cout << "Enter Student Mark:" << endl;
   cin >> mark;
}
void Student ::  ShowStudentData()
{
   cout << "Student Details are:" << endl;
   cout << "Name: " <<  name << endl<< "Mark: " <<  mark << endl;
}
int main(int argc, char *argv[])
{
    char ans='y';
    Student sobj;
    //We open student.dat in append mode
   ofstream out("student.dat", ios::app);
   if(out.is_open())
   {
      //Loop will continue until something other then y is entered
      while( ans == 'y')
      {
         cout << endl << "Continue ?";
         cin >> ans;
         if(ans == 'y')
         {
          sobj.GetStudentData();
```

```
            out.write((char*) & sobj, sizeof(sobj));
          }
       }
    }
    out.close();
    ifstream in("student.dat");
    if(in.is_open())
    {
       while(!in.eof())
       {
          in.read((char*) &sobj, sizeof(sobj));
          if(!in.eof())
          {
            sobj.ShowStudentData();
          }
       }
    }
in.close();
}
```