ot, your

Chapter 21

C++ File I/O

Although C++ I/O forms an integrated system, file I/O is sufficiently specialized that it is generally thought of as a special case, subject to its own constraints and quirks. In part, this is because the most common file is a disk file, and disk files have capabilities and features that most other devices don't. Keep in mind, however, that disk file I/O is simply a special case of the general I/O system and that most of the material discussed in this chapter also applies to streams connected to other types of devices.

<fstream> and the File Classes

To perform file I/O, you must include the header **<fstream>** in your program. It defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. Remember, **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios** (discussed in the preceding chapter). Another class used by the file system is **filebuf**, which provides low-level facilities to manage a file stream. Usually, you don't use **filebuf** directly, but it is part of the other file classes.

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, 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 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. 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 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 seldom 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
```

Note *Depending on your compiler, the mode parameter for* ***fstream::open( )*** *may not default to* ***in | out****. Therefore, you might need to specify this explicitly.*

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

As stated, 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.

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.)

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.

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.

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.

Note *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. Because of C++'s strong type checking, a pointer of one type will not automatically be converted into a pointer of another type.*

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. You can find out how many characters have been read by using another member function, called **gcount( )**, which has this prototype:

streamsize gcount();

It returns the number of characters read by the last binary input operation. The following program shows another example of **read( )** and **write( )** and illustrates the use of **gcount( )**:

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

int main()
{
  double fnum[4] = {99.75, -34.4, 1776.0, 200.1};
  int i;

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

  out.write((char *) &fnum, sizeof fnum);

  out.close();

  for(i=0; i<4; i++) // clear array
    fnum[i] = 0.0;

  ifstream in("numbers", ios::in | ios::binary);
  in.read((char *) &fnum, sizeof fnum);

  // see how many bytes have been read
  cout << in.gcount() << " bytes read\n";

  for(i=0; i<4; i++) // show values read from file
  cout << fnum[i] << " ";
```

```
  in.close();

  return 0;
}
```

The preceding program writes an array of floating-point values to disk and then reads them back. After the call to **read( )**, **gcount( )** is used to determine how many bytes were just read.

More get( ) Functions

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** if the end of the file is encountered. This form of **get( )** is similar to C's **getc( )** 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 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;
}
```

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.

The following program uses **eof()** to display the contents of a file in both hexadecimal and ASCII.

```
/* Display contents of specified file
   in both ASCII and in hex.
*/
#include <iostream>
#include <fstream>
#include <cctype>
#include <iomanip>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: Display <filename>\n";
    return 1;
  }
```

```
  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  register int i, j;
  int count = 0;
  char c[16];

  cout.setf(ios::uppercase);
  while(!in.eof()) {
    for(i=0; i<16 && !in.eof(); i++) {
      in.get(c[i]);
    }
    if(i<16) i--; // get rid of eof

    for(j=0; j<i; j++)
      cout << setw(3) << hex << (int) c[j];
    for(; j<16; j++) cout << "   ";

    cout << "\t";
    for(j=0; j<i; j++)
      if(isprint(c[j])) cout << c[j];
      else cout << ".";

    cout << endl;

    count++;
    if(count==16) {
      count = 0;
      cout << "Press ENTER to continue: ";
      cin.get();
      cout << endl;
    }
  }

  in.close();

  return 0;
}
```

When this program is used to display itself, the first screen looks like this:

```
2F 2A 20 44 69 73 70 6C 61 79 20 63 6F 6E 74 65   /* Display conte
6E 74 73 20 6F 66 20 73 70 65 63 69 66 69 65 64   nts of specified
20 66 69 6C 65  D  A 20 20 20 69 6E 20 62 6F 74    file..    in bot
68 20 41 53 43 49 49 20 61 6E 64 20 69 6E 20 68   h ASCII and in h
65 78 2E  D  A 2A 2F  D  A 23 69 6E 63 6C 75 64   ex..*/..#includ
65 20 3C 69 6F 73 74 72 65 61 6D 3E  D  A 23 69   e <iostream>..#i
6E 63 6C 75 64 65 20 3C 66 73 74 72 65 61 6D 3E   nclude <fstream>
 D  A 23 69 6E 63 6C 75 64 65 20 3C 63 63 74 79   ..#include <ccty
70 65 3E  D  A 23 69 6E 63 6C 75 64 65 20 3C 69   pe>..#include <i
6F 6D 61 6E 69 70 3E  D  A 75 73 69 6E 67 20 6E   omanip>..using n
61 6D 65 73 70 61 63 65 20 73 74 64 3B  D  A  D   amespace std;...
 A 69 6E 74 20 6D 61 69 6E 28 69 6E 74 20 61 72   .int main(int ar
67 63 2C 20 63 68 61 72 20 2A 61 72 67 76 5B 5D   gc, char *argv[]
29  D  A 7B  D  A 20 20 69 66 28 61 72 67 63 21   )..{..  if(argc!
3D 32 29 20 7B  D  A 20 20 20 20 63 6F 75 74 20   =2) {..    cout
3C 3C 20 22 55 73 61 67 65 3A 20 44 69 73 70 6C   << "Usage: Displ
Press ENTER to continue:
```

The ignore( ) Function

You can use the **ignore( )** member function to read and discard characters from the input stream. It has this prototype:

istream &ignore(streamsize *num*=1, int_type *delim*=EOF);

It reads and discards characters until either *num* characters have been ignored (1 by default) or the character specified by *delim* is encountered (**EOF** by default). If the delimiting character is encountered, it is not removed from the input stream. Here, **int_type** is defined as some form of integer.

The next program reads a file called TEST. It ignores characters until either a space is encountered or 10 characters have been read. It then displays the rest of the file.

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

int main()
{
  ifstream in("test");
```

```
  if(!in) {
    cout << "Cannot open file.\n";
    return 1;
  }

  /* Ignore up to 10 characters or until first
     space is found. */
  in.ignore(10, ' ');
  char c;
  while(in) {
    in.get(c);
    if(in) cout << c;
  }

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

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_type peek( );

It returns the next character in the stream or **EOF** if the end of the file is encountered. (**int_type** is defined as some form of integer.)

You can return the last character read from a stream to that stream by using **putback( )**. Its prototype is

istream &putback(char *c*);

where *c* is the last character read.

flush( )

When output is performed, data is not necessarily 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

```
ostream &flush( );
```

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

Note *Closing a file or terminating a program also flushes all buffers.*

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.

The following program demonstrates the **seekp( )** function. It allows you to change a specific character in a file. Specify a filename on the command line, followed by the number of the character in the file you want to change, followed by the new character. Notice that the file is opened for read/write operations.

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

int main(int argc, char *argv[])
{
  if(argc!=4) {
    cout << "Usage: CHANGE <filename> <character> <char>\n";
    return 1;
  }

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

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

  out.put(*argv[3]);
  out.close();

  return 0;
}
```

For example, to use this program to change the twelfth character of a file called TEST to a Z, use this command line:

```
change test 12 Z
```

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

```
#include <iostream>
#include <fstream>
```

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

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

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

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

  in.seekg(atoi(argv[2]), ios::beg);

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

  return 0;
}
```

The following program uses both **seekp( )** and **seekg( )** to reverse the first *<num>* characters in a file.

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

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

```
  fstream inout(argv[1], ios::in | ios::out | ios::binary);

  if(!inout) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  long e, i, j;
  char c1, c2;
  e = atol(argv[2]);

  for(i=0, j=e; i<j; i++, j--) {
    inout.seekg(i, ios::beg);
    inout.get(c1);
    inout.seekg(j, ios::beg);
    inout.get(c2);

    inout.seekp(i, ios::beg);
    inout.put(c2);
    inout.seekp(j, ios::beg);
    inout.put(c1);
  }

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

To use the program, specify the name of the file that you want to reverse, followed by the number of characters to reverse. For example, to reverse the first 10 characters of a file called TEST, use this command line:

```
reverse test 10
```

If the file had contained this:

```
This is a test.
```

it will contain the following after the program executes:

```
a si sihTtest.
```

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.

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 (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 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.

Customized I/O and Files

In Chapter 20 you learned how to overload the insertion and extraction operators relative to your own classes. In that chapter, only console I/O was performed, but because all C++ streams are the same, you can use the same overloaded inserter or extractor function to perform I/O on the console or a file with no changes whatsoever. As an example, the following program reworks the phone book example in Chapter 20 so that it stores a list on disk. The program is very simple: It allows you to add names to the list or to display the list on the screen. It uses custom inserters and extractors to input and output the telephone numbers. You might find it interesting to enhance the program so that it will find a specific number or delete unwanted numbers.

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

class phonebook {
  char name[80];
  char areacode[4];
  char prefix[4];
  char num[5];
public:
  phonebook() { };
  phonebook(char *n, char *a, char *p, char *nm)
  {
    strcpy(name, n);
    strcpy(areacode, a);
    strcpy(prefix, p);
    strcpy(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 << "-";
  stream << 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;
  char c;

  fstream pb("phone", ios::in | ios::out | ios::app);

  if(!pb) {
    cout << "Cannot open phone book file.\n";
    return 1;
  }

  for(;;) {
    do {
      cout << "1. Enter numbers\n";
      cout << "2. Display numbers\n";
      cout << "3. Quit\n";
      cout << "\nEnter a choice: ";
      cin >> c;
    } while(c<'1' || c>'3');

    switch(c) {
      case '1':
        cin >> a;
        cout << "Entry is: ";
        cout << a;  // show on screen
        pb << a;  // write to disk
        break;
      case '2':
        char ch;
        pb.seekg(0, ios::beg);
        while(!pb.eof()) {
          pb.get(ch);
          if(!pb.eof()) cout << ch;
        }
        pb.clear();  // reset eof
        cout << endl;
```

```
        break;
      case '3':
        pb.close();
        return 0;
    }
  }
}
```

Notice that the overloaded << operator can be used to write to a disk file or to the screen without any changes. This is one of the most important and useful features of C++'s approach to I/O.