FSTREAM AND THE FILE CLASSES IN CPP

<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
```

*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.)