Simple File Input/Output

Sometimes keyboard input is not the best choice. For example, suppose you've written a program to analyze stocks, and you've downloaded a file of 1,000 stock prices. It would be far more convenient to have the program read the file directly than to hand-enter all the values. Similarly, it can be convenient to have a program write output to a file so that you have a permanent record of the results.

Fortunately, C++ makes it simple to transfer the skills you've acquired for keyboard input and display output (collectively termed *console I/O*) to file input and output (*file I/O*). Chapter 17 explores these topics more extensively, but we'll look at simple text file I/O now.

Text I/O and Text Files

Let's re-examine the concept of text *I/O*. When you use `cin` for input, the program views input as a series of bytes, with each byte being interpreted as a character code. No matter what the destination data type, the input begins as character data—that is, text data. The `cin` object then has the responsibility of translating text to other types. To see how this works, let's examine how different code handles the same line of input.

Suppose you have the following sample line of input:

```
38.5 19.2
```

Let's see how this line of input is handled by `cin` when used with different data types. First, let's try type `char`:

```
char ch;
cin >> ch;
```

The first character in the input line is assigned to `ch`. In this case, the first character is the digit `3`, and the character code (in binary) for this digit is stored in `ch`. The input and the destination are both characters, so no translation is needed. (Note that it's not the numeric value `3` that is stored; rather, it is the character code for the digit 3.) After the input statement, the digit character `8` is the next character in the input queue and will be the next character examined by the next input operation.

Next, let's try the `int` type with the same input:

```
int n;
cin >> n;
```

In this case, `cin` reads up to the first non-digit character. That is, it reads the `3` digit and the `8` digit, leaving the period as the next character in the input queue. Then `cin` computes that these two characters correspond to the numeric value `38`, and the binary code for `38` is copied to `n`.

Next, let's try the `double` type:

```
double x;
cin >> x;
```

In this case, cin reads up to the first character that's not part of a floating-point number. That is, it reads the 3 digit, the 8 digit, the period character, and the 5 digit, leaving the space as the next character in the input queue. Then cin computes that these four characters correspond to the numeric value 38.5, and the binary code (floating-point format) for 38.5 is copied to x.

Next, let's try the char array type:

```
char word[50];
cin >> word;
```

In this case, cin reads up to the whitespace character. That is, it reads the 3 digit, the 8 digit, the period character, and the 5 digit, leaving the space as the next character in the input queue. Then cin stores the character code for these four characters in the array word and adds a terminating null character. No translation is needed.

Finally, let's try an input variant for the char array type:

```
char word[50];
cin.geline(word,50);
```

In this case, cin reads up through the newline character (the sample input line had fewer than 50 characters). All the characters through the final 2 digit are stored in the array word, and a null character is added. The newline character is discarded, and the next character in the input queue will be the first character on the next line. No translation is needed.

On output, the opposite translations take place. That is, integers are converted to sequences of digit characters, and floating-point numbers are converted to sequences of digits and other characters (for example, 284.53 or -1.587E+06). Character data requires no translation.

The main point to this is that all the input starts out as text. Therefore, the file equivalent to console input is a text file—that is, a file in which each byte stores a character code. Not all files are text files. For example, databases and spreadsheets store numeric data in numeric forms—that is, in binary integer or binary floating-point form. Also, word processing files may contain text information, but they also contain non-text data to describe formatting, fonts, printers, and the like.

The file I/O discussed in this chapter parallels console I/O and thus should be used with text files. To create a text file for input, you use a text editor, such as Notepad for Windows, or vi or emacs for Unix/Linux. You can use a word processor, as long as you save the file in text format. The code editors that are part of IDEs also produce text files; indeed, the source code files are examples of text files. Similarly, you can use text editors to look at files created with text output.

Writing to a Text File

For file output, C++ uses analogs to cout. So to prepare for file output, let's review some basic facts about using cout for console output:

- You must include the `iostream` header file.
- The `iostream` header file defines an `ostream` class for handling output.
- The `iostream` header file declares an `ostream` variable, or object, called `cout`.
- You must account for the `std` namespace; for example, you can use the `using` directive or the `std::` prefix for elements such as `cout` and `endl`.
- You can use `cout` with the `<<` operator to read a variety of data types.

File output parallels this very closely:

- You must include the `fstream` header file.
- The `fstream` header file defines an `ofstream` class for handling output.
- You need to declare one or more `ofstream` variables, or objects, which you can name as you please, as long as you respect the usual naming conventions.
- You must account for the `std` namespace; for example, you can use the `using` directive or the `std::` prefix for elements such as `ofstream`.
- You need to associate a specific `ofstream` object with a specific file; one way to do so is to use the `open()` method.
- When you're finished with a file, you should use the `close()` method to close the file.
- You can use an `ofstream` object with the `<<` operator to output a variety of data types.

Note that although the `iostream` header file provides a predefined `ostream` object called `cout`, you have to declare your own `ofstream` object, choosing a name for it and associating it with a file. Here's how you declare such objects:

```
ofstream outFile;           // outFile an ofstream object
ofstream fout;              // fout an ofstream object
```

Here's how you can associate the objects with particular files:

```
outFile.open("fish.txt");   // outFile used to write to the fish.txt file
char filename[50];
cin >> filename;            // user specifies a name
fout.open(filename);        // fout used to read specified file
```

Note that the `open()` method requires a C-style string as its argument. This can be a literal string or a string stored in an array.

Here's how you can use these objects:

```
double wt = 125.8;
outFile << wt;          // write a number to fish.txt
char line[81] = "Objects are closer than they appear.";
fout << line << endl;   // write a line of text
```

The important point is that after you've declared an `ofstream` object and associated it with a file, you use it exactly as you would use `cout`. All the operations and methods

available to `cout`, such as `<<`, `endl`, and `setf()`, are also available to `ofstream` objects, such as `outFile` and `fout` in the preceding examples.

In short, these are the main steps for using file output:

1. Include the `fstream` header file.
2. Create an `ofstream` object.
3. Associate the `ofstream` object with a file.
4. Use the `ofstream` object in the same manner you would use `cout`.

The program in Listing 6.15 demonstrates this approach. It solicits information from the user, sends output to the display, and then sends the same output to a file. You can use a text editor to examine the output file.

Listing 6.15 **outfile.cpp**

```
// outfile.cpp -- writing to a file
#include <iostream>
#include <fstream>                  // for file I/O

int main()
{
    using namespace std;

    char automobile[50];
    int year;
    double a_price;
    double d_price;

    ofstream outFile;               // create object for output
    outFile.open("carinfo.txt");    // associate with a file

    cout << "Enter the make and model of automobile: ";
    cin.getline(automobile, 50);
    cout << "Enter the model year: ";
    cin >> year;
    cout << "Enter the original asking price: ";
    cin >> a_price;
    d_price = 0.913 * a_price;

// display information on screen with cout

    cout << fixed;
    cout.precision(2);
    cout.setf(ios_base::showpoint);
    cout << "Make and model: " << automobile << endl;
    cout << "Year: " << year << endl;
```

```
    cout << "Was asking $" << a_price << endl;
    cout << "Now asking $" << d_price << endl;

// now do exact same things using outFile instead of cout

    outFile << fixed;
    outFile.precision(2);
    outFile.setf(ios_base::showpoint);
    outFile << "Make and model: " << automobile << endl;
    outFile << "Year: " << year << endl;
    outFile << "Was asking $" << a_price << endl;
    outFile << "Now asking $" << d_price << endl;

    outFile.close();                // done with file
    return 0;
}
```

Note that the final section of the program in Listing 6.15 duplicates the `cout` section, with `cout` replaced by `outFile`. Here is a sample run of this program:

```
Enter the make and model of automobile: Flitz Perky
Enter the model year: 2009
Enter the original asking price: 13500
Make and model: Flitz Perky
Year: 2009
Was asking $13500.00
Now asking $12325.50
```

The screen output comes from using `cout`. If you check the directory or folder that contains the executable program, you should find a new file called `carinfo.txt`. (Or it may be in some other folder, depending on how the compiler is configured.) It contains the output generated by using `outFile`. If you open it with a text editor, you should find the following contents:

```
Make and model: Flitz Perky
Year: 2009
Was asking $13500.00
Now asking $12325.50
```

As you can see, `outFile` sends precisely the same sequence of characters to the `carinfo.txt` file that `cout` sends to the display.

Program Notes

After the program in Listing 6.15 declares an `ofstream` object, you can use the `open()` method to associate the object with a particular file:

```
ofstream outFile;               // create object for output
outFile.open("carinfo.txt");    // associate with a file
```

When the program is done using a file, it should close the connection:

```
outFile.close();
```

Notice that the `close()` method doesn't require a filename. That's because `outFile` has already been associated with a particular file. If you forget to close a file, the program will close it automatically if the program terminates normally.

Notice that `outFile` can use the same methods that `cout` does. Not only can it use the `<<` operator, but it can use the various formatting methods, such as `setf()` and `precision()`. These methods affect only the object that invokes the method. For example, you can provide different values for different objects:

```
cout.precision(2);       // use a precision of 2 for the display
outFile.precision(4);    // use a precision of 4 for file output
```

The main point you should remember is that after you set up an `ofstream` object such as `outFile`, you use it in precisely the same matter as you use `cout`.

Let's go back to the `open()` method:

```
outFile.open("carinfo.txt");
```

In this case, the file `carinfo.txt` does not exist before the program runs. In this circumstance, the `open()` method creates a brand new file by that name. When the file `carinfo.txt` exists, what happens if you run the program again? By default, `open()` first truncates the file; that is, it trims `carinfo.txt` to zero length, discarding the current contents. The contents are then replaced with the new output. Chapter 17 reveals how to override this default behavior.

> **Caution**
>
> When you open an existing file for output, by default it is truncated to a length of zero bytes, so the contents are lost.

It is possible that an attempt to open a file for output might fail. For example, a file having the requested name might already exist and have restricted access. Therefore, a careful programmer would check to see if the attempt succeeded. You'll learn the technique for this in the next example.

Reading from a Text File

Next, let's examine text file input. It's based on console input, which has many elements. So let's begin with a summary those elements:

- You must include the `iostream` header file.
- The `iostream` header file defines an `istream` class for handling input.
- The `iostream` header file declares an `istream` variable, or object, called `cin`.
- You must account for the `std` namespace; for example, you can use the `using` directive or the `std::` prefix for elements such as `cin`.

- You can use `cin` with the `>>` operator to read a variety of data types.
- You can use `cin` with the `get()` method to read individual characters and with the `getline()` method to read a line of characters at a time.
- You can use `cin` with methods such as `eof()` and `fail()` to monitor the success of an input attempt.
- The object `cin` itself, when used as a test condition, is converted to the Boolean value `true` if the last read attempt succeeded and to `false` otherwise.

File input parallels this very closely:

- You must include the `fstream` header file.
- The `fstream` header file defines an `ifstream` class for handling input.
- You need to declare one or more `ifstream` variables, or objects, which you can name as you please, as long as you respect the usual naming conventions.
- You must account for the `std` namespace; for example, you can use the `using` directive or the `std::` prefix for elements such as `ifstream`.
- You need to associate a specific `ifstream` object with a specific file; one way to do so is to use the `open()` method.
- When you're finished with a file, you should use the `close()` method to close the file.
- You can use an `ifstream` object with the `>>` operator to read a variety of data types.
- You can use an `ifstream` object with the `get()` method to read individual characters and with the `getline()` method to read a line of characters at a time.
- You can use an `ifstream` object with methods such as `eof()` and `fail()` to monitor the success of an input attempt.
- An `ifstream` object itself, when used as a test condition, is converted to the Boolean value `true` if the last read attempt succeeded and to `false` otherwise.

Note that although the `iostream` header file provides a predefined `istream` object called `cin`, you have to declare your own `ifstream` object, choosing a name for it and associating it with a file. Here's how you declare such objects:

```
ifstream inFile;             // inFile an ifstream object
ifstream fin;                // fin an ifstream object
```

Here's how you can associate them with particular files:

```
inFile.open("bowling.txt");  // inFile used to read bowling.txt file
char filename[50];
cin >> filename;             // user specifies a name
fin.open(filename);          // fin used to read specified file
```

Note that the `open()` method requires a C-style string as its argument. This can be a literal string or a string stored in an array.

Here's how you can use these objects:

```
double wt;
inFile >> wt;          // read a number from bowling.txt
char line[81];
fin.getline(line, 81); // read a line of text
```

The important point is that after you've declared an `ifstream` object and associated it with a file, you can use it exactly as you would use `cin`. All the operations and methods available to `cin` are also available to `ifstream` objects, such as `inFile` and `fin` in the preceding examples.

What happens if you attempt to open a non-existent file for input? This error causes subsequent attempts to use the `ifstream` object for input to fail. The preferred way to check whether a file was opened successfully is to use the `is_open()` method. You can use code like this:

```
inFile.open("bowling.txt");
if (!inFile.is_open())
{
    exit(EXIT_FAILURE);
}
```

The `is_open()` method returns `true` if the file was opened successfully, so the expression `!inFile.is_open()` is `true` if the attempt fails. The `exit()` function is prototyped in the `cstdlib` header file, which also defines `EXIT_FAILURE` as an argument value used to communicate with the operating system. The `exit()` function terminates the program.

The `is_open()` method is relatively new to C++. If your compiler doesn't support it, you can use the older `good()` method instead. As Chapter 17 discusses, `good()` doesn't check quite as extensively as `is_open()` for possible problems.

The program in Listing 6.16 opens a file specified by the user, reads numbers from the file, and reports the number of values, their sum, and their average. It's important that you design the input loop correctly, and the following "Program Notes" section discusses this in more detail. Notice that this program benefits greatly from using `if` statements.

Listing 6.16 **sumafile.cpp**

```
// sumafile.cpp -- functions with an array argument
#include <iostream>
#include <fstream>          // file I/O support
#include <cstdlib>          // support for exit()
const int SIZE = 60;
int main()
{
    using namespace std;
    char filename[SIZE];
    ifstream inFile;        // object for handling file input
```

```
    cout << "Enter name of data file: ";
    cin.getline(filename, SIZE);
    inFile.open(filename);  // associate inFile with a file
    if (!inFile.is_open())  // failed to open file
    {
        cout << "Could not open the file " << filename << endl;
        cout << "Program terminating.\n";
        exit(EXIT_FAILURE);
    }
    double value;
    double sum = 0.0;
    int count = 0;          // number of items read

    inFile >> value;        // get first value
    while (inFile.good())   // while input good and not at EOF
    {
        ++count;            // one more item read
        sum += value;       // calculate running total
        inFile >> value;    // get next value
    }
    if (inFile.eof())
        cout << "End of file reached.\n";
    else if (inFile.fail())
        cout << "Input terminated by data mismatch.\n";
    else
        cout << "Input terminated for unknown reason.\n";
    if (count == 0)
        cout << "No data processed.\n";
    else
    {
        cout << "Items read: " << count << endl;
        cout << "Sum: " << sum << endl;
        cout << "Average: " << sum / count << endl;
    }
    inFile.close();         // finished with the file
    return 0;
}
```

To use the program in Listing 6.16, you first have to create a text file that contains numbers. You can use a text editor, such as the text editor you use to write source code, to create this file. Let's assume that the file is called `scores.txt` and has the following contents:

```
18 19 18.5 13.5 14
16 19.5 20 18 12 18.5
17.5
```

The program has to be able to find the file. Typically, unless your input includes a pathname with the file, the program will look in the same folder or directory that contains the executable file.

> **Caution**
>
> A Windows text file uses the carriage return character followed by a linefeed character to terminate a line of text. (The usual C++ text mode translates this combination to newline character when reading a file and reverses the translation when writing to a file.) Some text editors, such as the Metrowerks CodeWarrior IDE editor, don't automatically add a this combination to the final line. Therefore, if you use such an editor, you need to press the Enter key after typing the final text and before exiting the file.

Here's a sample run of the program in Listing 6.16:

```
Enter name of data file: scores.txt
End of file reached.
Items read: 12
Sum: 204.5
Average: 17.0417
```

Program Notes

Instead of hard-coding a filename, the program in Listing 6.16 stores a user-supplied name in the character array `filename`. Then the array is used as an argument to `open()`:

```
inFile.open(filename);
```

As discussed earlier in this chapter, it's vital to test whether the attempt to open the file succeeded. Here are a few of the things that might go wrong: The file might not exist, the file might be located in another directory or file folder, access might be denied, and the user might mistype the name or omit a file extension. Many a beginner has spent a long time trying to figure what's wrong with a file-reading loop when the real problem was that the program didn't open the file. Testing for file-opening failure can save you such misspent effort.

You need to pay close attention to the proper design of a file-reading loop. There are several things to test for when reading from a file. First, the program should not try to read past the EOF. The `eof()` method returns `true` if the most recent attempt to read data ran into the EOF. Second, the program might encounter a type mismatch. For instance, Listing 6.16 expects a file containing only numbers. The `fail()` method returns `true` if the most recent read attempt encountered a type mismatch. (This method also returns `true` if the EOF is encountered.) Finally, something unexpected may go wrong—for example, a corrupted file or a hardware failure. The `bad()` method returns `true` if the most recent read attempt encountered such a problem. Rather than test for these conditions individually, it's simpler to use the `good()` method, which returns `true` if nothing when wrong:

```
while (inFile.good())   // while input good and not at EOF
{
    ...
}
```

Then, if you like, you can use the other methods to determine exactly why the loop terminated:

```
if (inFile.eof())
    cout << "End of file reached.\n";
else if (inFile.fail())
    cout << "Input terminated by data mismatch.\n";
else
    cout << "Input terminated for unknown reason.\n";
```

This code comes immediately after the loop so that it investigates why the loop terminated. Because `eof()` tests just for the EOF and `fail()` tests for both the EOF and type mismatch, this code tests for the EOF first. That way, if execution reaches the `else if` test, the EOF has already been excluded, so a `true` value for `fail()` unambiguously identifies type mismatch as the cause of loop termination.

It's particularly important that you understand that `good()` reports on the most recent attempt to read input. That means there should be an attempt to read input *immediately* before applying the test. A standard way of doing that is to have one input statement immediately before the loop, just before the first execution of the loop test, and a second input statement at the end of the loop, just before subsequent executions of the loop test:

```
// standard file-reading loop design
inFile >> value;        // get first value
while (inFile.good())   // while input good and not at EOF
{
    // loop body goes here
    inFile >> value;    // get next value
}
```

You can condense this somewhat by using the fact that the following expression evaluates to `inFile` and that `inFile`, when placed in a context in which a `bool` value is expected, evaluates to `inFile.good()`—that is, to `true` or `false`:

```
inFile >> value
```

Thus, you can replace the two input statements with a single input statement used as a loop test. That is, you can replace the preceding loop structure with this:

```
// abbreviated file-reading loop design
// omit pre-loop input
while (inFile >> value)   // read and test for success
{
    // loop body goes here
    // omit end-of-loop input
}
```

Simple File I/O

Suppose you want a program to write to a file. You must do the following:

1. Create an `ofstream` object to manage the output stream.
2. Associate that object with a particular file.
3. Use the object the same way you would use `cout`; the only difference is that output goes to the file instead of to the screen.

To accomplish this, you begin by including the `fstream` header file. Including this file automatically includes the `iostream` file for most, but not all, implementations, so you may not have to include `iostream` explicitly. Then you declare an `ofstream` object:

```
ofstream fout;      // create an ofstream object named fout
```

The object's name can be any valid C++ name, such as `fout`, `outFile`, `cgate`, or `didi`.

Next, you must associate this object with a particular file. You can do so by using the `open()` method. Suppose, for example, that you want to open the `jar.txt` file for output. You would do the following:

```
fout.open("jar.txt");  // associate fout with jar.txt
```

You can combine these two steps (creating the object and associating a file) into a single statement by using a different constructor:

```
ofstream fout("jar.txt");  // create fout object, associate it with jar.txt
```

When you've gotten this far, you use `fout` (or whatever name you choose) in the same manner as `cout`. For example, if you want to put the words `Dull Data` into the file, you can do the following:

```
fout << "Dull Data";
```

Indeed, because `ostream` is a base class for the `ofstream` class, you can use all the `ostream` methods, including the various insertion operator definitions and the formatting methods and manipulators. The `ofstream` class uses buffered output, so the program allocates space for an output buffer when it creates an `ofstream` object such as `fout`. If you create two `ofstream` objects, the program creates two buffers, one for each object. An `ofstream` object such as `fout` collects output byte-by-byte from the program; then, when the buffer is filled, it transfers the buffer contents en masse to the destination file. Because disk drives are designed to transfer data in larger chunks, not byte-by-byte, the buffered approach greatly speeds up the transfer rate of data from a program to a file.

Opening a file for output this way creates a new file if there is no file of that name. If a file by that name exists prior to opening it for output, the act of opening it truncates it so that output starts with a clean file. Later in this chapter you'll see how to open an existing file and retain its contents.

Caution

Opening a file for output in the default mode automatically truncates the file to zero size, in effect disposing of the prior contents.

The requirements for reading a file are much like those for writing to a file:

1. Create an `ifstream` object to manage the input stream.
2. Associate that object with a particular file.
3. Use the object the same way you would use `cin`.

The steps for reading a file are similar to those for writing to a file. First, of course, you include the `fstream` header file. Then you declare an `ifstream` object and associate it with the filename. You can do so in two statements or one:

```
// two statements
ifstream fin;              // create ifstream object called fin
fin.open("jellyjar.txt");  // open jellyjar.txt for reading
// one statement
ifstream fis("jamjar.txt"); // create fis and associate with jamjar.txt
```

You can then use `fin` or `fis` much as you would use `cin`. For example, you can use the following:

```
char ch;
fin >> ch;               // read a character from the jellyjar.txt file
char buf[80];
fin >> buf;              // read a word from the file
fin.getline(buf, 80);    // read a line from the file
string line;
getline(fin, line);      // read from a file to a string object
```

Input, like output, is buffered, so creating an `ifstream` object such as `fin` creates an input buffer, which the `fin` object manages. As with output, buffering moves data much faster than byte-by-byte transfer.

The connections with a file are closed automatically when the input and output stream objects expire—for example, when the program terminates. Also you can close a connection with a file explicitly by using the `close()` method:

```
fout.close();    // close output connection to file
fin.close();     // close input connection to file
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. However, the stream management apparatus remains in place. For example, the `fin` object still exists, along with the input buffer it manages. As you'll see later, you can reconnect the stream to the same file or to another file.

Let's look at a short example. The program in Listing 17.16 asks for a filename. It creates a file that has that name, writes some information to it, and closes the file. Closing the file flushes the buffer, guaranteeing that the file is updated. Then the program opens the

same file for reading and displays its contents. Note that the program uses `fin` and `fout` in the same manner as you'd use `cin` and `cout`. Also, the program reads the filename into a `string` object and then uses the `c_str()` method to provide C-style string arguments for the `ofstream` and `ifstream` constructors.

Listing 17.16 **fileio.cpp**

```
// fileio.cpp -- saving to a file
#include <iostream> // not needed for many systems
#include <fstream>
#include <string>

int main()
{
    using namespace std;
    string filename;

    cout << "Enter name for new file: ";
    cin >> filename;

// create output stream object for new file and call it fout
    ofstream fout(filename.c_str());

    fout << "For your eyes only!\n";        // write to file
    cout << "Enter your secret number: ";   // write to screen
    float secret;
    cin >> secret;
    fout << "Your secret number is " << secret << endl;
    fout.close();           // close file

// create input stream object for new file and call it fin
    ifstream fin(filename.c_str());
    cout << "Here are the contents of " << filename << ":\n";
    char ch;
    while (fin.get(ch))     // read character from file and
        cout << ch;         // write it to screen
    cout << "Done\n";
    fin.close();

    return 0;
}
```

Here is a sample run of the program in Listing 17.16:

```
Enter name for new file: pythag
Enter your secret number: 3.14159
```

```
Here are the contents of pythag:
For your eyes only!
Your secret number is 3.14159
Done
```

If you check the directory that contains the program, you should find a file named `pythag`, and any text editor should show the same contents that the program output displays. (So much for secrecy.)

Stream Checking and `is_open()`

The C++ file stream classes inherit a stream-state member from the `ios_base` class. This member, as discussed earlier, stores information that reflects the stream status: All is well, end-of-file has been reached, I/O operation failed, and so on. If all is well, the stream state is zero (no news is good news). The various other states are recorded by setting particular bits to `1`. The file stream classes also inherit the `ios_base` methods that report about the stream state and that are summarized in Table 17.4. You can check the stream state to find whether the most recent stream operation succeeded or failed. For file streams, this includes checking the success of an attempt to open a file. For example, attempting to open a non-existent file for input sets `failbit`. So you could check this way:

```
fin.open(argv[file]);
if (fin.fail())  // open attempt failed
{
   ...
}
```

Or because an `ifstream` object, like an `istream` object, is converted to a `bool` type where a `bool` type is expected, you could use this:

```
fin.open(argv[file]);
if (!fin)  // open attempt failed
{
   ...
}
```

However, newer C++ implementations have a better way to check whether a file has been opened—the `is_open()` method:

```
if (!fin.is_open())  // open attempt failed
{
   ...
}
```

The reason this is better is that it tests for some subtle problems that the other forms miss, as discussed in the following Caution.

Caution

In the past, the usual tests for successful opening of a file were the following:

```
if(fin.fail()) ...  // failed to open
if(!fin.good()) ... // failed to open
if (!fin) ...       // failed to open
```

The `fin` object, when used in a test condition, is converted to `false` if `fin.good()` is `false` and to `true` otherwise, so the two forms are equivalent. However, these tests fail to detect one circumstance, which is attempting to open a file by using an inappropriate file mode (see the "File Modes" section, later in this chapter). The `is_open()` method catches this form of error, along with those caught by the `good()` method. However, older C++ implementations do not have `is_open()`.

Opening Multiple Files

You might require that a program open more than one file. The strategy for opening multiple files depends on how they will be used. If you need two files open simultaneously, you must create a separate stream for each file. For example, a program that collates two sorted files into a third file would create two `ifstream` objects for the two input files and an `ofstream` object for the output file. The number of files you can open simultaneously depends on the operating system.

However, you may plan to process a group of files sequentially. For example, you might want to count how many times a name appears in a set of 10 files. In this case, you can open a single stream and associate it with each file in turn. This conserves computer resources more effectively than opening a separate stream for each file. To use this approach, you declare an `ifstream` object without initializing it and then use the `open()` method to associate the stream with a file. For example, this is how you could handle reading two files in succession:

```
ifstream fin;           // create stream using default constructor
fin.open("fat.txt");    // associate stream with fat.txt file
...                     // do stuff
fin.close();            // terminate association with fat.txt
fin.clear();            // reset fin (may not be needed)
fin.open("rat.txt");    // associate stream with rat.txt file
...
fin.close();
```

We'll look at an example shortly, but first, let's examine a technique for feeding a list of files to a program in a manner that allows the program to use a loop to process them.

Command-Line Processing

File-processing programs often use command-line arguments to identify files. *Command-line arguments* are arguments that appear on the command line when you type a

command. For example, to count the number of words in some files on a Unix or Linux system, you would type this command at the command-line prompt:

```
wc report1 report2 report3
```

Here `wc` is the program name, and `report1`, `report2`, and `report3` are filenames passed to the program as command-line arguments.

C++ has a mechanism for letting a program running in a command-line environment access the command-line arguments. You can use the following alternative function heading for `main()`:

```
int main(int argc, char *argv[])
```

The `argc` argument represents the number of arguments on the command line. The count includes the command name itself. The `argv` variable is a pointer to a pointer to a `char`. This sounds a bit abstract, but you can treat `argv` as if it were an array of pointers to the command-line arguments, with `argv[0]` being a pointer to the first character of a string holding the command name, `argv[1]` being a pointer to the first character of a string holding the first command-line argument, and so on. That is, `argv[0]` is the first string from the command line, and so on. For example, suppose you have the following command line:

```
wc report1 report2 report3
```

In this case, `argc` would be `4`, `argv[0]` would be `wc`, `argv[1]` would be `report1`, and so on. The following loop would print each command-line argument on a separate line:

```
for (int i = 1; i < argc; i++)
       cout << argv[i] << endl;
```

Starting with `i = 1` just prints the command-line arguments; starting with `i = 0` would print the command name as well.

Command-line arguments, of course, go hand-in-hand with command-line operating systems such as the Windows command prompt mode, Unix, and Linux. Other setups may still allow you to use command-line arguments:

- Many Windows IDEs (Integrated Development Environments) have an option for providing command-line arguments. Typically, you have to navigate through a series of menu choices that lead to a box into which you can type the command-line arguments. The exact set of steps varies from vendor to vendor and from upgrade to upgrade, so check your documentation.
- Many Windows IDEs can produce executable files that run under in the Windows command prompt mode.

Listing 17.17 combines the command-line technique with file stream techniques to count characters in files listed on the command line.

Listing 17.17 **count.cpp**

```
// count.cpp -- counting characters in a list of files
#include <iostream>
#include <fstream>
#include <cstdlib>   // for exit()
int main(int argc, char * argv[])
{
    using namespace std;
    if (argc == 1)          // quit if no arguments
    {
        cerr << "Usage: " << argv[0] << " filename[s]\n";
        exit(EXIT_FAILURE);
    }

    ifstream fin;              // open stream
    long count;
    long total = 0;
    char ch;

    for (int file = 1; file < argc; file++)
    {
        fin.open(argv[file]);  // connect stream to argv[file]
        if (!fin.is_open())
        {
            cerr << "Could not open " << argv[file] << endl;
            fin.clear();
            continue;
        }
        count = 0;
        while (fin.get(ch))
            count++;
        cout << count << " characters in " << argv[file] << endl;
        total += count;
        fin.clear();           // needed for some implementations
        fin.close();           // disconnect file
    }
    cout << total << " characters in all files\n";

    return 0;
}
```

Note

Some C++ implementations require using `fin.clear()` at the end of the program, and others do not. It depends on whether associating a new file with the `ifstream` object automatically resets the stream state. In does no harm to use `fin.clear()` even if it isn't needed.

On a Linux system, for example, you could compile Listing 17.17 to an executable file called `a.out`. Then sample runs could look like this:

```
$ a.out
Usage: a.out filename[s]
$ a.out paris rome
3580 characters in paris
4886 characters in rome
8466 characters in all files
$
```

Note that the program uses `cerr` for the error message. A minor point is that the message uses `argv[0]` instead of `a.out`:

```
cerr << "Usage: " << argv[0] << " filename[s]\n";
```

This way, if you change the name of the executable file, the program will automatically use the new name.

The program uses the `is_open()` method to verify that it was able to open the requested file. Let's examine that matter further.

File Modes

The file mode describes how a file is to be used: read it, write to it, append it, and so on. When you associate a stream with a file, either by initializing a file stream object with a filename or by using the `open()` method, you can provide a second argument that specifies the file mode:

```
ifstream fin("banjo", mode1);  // constructor with mode argument
ofstream fout();
fout.open("harp", mode2);      // open() with mode arguments
```

The `ios_base` class defines an `openmode` type to represent the mode; like the `fmtflags` and `iostate` types, it is a `bitmask` type. (In the old days, it was type `int`.) You can choose from several constants defined in the `ios_base` class to specify the mode. Table 17.7 lists the constants and their meanings. C++ file I/O has undergone several changes to make it compatible with ANSI C file I/0.

Table 17.7 **File Mode Constants**

Constant	Meaning
`ios_base::in`	Open file for reading.
`ios_base::out`	Open file for writing.
`ios_base::ate`	Seek to end-of-file upon opening file.
`ios_base::app`	Append to end-of-file.
`ios_base::trunc`	Truncate file if it exists.
`ios_base::binary`	Binary file.

If the `ifstream` and `ofstream` constructors and the `open()` methods each take two arguments, how have we gotten by using just one in the previous examples? As you have probably guessed, the prototypes for these class member functions provide default values for the second argument (the file mode argument). For example, the `ifstream open()` method and constructor use `ios_base::in` (open for reading) as the default value for the mode argument, and the `ofstream open()` method and constructor use `ios_base::out | ios_base::trunc` (open for writing and truncate the file) as the default. The bitwise OR operator (|) is used to combine two bit values into a single value that can be used to set both bits. The `fstream` class doesn't provide a mode default, so you have to provide a mode explicitly when creating an object of that class.

Note that the `ios_base::trunc` flag means an existing file is truncated when opened to receive program output; that is, its previous contents are discarded. Although this behavior commendably minimizes the danger of running out of disk space, you can probably imagine situations in which you don't want to wipe out a file when you open it. Of course, C++ provides other choices. If, for example, you want to preserve the file contents and add (append) new material to the end of the file, you can use the `ios_base::app` mode:

```
ofstream fout("bagels", ios_base::out | ios_base::app);
```

Again, the code uses the | operator to combine modes. So `ios_base::out | ios_base::app` means to invoke both the `out` mode and the `app` mode (see Figure 17.6).

You can expect to find some differences among older C++ implementations. For example, some allow you to omit the `ios_base::out` in the previous example, and some don't. If you aren't using the default mode, the safest approach is to provide all the mode elements explicitly. Some compilers don't support all the choices in Table 17.6, and some may offer choices beyond those in the table. One consequence of these differences is that you may have to make some alterations in the following examples to use them on your system. The good news is that the development of the C++ Standard is providing greater uniformity.

Standard C++ defines parts of file I/O in terms of ANSI C standard I/O equivalents. A C++ statement like

```
ifstream fin(filename, c++mode);
```

is implemented as if it uses the C `fopen()` function:

```
fopen(filename, cmode);
```

Here *c++mode* is a type `openmode` value, such as `ios_base::in`, and *cmode* is the corresponding C-mode string, such as `"r"`. Table 17.8 shows the correspondence between C++ modes and C modes. Note that `ios_base::out` by itself causes truncation but that it doesn't cause truncation when combined with `ios_base::in`. Unlisted combinations, such as `ios_base::in | ios_base::trunc`, prevent the file from being opened. The `is_open()` method detects this failure.

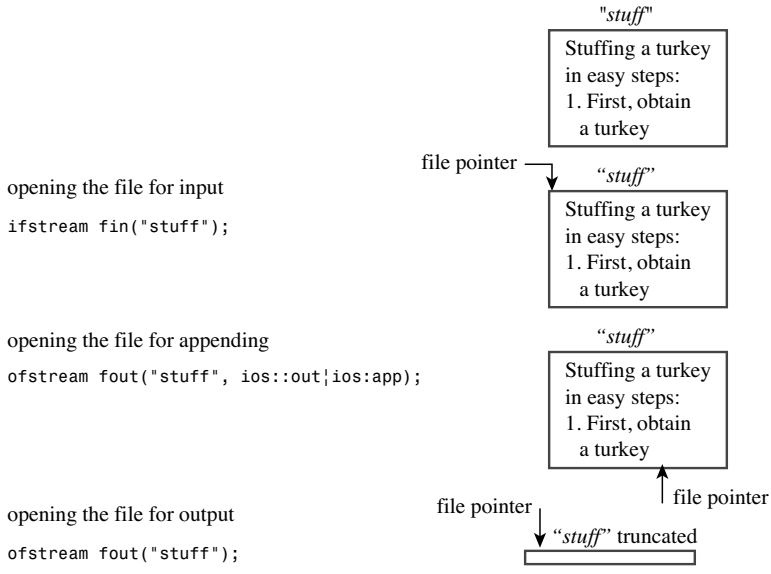

Figure 17.6 Some file-opening modes.

Table 17.8 **C++ and C File-Opening Modes**

C++ mode	C mode	Meaning
`ios_base::in`	`"r"`	Open for reading.
`ios_base::out`	`"w"`	(Same as `ios_base::out | ios_base::trunc`.)
`ios_base::out | ios_base::trunc`	`"w"`	Open for writing, truncating file if it already exists.
`ios_base::out | ios_base::app`	`"a"`	Open for writing, append only.
`ios_base::in | ios_base::out`	`"r+"`	Open for reading and writing, with writing permitted anywhere in the file.
`ios_base::in | ios_base::out | ios_base::trunc`	`"w+"`	Open for reading and writing, first truncating file if it already exists.
`c++mode` `| ios_base::binary`	`"`*`cmodeb`*`"`	Open in *`c++mode`* or corresponding *`cmode`* and in binary mode; for example, `ios_base::in | ios_base::binary` becomes `"rb"`.
`c++mode` `| ios_base::ate`	`"`*`cmode`*`"`	Open in indicated mode and go to end of file. C uses a separate function call instead of a mode code. For example, `ios_base::in | ios_base::ate` translates to the `"r"` mode and the C function call `fseek(file, 0, SEEK_END)`.

Note that both `ios_base::ate` and `ios_base::app` place you (or, more precisely, a file pointer) at the end of the file just opened. The difference between the two is that the `ios_base::app` mode allows you to add data to the end of the file only, while the `ios_base::ate` mode merely positions the pointer at the end of the file.

Clearly, there are many possible combinations of modes. We'll look at a few representative ones.

Appending to a File

Let's look at a program that appends data to the end of a file. The program maintains a file that contains a guest list. When the program begins, it displays the current contents of the file, if it exists. It can use the `is_open()` method after attempting to open the file to check whether the file exists. Next, the program opens the file for output, using the `ios_base::app` mode. Then it solicits input from the keyboard to add to the file. Finally, the program displays the revised file contents. Listing 17.18 illustrates how to accomplish these goals. Note how the program uses the `is_open()` method to test whether the file has been opened successfully.

> **Note**
>
> File I/O was perhaps the least standardized aspect of C++ in its earlier days, and many older compilers don't quite conform to the current standard. Some, for example, used modes such as `nocreate` that are not part of the current standard. Also only some compilers require the `fin.clear()` call before opening the same file a second time for reading.

Listing 17.18 **append.cpp**

```
// append.cpp -- appending information to a file
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>      // (for exit()

const char * file = "guests.txt";
int main()
{
    using namespace std;
    char ch;
```

```
// show initial contents
    ifstream fin;
    fin.open(file);

    if (fin.is_open())
    {
        cout << "Here are the current contents of the "
             << file << " file:\n";
        while (fin.get(ch))
            cout << ch;
        fin.close();
    }

// add new names
    ofstream fout(file, ios::out | ios::app);
    if (!fout.is_open())
    {
        cerr << "Can't open " << file << " file for output.\n";
        exit(EXIT_FAILURE);
    }

    cout << "Enter guest names (enter a blank line to quit):\n";
    string name;
    while (getline(cin,name) && name.size() > 0)
    {
          fout << name << endl;
    }
    fout.close();

// show revised file
    fin.clear();    // not necessary for some compilers
    fin.open(file);
    if (fin.is_open())
    {
        cout << "Here are the new contents of the "
             << file << " file:\n";
        while (fin.get(ch))
            cout << ch;
        fin.close();
   }
    cout << "Done.\n";
    return 0;

}
```

Here's a sample first run of the program in Listing 17.18:

```
Enter guest names (enter a blank line to quit):
Genghis Kant
Hank Attila
Charles Bigg

Here are the new contents of the guests.txt file:
Genghis Kant
Hank Attila
Charles Bigg
Done.
```

At this point the `guests.txt` file hasn't been created, so the program doesn't preview the file.

Next time the program is run, however, the `guests.txt` file does exist, so the program does preview the file. Also note that the new data are appended to the old file contents rather than replacing them:

```
Here are the current contents of the guests.txt file:
Genghis Kant
Hank Attila
Charles Bigg
Enter guest names (enter a blank line to quit):
Greta Greppo
LaDonna Mobile
Fannie Mae

Here are the new contents of the guests.txt file:
Ghengis Kant
Hank Attila
Charles Bigg
Greta Greppo
LaDonna Mobile
Fannie Mae
Done.
```

You should be able to read the contents of `guest.txt` with any text editor, including the editor you use to write your source code.

Binary Files

When you store data in a file, you can store the data in text form or in binary format. Text form means you store everything as text, even numbers. For example, storing the value -2.324216e+07 in text form means storing the 13 characters used to write this number. That requires converting the computer's internal representation of a floating-point number to character form, and that's exactly what the `<<` insertion operator does. Binary format, on the other hand, means storing the computer's internal representation of a value.

That is, instead of storing characters, the computer stores the (typically) 64-bit `double` representation of the value. For a character, the binary representation is the same as the text representation—the binary representation of the character's ASCII code (or equivalent). For numbers, however, the binary representation is much different from the text representation (see Figure 17.7).

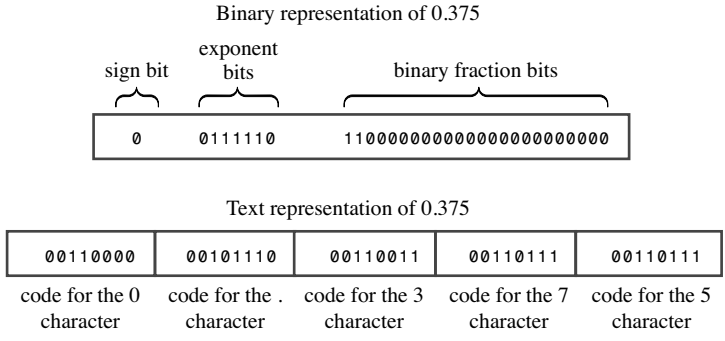

Figure 17.7 Binary and text representations of a floating-point number.

Each format has advantages. The text format is easy to read. With it, you can use an ordinary editor or word processor to read and edit a text file. You can easily transfer a text file from one computer system to another. The binary format is more accurate for numbers because it stores the exact internal representation of a value. There are no conversion errors or round-off errors. Saving data in binary format can be faster because there is no conversion and because you may be able to save data in larger chunks. And the binary format usually takes less space, depending on the nature of the data. Transferring to another system can be a problem, however, if the new system uses a different internal representation for values. Even different compilers on the same system may use different internal representations for structure layouts. In these cases, you (or someone) may have to write a program to translate one data format to another.

Let's look at a more concrete example. Consider the following structure definition and declaration:

```
const int LIM = 20;
struct planet
{
    char name[LIM];       // name of planet
    double population;    // its population
    double g;             // its acceleration of gravity
};
planet pl;
```

To save the contents of the structure `pl` in text form, you can use this:

```
ofstream fout("planets.dat", ios_base:: out | ios_base::app);
fout << pl.name << " " << pl.population << " " << pl.g << "\n";
```

Note that you have to provide each structure member explicitly by using the membership operator, and you have to separate adjacent data for legibility. If the structure contains, say, 30 members, this could get tedious.

To save the same information in binary format, you can use this:

```
ofstream fout("planets.dat",
              ios_base:: out | ios_base::app | ios_base::binary);
fout.write( (char *) &pl, sizeof pl);
```

This code saves the entire structure as a single unit, using the computer's internal representation of data. You won't be able to read the file as text, but the information will be stored more compactly and precisely than as text. And it is certainly easier to type the code. This approach makes two changes:

- It uses a binary file mode.
- It uses the `write()` member function.

Let's examine these changes more closely.

Some systems, such as Windows, support two file formats: text and binary. If you want to save data in binary form, you should use the binary file format. In C++ you do so by using the `ios_base::binary` constant in the file mode. If you want to know why you should do this on a Windows system, check the discussion in the following sidebar, "Binary Files and Text Files."

Binary Files and Text Files

Using a binary file mode causes a program to transfer data from memory to a file, or vice versa, without any hidden translation taking place. Such is not necessarily the case for the default text mode. For example, consider Windows text files. They represent a newline with a two-character combination: carriage return, linefeed. Macintosh text files represent a newline with a carriage return. Unix and Linux files represent a newline with a linefeed. C++, which grew up on Unix, also represents a newline with a linefeed. For portability, a Windows C++ program automatically translates the C++ newline to a carriage return and linefeed when writing to a text mode file; and a Macintosh C++ program translates the newline to a carriage return when writing to a file. When reading a text file, these programs convert the local newline back to the C++ form. The text format can cause problems with binary data because a byte in the middle of a `double` value could have the same bit pattern as the ASCII code for the newline character. Also there are differences in how end-of-file is detected. So you should use the binary file mode when saving data in binary format. (Unix systems have just one file mode, so on them the binary mode is the same as the text mode.)

To save data in binary form instead of text form, you can use the `write()` member function. This method, recall, copies a specified number of bytes from memory to a file. This chapter used it earlier to copy text, but it will copy any type of data byte-by-byte

with no conversion. For example, if you pass to it the address of a `long` variable and tell it to copy 4 bytes, it will copy the 4 bytes constituting the `long` value verbatim to a file and not convert it to text. The only awkwardness is that you have to type cast the address to type pointer-to-`char`. You can use the same approach to copy an entire `planet` structure. To get the number of bytes, you use the `sizeof` operator:

```
fout.write( (char *) &pl, sizeof pl);
```

This statement causes the program to go to the address of the `pl` structure and copy the 36 bytes (the value of the `sizeof pl` expression) beginning at this address to the file connected to `fout`.

To recover the information from a file, you use the corresponding `read()` method with an `ifstream` object:

```
ifstream fin("planets.dat", ios_base::in | ios_base::binary);
fin.read((char *) &pl, sizeof pl);
```

This copies `sizeof pl` bytes from the file to the `pl` structure. This same approach can be used with classes that don't use virtual functions. In that case, just the data members are saved, not the methods. If the class does have virtual methods, then a hidden pointer to a table of pointers to virtual functions is also copied. Because the next time you run the program it might locate the virtual function table at a different location, copying old pointer information into objects from a file can create havoc. (Also see the Note in Programming Exercise 6.)

Tip

The `read()` and `write()` member functions complement each other. You use `read()` to recover data that has been written to a file with `write()`.

Listing 17.19 uses these methods to create and read a binary file. In form, the program is similar to Listing 17.18, but it uses `write()` and `read()` instead of the insertion operator and the `get()` method. It also uses manipulators to format the screen output.

Note

Although the binary file concept is part of ANSI C, some C and C++ implementations do not provide support for the binary file mode. The reason for this oversight is that some systems have only one file type in the first place, so you can use binary operations such as `read()` and `write()` with the standard file format. Therefore, if your implementation rejects `ios_base::binary` as a valid constant, you can just omit it from your program. If your implementation doesn't support the `fixed` and `right` manipulators, you can use `cout.setf(ios_base::fixed, ios_base::floatfield)` and `cout.setf(ios_base::right, ios_base::adjustfield)`. Also you may have to substitute `ios` for `ios_base`. Other compilers, particularly older ones, may have other idiosyncrasies.

Listing 17.19 **binary.cpp**

```
// binary.cpp -- binary file I/O
#include <iostream> // not required by most systems
#include <fstream>
#include <iomanip>
#include <cstdlib>  // for exit()

inline void eatline() { while (std::cin.get() != '\n') continue; }
struct planet
{
    char name[20];      // name of planet
    double population;  // its population
    double g;           // its acceleration of gravity
};

const char * file = "planets.dat";

int main()
{
    using namespace std;
    planet pl;
    cout << fixed << right;

// show initial contents
    ifstream fin;
    fin.open(file, ios_base::in |ios_base::binary);  // binary file
    //NOTE: some systems don't accept the ios_base::binary mode
    if (fin.is_open())
    {
    cout << "Here are the current contents of the "
        << file << " file:\n";
    while (fin.read((char *) &pl, sizeof pl))
    {
        cout << setw(20) << pl.name << ": "
              << setprecision(0) << setw(12) << pl.population
              << setprecision(2) << setw(6) << pl.g << endl;
    }
    fin.close();
    }

// add new data
    ofstream fout(file,
             ios_base::out | ios_base::app | ios_base::binary);
    //NOTE: some systems don't accept the ios::binary mode
    if (!fout.is_open())
    {
```

```
        cerr << "Can't open " << file << " file for output:\n";
        exit(EXIT_FAILURE);
    }

    cout << "Enter planet name (enter a blank line to quit):\n";
    cin.get(pl.name, 20);
    while (pl.name[0] != '\0')
    {
        eatline();
        cout << "Enter planetary population: ";
        cin >> pl.population;
        cout << "Enter planet's acceleration of gravity: ";
        cin >> pl.g;
        eatline();
        fout.write((char *) &pl, sizeof pl);
        cout << "Enter planet name (enter a blank line "
                "to quit):\n";
        cin.get(pl.name, 20);
    }
    fout.close();

// show revised file
    fin.clear();    // not required for some implementations, but won't hurt
    fin.open(file, ios_base::in | ios_base::binary);
    if (fin.is_open())
    {
        cout << "Here are the new contents of the "
             << file << " file:\n";
        while (fin.read((char *) &pl, sizeof pl))
        {
            cout << setw(20) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << endl;
        }
        fin.close();
    }
    cout << "Done.\n";
    return 0;
}
```

Here is a sample initial run of the program in Listing 17.19:

```
Enter planet name (enter a blank line to quit):
Earth
Enter planetary population: 6928198253
Enter planet's acceleration of gravity: 9.81
```

```
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
               Earth:   6928198253  9.81
Done.
```

And here is a sample follow-up run:

```
Here are the current contents of the planets.dat file:
               Earth:   6928198253  9.81
Enter planet name (enter a blank line to quit):
Jenny's World
Enter planetary population: 32155648
Enter planet's acceleration of gravity: 8.93
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
               Earth:   6928198253  9.81
       Jenny's World:     32155648  8.93
Done.
```

You've already seen the major features of the program, but let's re-examine an old point. The program uses this code (in the form of the inline `eatline()` function) after reading the planet's `g` value:

```
while (std::cin.get() != '\n') continue;
```

This reads and discards input up through the newline character. Consider the next input statement in the loop:

```
cin.get(pl.name, 20);
```

If the newline were left in place, this statement would read the newline as an empty line, terminating the loop.

You might wonder if this program could use a `string` object instead of a character array for the `name` member of the `planet` structure. The answer is no—at least not without major changes in design. The problem is that a `string` object doesn't actually contain the string within itself; instead, it contains a pointer to the memory location where the string is stored. So if you copy the structure to a file, you don't copy the string data, you just copy the address of where the string was stored. When you run the program again, that address is meaningless.

Random Access

For our last file example, let's look at random access. *Random access* means moving directly to any location in the file instead of moving through it sequentially. The random access approach is often used with database files. A program will maintain a separate index file, giving the location of data in the main data file. Then it can jump directly to that location, read the data there, and perhaps modify it. This approach is done most simply if the file

consists of a collection of equal-sized records. Each record represents a related collection of data. For example, in the example in Listing 17.19, each file record would represent all the data about a particular planet. A file record corresponds rather naturally to a program structure or class.

This example is based on the binary file program in Listing 17.19, to take advantage of the fact that the `planet` structure provides a pattern for a file record. To add to the creative tension of programming, the example opens the file in a read-and-write mode so that it can both read and modify a record. You can do this by creating an `fstream` object. The `fstream` class derives from the `iostream` class, which, in turn, is based on both the `istream` and `ostream` classes, so it inherits the methods of both. It also inherits two buffers, one for input and one for output, and synchronizes the handling of the two buffers. That is, as the program reads the file or writes to it, it moves both an input pointer in the input buffer and an output pointer in the output buffer in tandem.

The example does the following:

1. Displays the current contents of the `planets.dat` file.
2. Asks which record you want to modify.
3. Modifies that record.
4. Shows the revised file.

A more ambitious program would use a menu and a loop to let you select from the list of actions indefinitely, but this version performs each action just once. This simplified approach allows you to examine several aspects of read/write files without getting bogged down in matters of program design.

Caution

This program assumes that the `planets.dat` file already exists and was created by the `binary.cpp` program in Listing 17.19.

The first question to answer is what file mode to use. In order to read the file, you need the `ios_base::in` mode. For binary I/O, you need the `ios_base::binary` mode. (Again, on some nonstandard systems you can omit—indeed, you may have to omit—this mode.) In order to write to the file, you need the `ios_base::out` or the `ios_base::app` mode. However, the append mode allows a program to add data to the end of the file only. The rest of the file is read-only; that is, you can read the original data but not modify it—so, to be able to modify the data, you have to use `ios_base::out`. As Table 17.8 indicates, using the `in` and `out` modes simultaneously provides a read/write mode, so you just have to add the binary element. As mentioned earlier, you use the `|` operator to combine modes. Thus, you need the following statement to set up business:

```
finout.open(file,ios_base::in | ios_base::out | ios_base::binary);
```

Next, you need a way to move through a file. The `fstream` class inherits two methods for this: `seekg()` moves the input pointer to a given file location, and `seekp()` moves the output pointer to a given file location. (Actually, because the `fstream` class uses buffers

for intermediate storage of data, the pointers point to locations in the buffers, not in the actual file.) You can also use `seekg()` with an `ifstream` object and `seekp()` with an `ofstream` object. Here are the `seekg()` prototypes:

```
basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);
basic_istream<charT,traits>& seekg(pos_type);
```

As you can see, they are templates. This chapter uses a template specialization for the `char` type. For the `char` specialization, the two prototypes are equivalent to the following:

```
istream & seekg(streamoff, ios_base::seekdir);
istream & seekg(streampos);
```

The first prototype represents locating a file position measured, in bytes, as an offset from a file location specified by the second argument. The second prototype represents locating a file position measured, in bytes, from the beginning of a file.

Type Escalation

When C++ was young, life was simpler for the `seekg()` methods. The `streamoff` and `streampos` types were `typedef`s for some standard integer type, such as `long`. However, the quest for creating a portable standard had to deal with the realization that an integer argument might not provide enough information for some file systems, so `streamoff` and `streampos` were allowed to be structure or class types, as long as they allowed some basic operations, such as using an integer value as an initialization value. Next, the old `istream` class was replaced with the `basic_istream` template, and `streampos` and `streamoff` were replaced with the template-based types `pos_type` and `off_type`. However, `streampos` and `streamoff` continue to exist as `char` specializations of `pos_type` and `off_type`. Similarly, you can use the `wstreampos` and `wstreamoff` types if you use `seekg()` with a `wistream` object.

Let's take a look at the arguments to the first prototype of `seekg()`. Values of the `streamoff` type are used to measure offsets, in bytes, from a particular location in a file. The `streamoff` argument represents the file position, in bytes, measured as an offset from one of three locations. (The type may be defined as an integer type or as a class.) The `seek_dir` argument is another integer type that is defined, along with three possible values, in the `ios_base` class. The constant `ios_base::beg` means measure the offset from the beginning of the file. The constant `ios_base::cur` means measure the offset from the current position. The constant `ios_base::end` means measure the offset from the end of the file. Here are some sample calls, assuming that `fin` is an `ifstream` object:

```
fin.seekg(30, ios_base::beg);    // 30 bytes beyond the beginning
fin.seekg(-1, ios_base::cur);    // back up one byte
fin.seekg(0, ios_base::end);     // go to the end of the file
```

Now let's look at the second prototype. Values of the `streampos` type locate a position in a file. It can be a class, but, if so, the class includes a constructor with a `streamoff` argument and a constructor with an integer argument, providing a path to convert both types to `streampos` values. A `streampos` value represents an absolute location in a file, measured

from the beginning of the file. You can treat a `streampos` position as if it measures a file location in bytes from the beginning of a file, with the first byte being byte 0. So the following statement locates the file pointer at byte 112, which would be the 113th byte in the file:

```
fin.seekg(112);
```

If you want to check the current position of a file pointer, you can use the `tellg()` method for input streams and the `tellp()` methods for output streams. Each returns a `streampos` value representing the current position, in bytes, measured from the beginning of the file. When you create an `fstream` object, the input and output pointers move in tandem, so `tellg()` and `tellp()` return the same value. But if you use an `istream` object to manage the input stream and an `ostream` object to manage the output stream to the same file, the input and output pointers move independently of one another, and `tellg()` and `tellp()` can return different values.

You can then use `seekg()` to go to the file beginning. Here is a section of code that opens a file, goes to the beginning, and displays the file contents:

```
fstream finout;     // read and write streams
finout.open(file,ios::in | ios::out |ios::binary);
//NOTE: Some Unix systems require omitting | ios::binary
int ct = 0;
if (finout.is_open())
{
    finout.seekg(0);    // go to beginning
    cout << "Here are the current contents of the "
          << file << " file:\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
        cout << ct++ << ": " << setw(LIM) << pl.name << ": "
        << setprecision(0) << setw(12) << pl.population
        << setprecision(2) << setw(6) << pl.g << endl;
    }
    if (finout.eof())
        finout.clear(); // clear eof flag
    else
    {
        cerr << "Error in reading " << file << ".\n";
        exit(EXIT_FAILURE);
    }
}
else
{
    cerr << file << " could not be opened -- bye.\n";
    exit(EXIT_FAILURE);
}
```

This is similar to the start of Listing 17.19, but there are some changes and additions. First, as just described, the program uses an `fstream` object with a read/write mode, and it uses `seekg()` to position the file pointer at the start of the file. (This isn't really needed for this example, but it shows how `seekg()` is used.) Next, the program makes the minor change of numbering the records as they are displayed. Then it makes the following important addition:

```
if (finout.eof())
    finout.clear(); // clear eof flag
else
{
    cerr << "Error in reading " << file << ".\n";
    exit(EXIT_FAILURE);
}
```

The problem this code addresses is that when the program finishes reading and displaying the entire file, it sets the `eofbit` element. This convinces the program that it's finished with the file and disables any further reading of or writing to the file. Using the `clear()` method resets the stream state, turning off `eofbit`. Now the program can once again access the file. The `else` part handles the possibility that the program quits reading the file for some reason other than reaching the end-of-file, such as a hardware failure.

The next step is to identify the record to be changed and then change it. To do this, the program asks the user to enter a record number. Multiplying the number by the number of bytes in a record yields the byte number for the beginning of the record. If `record` is the record number, the desired byte number is `record * sizeof pl`:

```
cout << "Enter the record number you wish to change: ";
long rec;
cin >> rec;
eatline();              // get rid of newline
if (rec < 0 || rec >= ct)
{
    cerr << "Invalid record number -- bye\n";
    exit(EXIT_FAILURE);
}
streampos place = rec * sizeof pl;  // convert to streampos type
finout.seekg(place);    // random access
```

The variable `ct` represents the number of records; the program exits if you try to go beyond the limits of the file.

Next, the program displays the current record:

```
finout.read((char *) &pl, sizeof pl);
cout << "Your selection:\n";
cout << rec << ": " << setw(LIM) << pl.name << ": "
<< setprecision(0) << setw(12) << pl.population
<< setprecision(2) << setw(6) << pl.g << endl;
if (finout.eof())
    finout.clear();     // clear eof flag
```

After displaying the record, the program lets you change the record:

```
cout << "Enter planet name: ";
cin.get(pl.name, LIM);
eatline();
cout << "Enter planetary population: ";
cin >> pl.population;
cout << "Enter planet's acceleration of gravity: ";
cin >> pl.g;
finout.seekp(place);    // go back
finout.write((char *) &pl, sizeof pl) << flush;

if (finout.fail())
{
    cerr << "Error on attempted write\n";
    exit(EXIT_FAILURE);
}
```

The program flushes the output to guarantee that the file is updated before proceeding to the next stage.

Finally, to display the revised file, the program uses `seekg()` to reset the file pointer to the beginning. Listing 17.20 shows the complete program. Don't forget that it assumes that a `planets.dat` file created using the `binary.cpp` program is available.

> **Note**
>
> The older the implementation, the more likely it is to run afoul of the C++ Standard. Some systems don't recognize the binary flag, the `fixed` and `right` manipulators, and `ios_base`.

Listing 17.20 **random.cpp**

```
// random.cpp -- random access to a binary file
#include <iostream>     // not required by most systems
#include <fstream>
#include <iomanip>
#include <cstdlib>      // for exit()
const int LIM = 20;
struct planet
{
    char name[LIM];     // name of planet
    double population;  // its population
    double g;           // its acceleration of gravity
};

const char * file = "planets.dat";  // ASSUMED TO EXIST (binary.cpp example)
inline void eatline() { while (std::cin.get() != '\n') continue; }

int main()
{
```

```
    using namespace std;
    planet pl;
    cout << fixed;

// show initial contents
    fstream finout;     // read and write streams
    finout.open(file,
           ios_base::in | ios_base::out | ios_base::binary);
    //NOTE: Some Unix systems require omitting | ios::binary
    int ct = 0;
    if (finout.is_open())
    {
        finout.seekg(0);    // go to beginning
        cout << "Here are the current contents of the "
             << file << " file:\n";
        while (finout.read((char *) &pl, sizeof pl))
        {
            cout << ct++ << ": " << setw(LIM) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << endl;
        }
        if (finout.eof())
            finout.clear(); // clear eof flag
        else
        {
            cerr << "Error in reading " << file << ".\n";
            exit(EXIT_FAILURE);
        }
    }
    else
    {
        cerr << file << " could not be opened -- bye.\n";
        exit(EXIT_FAILURE);
    }

// change a record
    cout << "Enter the record number you wish to change: ";
    long rec;
    cin >> rec;
    eatline();              // get rid of newline
    if (rec < 0 || rec >= ct)
    {
        cerr << "Invalid record number -- bye\n";
        exit(EXIT_FAILURE);
    }
    streampos place = rec * sizeof pl;  // convert to streampos type
    finout.seekg(place);    // random access
```

```
    if (finout.fail())
    {
        cerr << "Error on attempted seek\n";
        exit(EXIT_FAILURE);
    }

    finout.read((char *) &pl, sizeof pl);
    cout << "Your selection:\n";
    cout << rec << ": " << setw(LIM) << pl.name << ": "
         << setprecision(0) << setw(12) << pl.population
         << setprecision(2) << setw(6) << pl.g << endl;
    if (finout.eof())
        finout.clear();     // clear eof flag

    cout << "Enter planet name: ";
    cin.get(pl.name, LIM);
    eatline();
    cout << "Enter planetary population: ";
    cin >> pl.population;
    cout << "Enter planet's acceleration of gravity: ";
    cin >> pl.g;
    finout.seekp(place);    // go back
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
        cerr << "Error on attempted write\n";
        exit(EXIT_FAILURE);
    }

// show revised file
    ct = 0;
    finout.seekg(0);            // go to beginning of file
    cout << "Here are the new contents of the " << file
         << " file:\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
        cout << ct++ << ": " << setw(LIM) << pl.name << ": "
             << setprecision(0) << setw(12) << pl.population
             << setprecision(2) << setw(6) << pl.g << endl;
    }
    finout.close();
    cout << "Done.\n";
    return 0;
}
```

Here's a sample run of the program in Listing 17.20, based on a `planets.dat` file that has had a few more entries added since you last saw it:

```
Here are the current contents of the planets.dat file:
0:                Earth:   6928198253  9.81
1:        Jenny's World:     32155648  8.93
2:              Tramtor:  89000000000 15.03
3:              Trellan:      5214000  9.62
4:            Freestone:   3945851000  8.68
5:            Taanagoot:    361000004 10.23
6:                Marin:       252409  9.79
Enter the record number you wish to change: 2
Your selection:
2:              Tramtor:  89000000000 15.03
Enter planet name: Trantor
Enter planetary population: 89521844777
Enter planet's acceleration of gravity: 10.53
Here are the new contents of the planets.dat file:
0:                Earth:   6928198253  9.81
1:        Jenny's World:     32155648  8.93
2:              Trantor:  89521844777 10.53
3:              Trellan:      5214000  9.62
4:            Freestone:   3945851000  8.68
5:            Taanagoot:    361000004 10.23
6:                Marin:       252409  9.79
Done.
```

By using the techniques in this program, you can extend it so that it allows you to add new material and delete records. If you were to expand the program, it would be a good idea to reorganize it by using classes and functions. For example, you could convert the `planet` structure to a class definition; then you could overload the `<<` insertion operator so that `cout << pl` displays the class data members formatted as in the example. Also the example doesn't bother to verify input, so you could add code to check for numeric input where appropriate.

Working with Temporary Files

Developing applications often requires the use of temporary files whose lifetimes are transient and must be controlled by the program. Have you ever thought about how to go about this in C++? It's really quite easy to create a temporary file, copy the contents of another file, and delete the file. First of all, you need to come up with a naming scheme for your temporary file(s). But wait...how can you ensure that each file is assigned a unique name? The `tmpnam()` standard function declared in `cstdio` has you covered:

```
char* tmpnam( char* pszName );
```

The `tmpnam()` function creates a temporary name and places it in the C-style string that is pointed to by `pszName`. The constants `L_tmpnam` and `TMP_MAX`, both defined in `cstdio`,

limit the number of characters in the filename and the maximum number of times `tmpnam()` can be called without generating a duplicate filename in the current directory. The following example generates 10 temporary names:

```
#include <cstdio>
#include <iostream>

int main()
{
    using namespace std;
    cout << "This system can generate up to " << TMP_MAX
         << " temporary names of up to " << L_tmpnam
         << " characters.\n";
    char pszName[ L_tmpnam ] = {'\0'};
    cout << "Here are ten names:\n";
    for( int i=0; 10 > i; i++ )
    {
        tmpnam( pszName );
        cout << pszName << endl;
    }
    return 0;
}
```

More generally, by using `tmpnam()`, you can now generate `TMP_NAM` unique filenames with up to `L_tmpnam` characters per name. The names themselves depend on the compiler. You can run this program to see what names your compiler comes up with.

Incore Formatting

The `iostream` family supports I/O between a program and a terminal. The `fstream` family uses the same interface to provide I/O between a program and a file. The C++ library also provides an `sstream` family, which uses the same interface to provide I/O between a program and a `string` object. That is, you can use the same `ostream` methods you've used with `cout` to write formatted information into a `string` object, and you can use `istream` methods such as `getline()` to read information from a `string` object. The process of reading formatted information from a `string` object or of writing formatted information to a `string` object is termed *incore* formatting. Let's take a brief look at these facilities. (The `sstream` family of `string` support supersedes the `strstream.h` family of `char`-array support.)

The `sstream` header file defines an `ostringstream` class that is derived from the `ostream` class. (There is also a `wostringstream` class based on `wostream`, for wide character sets.) If you create an `ostringstream` object, you can write information to it, which it stores. You can use the same methods with an `ostringstream` object that you can with `cout`. That is, you can do something like the following:

```
ostringstream outstr;
double price = 380.0;
```

```
char * ps = " for a copy of the ISO/EIC C++ standard!";
outstr.precision(2);
outstr << fixed;
outstr << "Pay only CHF " << price << ps << endl;
```

The formatted text goes into a buffer, and the object uses dynamic memory allocation to expand the buffer size as needed. The `ostringstream` class has a member function, called `str()`, that returns a string object initialized to the buffer's contents:

```
string mesg = outstr.str();    // returns string with formatted information
```

Using the `str()` method "freezes" the object, and you can no longer write to it.

Listing 17.21 provides a short example of incore formatting.

Listing 17.21 **strout.cpp**

```
// strout.cpp -- incore formatting (output)
#include <iostream>
#include <sstream>
#include <string>
int main()
{
    using namespace std;
    ostringstream outstr;   // manages a string stream

    string hdisk;
    cout << "What's the name of your hard disk? ";
    getline(cin, hdisk);
    int cap;
    cout << "What's its capacity in GB? ";
    cin >> cap;
    // write formatted information to string stream
    outstr << "The hard disk " << hdisk << " has a capacity of "
           << cap << " gigabytes.\n";
    string result = outstr.str();   // save result
    cout << result;                 // show contents

    return 0;

}
```

Here's a sample run of the program in Listing 17.21:

```
What's the name of your hard disk? Datarapture
What's its capacity in GB? 2000
The hard disk Datarapture has a capacity of 2000 gigabytes.
```

The `istringstream` class lets you use the `istream` family of methods to read data from an `istringstream` object, which can be initialized from a `string` object.

Suppose `facts` is a `string` object. To create an `istringstream` object associated with this string, you can use the following:

```
istringstream instr(facts);     // use facts to initialize stream
```

Then you use `istream` methods to read data from `instr`. For example, if `instr` contained a bunch of integers in character format, you could read them as follows:

```
int n;
int sum = 0;
while (instr >> n)
    sum += n;
```

Listing 17.22 uses the overloaded `>>` operator to read the contents of a string one word at a time.

Listing 17.22 **strin.cpp**

```
// strin.cpp -- formatted reading from a char array
#include <iostream>
#include <sstream>
#include <string>
int main()
{
    using namespace std;
    string lit = "It was a dark and stormy day, and "
                 " the full moon glowed brilliantly. ";
    istringstream instr(lit);   // use buf for input
    string word;
    while (instr >> word)       // read a word a time
        cout << word << endl;
    return 0;

}
```

Here is the output of the program in Listing 17.22:

```
It
was
a
dark
and
stormy
day,
and
the
full
moon
glowed
brilliantly.
```