How to work with file streams

At this point, you have most of the skills you need for working with I/O streams. However, you've only learned how to apply these skills to working with the console. Now, you'll learn how to apply these skills to working with a file. This is known as *file I/O*, and it's important because the data that's stored in *main memory* (*RAM*) is lost when the program ends. However, if you write that data to a file, you can read it from the file the next time the program runs.

How to read and write a file

Figure 5-10 shows how to read and write a file. To do that, you can use the three classes of the fstream header file that are summarized at the top of this figure. To start, you can create an object for the file stream. Then, you can use the open() function to open the stream that connects the file.

For the open() function, the filename argument specifies a path for a file. In this book, the examples only specify the filename. As a result, the open() function looks for the file in the working directory. The location of the working directory depends on the IDE that you're using. For example, if you're using Visual Studio, the working directory is the directory where the source code file that contains the main() function for the program is stored. If you're using Xcode, however, the working directory is the directory where the executable file for the program is stored. If you want to work with a file that isn't in the working directory, you can specify the full path to the file as described in appendix B.

When you open a file stream object, some of the operating system's resources are used to work with that object. As a result, when you're done working with the object, you should call its close() function to close it. This frees the resources that the file stream object is using and prevents problems that can occur if you don't close it.

The examples in this figure work with a *text file*, which is a file that stores data as a series of characters. The first example in this figure shows how to write data to a text file using an ofstream object. To start, it creates an ofstream object named output_file. Then, it opens a file named "names.txt" that's in the current working directory. If this file doesn't already exist, this code creates the file. And if this file does exist, any data it contains is deleted by default. In the next figure, though, you'll learn how to open an output file so you can append data to it.

Next, this code inserts three names into the output file stream where each name is followed by a newline character. Finally, this code closes the ofstream object, which frees all resources being used by this object. This also flushes any remaining data in the buffer.

The second example in this figure shows how to read data from a file. To start, it defines an ifstream object named input_file. Then, it attempts to open a file named "names.txt" that's in the current working directory.

If the file opens successfully, the ifstream object returns a true value. In that case, the code uses a while loop to read each line from the file and display each

Three file stream classes

Class	Description
`ifstream`	Creates an input stream object that reads data from a file on disk.
`ofstream`	Creates an output stream object that writes data to a file on disk.
`fstream`	Can create an input stream, an output stream, or both. See next figure.

Two member functions of the file stream classes

Function	Description
`open(filename)`	Opens a file for reading (input file) or writing (output file). If input file doesn't exist, the operation fails. If output file doesn't exist, it's created.
`close()`	Closes the file and flushes the buffer.

How to include the header file for the file stream classes

```
#include <fstream>
```

How to write data to a file

```
ofstream output_file;            // create output file stream object
output_file.open("names.txt");   // open stream - create file if necessary
output_file << "Grace\n";        // write to stream
output_file << "Ada\n";          // write to stream
output_file << "Alan\n";         // write to stream
output_file.close();             // close stream - flush data to file
```

The contents of the file

```
Grace\nAda\nAlan\n
```

The contents of the file when viewed in a text editor

```
Grace
Ada
Alan
```

How to read data from a file

```
ifstream input_file;             // create input file stream object
input_file.open("names.txt");    // open stream
if (input_file) {                // if stream opened successfully...
    string line;
    while (getline(input_file, line))     // read line from stream
        cout << line << '\n';             // write line to console
    input_file.close();                   // close stream
}
```

The console

```
Grace
Ada
Alan
```

How to define a file stream object and open it in one statement

```
ifstream input_file("names.txt");
```

Figure 5-10 How to read and write a file

line on the console, followed by a newline character. After the while loop, this code closes the ifstream object, which flushes the buffer and frees all resources being used by this object.

If the file doesn't open successfully, this code doesn't do anything. That can happen, for example, if the file doesn't exist. In this case, it isn't necessary to close the ifstream object since the code wasn't able to open that object in the first place.

The last example shows how to create an object for a file stream and open a file in one statement. To do that, you code the name of the file in parentheses after the name of the file stream. This coding technique is more concise than the technique presented in the first two examples, and it's commonly used by C++ programmers.

How to append data to a file

Figure 5-11 shows how to append data to a file. To do that, you can pass a second argument to the open() function of an ofstream object that uses a *file access flag* to specify the mode for the stream.

In the first example, for instance, the second statement calls the open() function of the ofstream object and passes "names.txt" as the first argument and ios:app as the second argument. As a result, C++ opens the file in append mode so the existing data isn't deleted. Then, when the third statement writes data to the file, it appends the data to the end of the file. Since the code in the previous figure already added three names to this file, this example appends a fourth name.

How to use the fstream object to work with files

This figure also shows how to use an fstream object for input and output. When you use this type of object, you need to use the file access flags to specify whether the stream will be used for input or output.

The statements in the second example in figure 5-11 illustrate how this works. Here, the first three open() functions open a file for output. The first open() function causes all the data in the file to be deleted, since that's the default. The second open() function is similar, but it includes the default ios::trunc access flag. Notice that to pass more than one flag to the open() function, you separate the flags with the | operator. The third open() function also uses two flags. This time, though, the second flag indicates that data should be appended to the file.

The last two statements in this figure show how to open an fstream object for input. To do that, you use the ios::in flag. As you saw in the previous figure, you can also define and open an fstream object using a single statement.

If you compare the use of the fstream object with the use of the ifstream and ofstream objects, you'll see that the code for opening ifstream and ofstream objects is somewhat simpler. Because of that, you'll use these objects most of

The syntax for the open() member function of a file stream object

```
object.open(file_name, file_access_flag [ | file_access_flag] ...);
```

Common file access flags

Flag	Description
ios::out	Output mode. If the file exists, any existing data in the file is deleted when the file is opened unless ios::app is also specified. If the file doesn't exist, it's created.
ios::app	Append mode. Any existing data in the file is preserved, and new data is written (appended) to the end of the file. If the file doesn't exist, it's created.
ios::trunc	Truncate mode. Any existing data in the file is deleted (truncated) when the file is opened. This is the default mode that's used by ios::out.
ios::in	Input mode. If the file doesn't exist, the open operation fails.

How to use an ofstream object to append data

```
ofstream output_file;
output_file.open("names.txt", ios::app);  // open in append mode
output_file << "Mary\n";                  // write to stream
output_file.close();                      // close stream and flush
```

The contents of the file when viewed in a text editor

```
Grace
Ada
Alan
Mary
```

How to use a fstream object for input and output

Declare the fstream object

```
fstream file;
```

How to open a file stream that deletes existing data from the output file

```
file.open("names.txt", ios::out);
```

Another way to open a file stream that deletes existing data from the output file

```
file.open("names.txt", ios::out | ios::trunc);
```

How to open a file stream that appends data to the output file

```
file.open("names.txt", ios::out | ios::app);
```

How to open a file stream that reads data from the input file

```
file.open("names.txt", ios::in);
```

How to define a fstream object in one statement

```
fstream file("names.txt", ios::in);
```

Description

- The open() member function of the ofstream and fstream classes accept two arguments. The first is the name of the file to open, and the second is a *file access flag* that indicates the type of stream to create.
- To pass more than one flag to the open() function, separate each flag with the | operator.

Figure 5-11 How to append data to a file and use the fstream object

the time. However, an fstream object provides more flexibility since it allows you to change the input/output mode for a stream. So, you might want to use it for a program that needs to open a file in input mode, read and process input data, switch to output mode, and write the processed data to the file.

How to check for errors when working with files

Figure 5-12 shows how to check for errors when working with files. To start, the first example in this figure shows a while loop that doesn't check for errors as it extracts numbers from the file named info.txt. Here, the condition in the while loop extracts the number. This causes the loop to execute until the extraction operation fails. In this case, the loop ends when the extraction operator attempts to extract "three" from the file. As a result, the code displays 100 and 200 on the console, but not 400, which comes after "three" and "hundred".

To fix this problem, the second example uses the error bits described in figure 5-4 to check the state of the file stream after the extraction operation. Here, the while loop continues until the code executes a break statement. If the extraction was successful, this code displays the number on the console.

If the extraction wasn't successful, this code uses the eof() and fail() functions of the ifstream object to check why the operation failed. If the operation reached the end of the file, this code exits the loop. However, if the operation failed for another reason, this code uses the clear() function to reset the stream to a good state, and it discards the rest of the data in the buffer. This discards all non-numeric data such as "three" and "hundred".

When the extraction operation reaches the end of the file, it sets both the eof and fail bits to true. As a result, if you want your code to work correctly, you must check the eof bit before checking the fail bit as shown in this example. If you check the fail bit first, the else if clause for the eof bit will never be executed, and the code will enter an infinite loop.

When you work with files, you often need to check for errors as you process them. That's especially true of files that you get from outside sources that might have inconsistent data. You'll learn more about this as you progress through this chapter.

The data in the file named info.txt

```
100
200
three hundred
400
```

A while loop that displays the data in the text file

```
ifstream input_file;
input_file.open("info.txt");          // open file for reading
if (input_file) {                     // if file opened successfully...
    int num;
    while (input_file >> num) {       // extract int - while no error...
        cout << num << '|';           // display value
    }
    input_file.close();
}
```

The console

```
100|200|
```

How to handle data conversion errors

```
ifstream input_file;
input_file.open("info.txt");        // open file for reading
if (input_file) {                   // if file opened successfully...
    int num;
    while (true) {                  // loop until there's a break statement
        if (input_file >> num) {        // if extraction is good...
            cout << num << '|';         // display number
        }
        else if (input_file.eof()) {    // if end of file...
            break;                      // exit loop
        }
        else if (input_file.fail()) {   // if extraction failed...
            input_file.clear();         // fix stream and try again
            input_file.ignore(numeric_limits<streamsize>::max(), '\n');
        }
    }
    input_file.close();
}
```

The console

```
100|200|400|
```

Description

- You can use the error bits described in figure 5-4 to check the state of the file stream after operations that might fail, such as extracting data.
- You can use the eof() member function to check whether the operation has reached the end of the file.
- You can use the fail() member function to check whether the operation failed.

Figure 5-12 How to check for errors when working with files

How to read and write delimited data

Figure 5-13 shows how to work with a *table* that stores its data in *columns* and *rows*, which are also referred to as *fields* and *records*. When you store a table of data in a file, it's common to separate the columns and rows of the table with characters known as *delimiters*. There are many ways to do this, but one common way is to use a tab character (\t) to separate columns and a newline character (\n) to separate rows. This type of file is known as a *tab-delimited file*.

The first example in this figure defines a string variable named filename that contains a value of "temps.txt". This variable is used by the second and third examples to write and read tab-delimited data.

The second example begins by defining and opening an ofstream object for the file specified by the filename variable. Then, it sets the output stream so it uses fixed-point notation with 1 decimal place.

After setting up the ofstream object, this code writes two columns and three rows of data to the file. Here, the code separates each row with a newline character, and it separates each column with a tab character. Finally, this code closes the ofstream object, which flushes all data to the file and frees all resources being used by the stream.

The third example begins by writing "TEMPERATURES" to the console, followed by column headings of "Low" and "High". Then, this code defines and opens an ifstream object for the file specified by the filename variable.

After opening the file, this code defines two variables to store the low and high temperatures. Then, it checks if the ifstream object was opened successfully. If so, it sets the input stream so it uses fixed-point notation with 1 decimal place.

Next, this code uses a while loop to extract the low and high temperatures from the file into the low and high variables. Within the loop, the code displays the low and high temperatures in two columns on the console. Like the two heading columns, these columns are 8 characters wide and right justified.

After the while loop, this code closes the ifstream object. This flushes the buffer and frees all resources being used by this stream.

As you review this code, note that it will only work if the data in the file always alternates between the low and high temperatures. In some cases, that may be a valid assumption. In other cases, you'll need to write code that checks for errors when reading the data. To do that, you can sometimes use a string stream as described later in this chapter.

Code that defines a filename

```
string filename = "temps.txt";
```

Code that writes tab-delimited data to a file

```
ofstream output_file;
output_file.open(filename);
output_file << fixed << setprecision(1);
output_file << 48.4 << '\t' << 57.2 << '\n';
output_file << 46.0 << '\t' << 50.0 << '\n';
output_file << 78.3 << '\t' << 101.4 << '\n';
output_file.close();
```

The contents of the file when viewed in a text editor

```
48.4    57.2
46.0    50.0
78.3    101.4
```

Code that reads tab-delimited data from a file

```
cout << "TEMPERATURES\n";
cout << setw(8) << "Low" << setw(8) << "High" << endl;

ifstream input_file;
input_file.open(filename);

double low;
double high;
if (input_file) {                // if file opened successfully...
    cout << fixed << setprecision(1);
    while (input_file >> low >> high) {
        cout << setw(8) << low << setw(8) << high << '\n';
    }
    input_file.close();
}
```

The console

```
TEMPERATURES
     Low    High
    48.4    57.2
    46.0    50.0
    78.3   101.4
```

Description

- When storing a *table* of data in a file, it's common to separate the *columns* and *rows* of the table with characters known as *delimiters*.
- A file that uses tab (\t) and newline (\n) characters to separate columns and rows is known as a *tab-delimited file*.

Figure 5-13 How to read and write delimited data

The Temperature Manager program

Figure 5-14 shows a Temperature Manager program that allows you to view pairs of high and low temperatures that are stored in a file. In addition, it allows you to add temperatures to the file. And when you're done, it allows you to exit the program. To perform these tasks, you can enter a command of 'v', 'a', or 'x' respectively.

The code for this program begins by including all of the necessary header files. This includes the iomanip, fstream, and limits files described in this chapter.

Within the main() function, the first statement defines a string variable named filename and initializes it to a value of "temps.txt". Later in this function, the file streams for input and output both use this filename. As a result, you can be sure that operations for file input and output are both working with the same file.

After displaying the list of commands, this program enters a while loop that continues until the user enters 'x' to exit. Within this loop, the first four statements display the Command prompt and get the char that corresponds with the command entered by the user.

The console

```
The Temperature Manager program

COMMANDS
v - View temperatures
a - Add a temperature
x - Exit

Command: v

TEMPERATURES
     Low    High
-------- -------
    48.4    57.2
    46.0    50.0
    78.3   101.4

Command: a

Enter low temp: 54
Enter high temp: 61
Temps have been saved.

Command: x

Bye!
```

The code

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <limits>
#include <string>

using namespace std;

int main()
{
    string filename = "temps.txt";

    cout << "The Temperature Manager program\n\n";

    cout << "COMMANDS\n"
         << "v - View temperatures\n"
         << "a - Add a temperature\n"
         << "x - Exit\n";

    char command = 'v';
    while (command != 'x') {
        // get command from user
        cout << endl;
        cout << "Command: ";
        cin >> command;
        cout << endl;
```

Figure 5-14 The Temperature Manager program (part 1 of 2)

After getting the command character, this code checks whether that character is equal to 'v'. If it is, the code continues by reading all the low and high temperatures stored in the file and displaying them on the console. But first, it attempts to open the file and then checks that it was opened successfully. Since this code is similar to code you saw in figure 5-13, you shouldn't have much trouble understanding how it works. After executing this code, program execution reaches the end of the while loop and jumps back to the top of the loop, which causes the program to get another command from the user.

If the command character is equal to 'a', this code gets a low and a high temperature from the user. Then, it defines and opens an ofstream object and uses it to append the low and high temperatures to the end of the file. Since this file is a tab-delimited file, this code separates the two temperatures with a tab and follows them with a new line. Again, after this code is executed, program execution reaches the end of the while loop and jumps back to the top of the loop, which causes the program to get another command from the user.

If the command character is equal to 'x', the code displays a "Bye!" message on the console to indicate that the program is ending. Then, program execution jumps back to the top of the while loop, and the condition for the loop evaluates to false. This causes the loop to end, which ends the main() function and the program.

If the command character isn't one of the characters in the previous if or else if clauses, the else clause displays a message on the console. This message indicates that the command is invalid and that the user should try again. Because this is the last statement in the while loop, code execution jumps to the top of the loop, and the code prompts the user to enter another command.

As you review this code, note that the code that adds the new low and high temperatures to the file needs to use the ios:app file input flag to open the file in append mode. Otherwise, all existing data in the file would be deleted when the file was opened.

Also, note that this code doesn't validate the temperature data that's entered by the user. This helps keep the focus on working with file streams. In general, though, validating this data would be a good practice. In addition, it's a good practice to code functions for data validation and to store those functions in a separate file. You'll learn how to do that in chapter 7.

The code (continued)

```
            if (command == 'v') {
                cout << "TEMPERATURES\n";
                cout << setw(8) << "Low" << setw(8) << "High" << endl;
                cout << "-------- -------" << endl;

                ifstream input_file;
                input_file.open(filename);

                double low;
                double high;
                if (input_file) {     // if file opened successfully...
                    cout << fixed << setprecision(1);
                    while (input_file >> low >> high) {
                        cout << setw(8) << low << setw(8) << high << '\n';
                    }
                    input_file.close();
                }
                else {
                    cout << "Unable to open file.\n";
                }
            }
            else if (command == 'a') {
                double low;
                cout << "Enter low temp: ";
                cin >> low;

                double high;
                cout << "Enter high temp: ";
                cin >> high;

                ofstream output_file;
                output_file.open(filename, ios::app);
                output_file << low << '\t' << high << '\n';
                output_file.close();
                cout << "Temps have been saved.\n";
            }
            else if (command == 'x') {
                cout << "Bye!\n\n";
            }
            else {
                cout << "Invalid command. Try again.\n";
            }
        }
    }
```

Figure 5-14 The Temperature Manager program (part 2 of 2)