CHAPTER 7

# File Handling and Applications

After studying this chapter, you will be able to:

- Understand computer files and perform file operations
- Work with sequential files and control break logic

In this chapter, you learn how to open and close files in C++, how to use C++ to read data from and write data to a file in a program, and how to work with sequential files in a C++ program. You should do the exercises and labs in this chapter after you have finished Chapter 7 in *Programming Logic and Design, Eighth Edition*.



# File Handling

Business applications are often required to manipulate large amounts of data that are stored in one or more files. As you learned in Chapter 7 of *Programming Logic and Design*, data is organized in a hierarchy. At the lowest level of the hierarchy is a **field**, which is a group of characters. On the next level up is a **record**, which is a group of related fields. For example, you could write a program that processes employee records with each employee record consisting of three fields: the employee's first name, the employee's last name, and the employee's department number.

In C++, to use the data stored in a file, the program must first open the file and then read the data from the file. You use prewritten classes that are part of a C++ library to accomplish this. In the next section, you learn how to use these classes to open a file, close a file, read data from a file, and write data to a file.

## Using Input and Output Classes

To use the classes that you need to perform file input (`ifstream`) and output (`ofstream`), you must add the following preprocessor directive at the beginning of your C++ program:

```
#include <fstream>
```

The classes included in the `fstream` class are prewritten for you by the C++ development team and allow you to simplify your programming tasks by creating objects using these classes. You can then use the attributes and methods of these objects in your C++ programs.

## Opening a File for Reading

To open a file and read data into a C++ program, you declare a file input stream object (variable) as shown below:

```
ifstream data_in;
```

You can think of `ifstream` as the data type for input files. In Chapter 10, you will learn that `ifstream` is actually a class and `data_in` is an object.

You then use the `open` method (function) to specify the name of the file to open. Look at the following example:

```
data_in.open("inputFile.dat");
```

In the example, you use the `ifstream` object (`data_in`), followed by a dot (`.`), followed by the name of the method (`open`) to open a file for input. Within the parentheses, you place the name of the file you want to open within a pair of double quotes. This statement opens the file named `inputFile.dat` for reading. This means that the program can now read data from the file. In this example, the file named `inputFile.dat` must be saved in the same folder as the C++ program that is using the file. To open a file that is saved in a different folder, a path must be specified as in this example:

```
data_in.open("C:\myC++Programs\Chapter7\inputFile.dat");
```

## Reading Data from an Input File

Once you have opened the input file, you are ready to read the data in the file. You can use the extraction operator >> just as you used it with `cin`.

Assume that the input file for a program is organized so an employee's first name is on one line, followed by his last name on the next line, followed by his salary on the third, as follows:

Tim

Moriarty

4000.00

To allow the program to read this data, you would write the following C++ code:

```
ifstream data_in;
string firstName, lastName;
double salary;
data_in.open("inputFile.dat");
data_in >> firstName;
data_in >> lastName;
data_in >> salary;
```

The first line in the example declares `data_in`, an `ifstream` object. The second line declares two `string` variables named `firstName` and `lastName`. The third line declares a `double` named `salary`. Next, `data_in` is used with the `open` method to open the file named `inputFile.dat`. The extraction operator >> is then used three times to read the three lines of input from the file (`inputFile.dat`) associated with `data_in`. After this code executes, the variable named `firstName` contains the value *Tim*, the variable named `lastName` contains the value *Moriarty*, and the variable named `salary` contains the value *4000.00*.

## Reading Data Using a Loop and EOF

In a program that has to read large amounts of data, it is usually best to have the program use a loop. In the loop, the program continues to read from the file until EOF (end of file) is encountered. In C++, the `eof` method (function) returns a `true` value when EOF is reached and a `false` value when EOF has not been reached. The `eof` method can be used with `cin` for keyboard input or with an `ifstream` object, such as `data_in`.

The C++ code that follows shows how to use the `eof` method as part of a loop:

```
ifstream data_in;
string firstName;
data_in.open("inputFile.dat");
data_in >> firstName;
while(!(data_in.eof()))
{
   cout << firstName;
   data_in >> firstName;
}
```

For keyboard input, you enter the EOF character by typing Ctrl+Z (Windows) or Ctrl+D (UNIX/Linux).

In this example, a priming read is used on the fourth line, `data_in >> firstName;`, and then the `eof` method is used as the expression to be tested in the `while` loop. As long as the value returned by `eof` is not equal to `true`, the expression is `true` and the loop is entered. Notice the negation operator `!` is used to reverse the value returned by the `eof` method. As soon as EOF is encountered, the test becomes `false` and the program exits the loop. The parentheses that surround the `eof` method call are used to control precedence.

## Opening a File for Writing

To write data from a C++ program to an output file, the program must first open a file. Similar to input files, you must first declare a file stream object (variable). For output files, you declare an output stream object as shown below:

```
ofstream data_out;
```

You can think of `ofstream` as the data type for output files. In Chapter 10, you will learn that `ofstream` is actually a class and `data_out` is an object.

You then use the `open` method (function) to specify the name of the file to open. Look at the following example:

```
data_out.open("outputFile.dat");
```

In the example, you use the `ofstream` object (`data_out`), followed by a dot (`.`), followed by the name of the method (`open`) to open a file for output. Within the parentheses, you place the name of the file you want to open within a pair of double quotes. This statement opens the file named `outputFile.dat` for writing. This means that the program can now write data to the file. In this example, the file named `outputFile.dat` is saved in the same folder as the C++ program that is using the file. To open a file that is saved in a different folder, a path must be specified as in this example:

```
data_out.open("C:\myC++Programs\Chapter7\outputFile.dat");
```

## Writing Data to an Output File

Once you have opened the output file, you are ready to write data to the file. You can use the insertion operator << just as you used it with `cout`.

As an example, assume that an employee's `firstName`, `lastName`, and `salary` have been read from an input file as in the previous example, and that the employee is to receive a 15 percent salary increase that is calculated as follows:

```
const double INCREASE = 1.15;
double newSalary;
newSalary = salary * INCREASE;
```

You now want to write the employee's `firstName`, `lastName`, and `newSalary` to the output file named `outputFile.dat`. The code that follows accomplishes this task:

```
ofstream data_out;
data_out.open("outputFile.dat");
data_out << firstName << endl;
data_out << lastName << endl;
data_out << newSalary << endl;
data_out.close();
```

The last line in the previous example uses `data_out` and the `close` method to close the output file. It is a good programming practice to close input and output files when they are no longer needed by a program. The C++ program `fileIOTest.cpp` shown in Figure 7-1 implements the file input and output operations discussed in this section.

```
// fileIOTest.cpp
// Input:  inputFile.dat
// Output: outputFile.dat

#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string firstName;
   string lastName;
   double salary;
   double new_salary;
   const double INCREASE = 1.15;
   ifstream data_in;
   ofstream data_out;

   // Open input and output file
   data_in.open("inputFile.dat");
   data_out.open("outputFile.dat");

   // Read record from input file
   data_in >> firstName;
   data_in >> lastName;
   data_in >> salary;
   // Calculate new salary
   new_salary = salary * INCREASE;
   // Write record to output file
   data_out << firstName << endl;
   data_out << lastName << endl;
   data_out << new_salary << endl;
   // Close files
   data_in.close();
   data_out.close();
   return 0;
} // End of main function
```

**Figure 7-1** Reading and writing file data

When writing code that opens files and writes to files, you need to be aware of two potential problems: first, the program might try to open a nonexistent file; and second, it might try to read beyond the EOF marker. You will learn how to handle these error situations as you learn more about C++. They are beyond the scope of this book.

## Exercise 7-1: Opening Files and Performing File Input

In this exercise, you use what you have learned about opening a file and getting input into a program from a file. Study the following code, and then answer Questions 1–3.

```
1  ofstream data_in;
2  data_in.open(myDVDFile.dat);
3  string dvdName, dvdPrice, dvdShelf;
4  data_in >> dvdShelf;
5  data_in >> dvdPrice;
6  data_in >> dvdName;
```

**Figure 7-2** Code for Exercise 7-1

1. Describe the error on line 1, and explain how to fix it.

   _____

2. Describe the error on line 2, and explain how to fix it.

   _____

3. Consider the following data from the input file `myDVDFile.dat`:

   ```
   Fargo 8.00 1A
   Amadeus 20.00 2C
   Casino 7.50 3B
   ```

   a. What value is stored in the variable named `dvdName`?

   _____

   b. What value is stored in the variable named `dvdPrice`?

   _____

   c. What value is stored in the variable named `dvdShelf`?

   _____

   d. If there is a problem with the values of these variables, what is the problem and how could you fix it?

   _____

## Lab 7-1: Opening Files and Performing File Input

In this lab, you open a file and read input from that file in a prewritten C++ program. The program should read and print the names of flowers and whether they are grown in shade or sun. The data is stored in the input file named `flowers.dat`.

1. Open the source code file named `Flowers.cpp` using Notepad or the text editor of your choice.
2. Declare the variables you will need.

3. Write the C++ statements that will open the input file `flowers.dat` for reading.
4. Write a `while` loop to read the input until EOF is reached.
5. In the body of the loop, print the name of each flower and where it can be grown (sun or shade).
6. Save this source code file in a directory of your choice, and then make that directory your working directory.
7. Compile the source code file `Flowers.cpp`.
8. Execute the program.

## Understanding Sequential Files and Control Break Logic

As you learned in Chapter 7 of *Programming Logic and Design* a **sequential file** is a file in which records are stored one after another in some order. The records in a sequential file are organized based on the contents of one or more fields, such as ID numbers, part numbers, or last names.

A **single-level control break** program reads data from a sequential file and causes a break in the logic based on the value of a single variable. In Chapter 7 of *Programming Logic and Design*, you learned about techniques you can employ to implement a single-level control break program. Be sure you understand these techniques before you continue on with this chapter. The program described in Chapter 7 of *Programming Logic and Design* that produces a report of customers by state is an example of a single-level control break program. This program reads a record for each client, keeps a count of the number of clients in each state, and prints a report. As shown in Figure 7-3, the report generated by this program includes clients' names, cities, and states, along with a count of the number of clients in each state.

Company Clients by State of Residence

| Name | City | State | |
|---|---|---|---|
| Albertson | Birmingham | Alabama | |
| Davis | Birmingham | Alabama | |
| Lawrence | Montgomery | Alabama | |
| | | Count for Alabama | 3 |
| Smith | Anchorage | Alaska | |
| Young | Anchorage | Alaska | |
| Davis | Fairbanks | Alaska | |
| Mitchell | Juneau | Alaska | |
| Zimmer | Juneau | Alaska | |
| | | Count for Alaska | 5 |
| Edwards | Phoenix | Arizona | |
| | | Count for Arizona | 1 |

**Figure 7-3** Control break report with totals after each state

Each client record is made up of the following fields: `Name`, `City`, and `State`. Note the following example records, each made up of three lines:

```
Albertson
Birmingham
Alabama
Lawrence
Montgomery
Alabama
Smith
Anchorage
Alaska
```

Remember that input records for a control break program are usually stored in a data file on a storage device, such as a disk, and the records are sorted according to a predetermined control break variable. For example, the control break variable for this program is `state`, so the input records would be sorted according to `state`.

Figure 7-4 includes the pseudocode for the Client By State program, and Figure 7-5 includes the C++ code that implements the program.

```
Start
   Declarations
      InputFile inFile
      string TITLE = "Company Clients by State of Residence"
      string COL_HEADS = "Name   City   State"
      string name
      string city
      string state
      num count = 0
      String oldState
   getReady()
   while not eof
      produceReport()
   endwhile
   finishUp()
stop

getReady()
   output TITLE
   output COL_HEADS
   open inFile "ClientsByState.dat"
   input name, city, state from inFile
   oldState = state
return
```

**Figure 7-4** Client By State program pseudocode *(continues)*

*(continued)*

```
produceReport()
   if state <> oldState then
      controlBreak()
   endif
   output name, city, state
   count = count + 1
   input name, city, state from inFile
return

controlBreak()
   output "Count for ", oldState, count
   count = 0
   oldState = state
return

finishUp()
   output "Count for ", oldState, count
   close inFile
return
```



**Figure 7-4** Client By State program pseudocode

```
 1  // ClientByState.cpp - This program creates a report that lists
 2  // clients with a count of the number of clients for each state.
 3  // Input:  client.dat
 4  // Output:  Report
 5
 6  #include <fstream>
 7  #include <iostream>
 8  #include <string>
 9  using namespace std;
10
11  int main()
12  {
13     // Declarations
14     ifstream fin;
15     const string TITLE = "\n\nCompany Clients by State of Residence\n\n";
16     string name = "", city = "", state = "";
17     int count = 0;
18     string oldState = "";
19     bool done;
20
```

**Figure 7-5** Client By State program written in C++ *(continues)*

*(continued)*

```
21     fin.open("client.dat");
22     // This is the work done in the getReady() function
23     cout << TITLE << endl;
24     fin >> name;
25     if(!(fin.eof()))
26     {
27        fin >> city;
28        fin >> state;
29        done = false;
30        oldState = state;
31     }
32     else
33        done = true;
34     while(done == false)
35     {
36      // This is the work done in the produceReport() function
37      if(state != oldState)
38      {
39           // This is the work done in the controlBreak() function
40          cout << "\t\t\tCount for " << oldState << " " << count << endl;
41          count = 0;
42          oldState = state;
43      }
44      cout << name << " " << city << " " << state << endl;
45      count++;
46      fin >> name;
47      if(!(fin.eof()))
48      {
49        fin>> city;
50        fin >> state;
51        done = false;
52      }
53      else
54        done = true;
55     }
56     // This is the work done in the finishUp() function
57     cout << "\t\t\tCount for " << oldState << " " << count << endl;
58     fin.close();
59
60      return 0;
61  }  // End of main() class.
```

**Figure 7-5** Client By State program written in C++

As you can see in Figure 7-5, the C++ program begins on line 1 with comments that describe what the program does. (The line numbers shown in this program are not part of the C++ code. They are included for reference only.) The program also includes comments that describe the program's input and output. Next comes the C++ code that defines the `main()` function (line 11).

Within the `main()` function, lines 14 through 19 declare variables and constants and initialize them when appropriate. Line 21 opens the input file named `client.dat`. Lines 23 through 33 include the work done in the `getReady()` function, which includes printing the heading for the report this program generates and performing a priming read. You learned about performing a priming read in Chapter 3 of this book and in Chapter 3 of *Programming Logic and Design*.

Notice that the C++ code in the priming read (lines 24 through 28) is a little different than the pseudocode. An `if` statement is used on line 25 to test if a client's name was read from the input file or if EOF was encountered. If EOF is not encountered, the result of this test will be `true`, causing the execution of the input statements that read the city and state from the input file. The `bool` value `false` is also assigned to the variable named `done` on line 29, followed by assigning the current value of `state` to the variable named `oldState` on line 30. Remember that the variable `state` serves as the control break variable. If EOF is encountered, the result of this test will be `false`, causing the `bool` value `true` to be assigned to the variable named `done` on line 33. The `bool` variable named `done` is used later in the program to control the `while` loop.

Next comes the `while` loop (line 34), which continues to execute as long as the value of the `bool` variable `done` is `false`. The body of the `while` loop contains the work done in the `produceReport()` function. First, on line 37, an `if` statement tests the control break variable, `state`, to determine if the record the program is currently working with has the same state as the previous record's state. If it does not, this indicates the beginning of a new state. As a result, the program performs the work done in the `controlBreak()` function (lines 40 through 42). The work of the `controlBreak()` function does the following:

1. Prints the value of the variable named `count` that contains the count of clients in the current state (line 40)
2. Assigns the value 0 to the variable named `count` to prepare for the next state
3. Assigns the value of the variable named `state` to the variable named `oldState` to prepare for the next state

If the record the program is currently working with has the same state as the previous record's state, the `controlBreak()` function's work is not performed. Whether or not the current record's state is the same state as the previous record's state, the next statement to execute (line 44) prints the client's name, city, and state. Then the variable named `count` is incremented on line 45 followed by the program reading the next client's record on lines 46 through 51 using the same technique as the priming read.

The condition in the `while` loop on line 34 is then tested again, causing the loop to continue executing until the value of the variable named `done` is `true`. The variable named `done` is assigned the value `true` when the program encounters EOF when reading from the input file on line 54.

When the `while` loop is exited, the last section of the program executes. This consists of the work done in the `finishUp()` function and comprises:

- Printing the value of the variable named `count` (which is the count of the clients in the last state in the input file) on line 57
- Closing the input file (line 58)



## Exercise 7-2: Accumulating Totals in Single-Level Control Break Programs

In this exercise, you will use what you have learned about accumulating totals in a single-level control break program. Study the following code, and then answer Questions 1–4.

```
if(sectionNum != oldSectionNum)
{
      System.out.println("Section Number " + oldSectionNum);
      totalSections = 1;
      oldSectionNum = sectionNum;
}
```

1. What is the control break variable?

   _____

2. The value of the control break variable should never be changed. True or false?

   _____

3. Is `totalSections` being calculated correctly?

   _____

   If not, how can you fix the code?

   _____

4. In a control break program, it doesn't matter if the records in the input file are in a specified order. True or false?

   _____

## Lab 7-2: Accumulating Totals in Single-Level Control Break Programs

In this lab, you will use what you have learned about accumulating totals in a single-level control break program to complete a C++ program. The program should produce a report for a supermarket manager to help her keep track of hours worked by her part-time employees. The report should include the day of the week and the number of hours worked for each employee for each day of the week and the total hours for the day of the week. The report should look similar to the one shown in Figure 7-6.

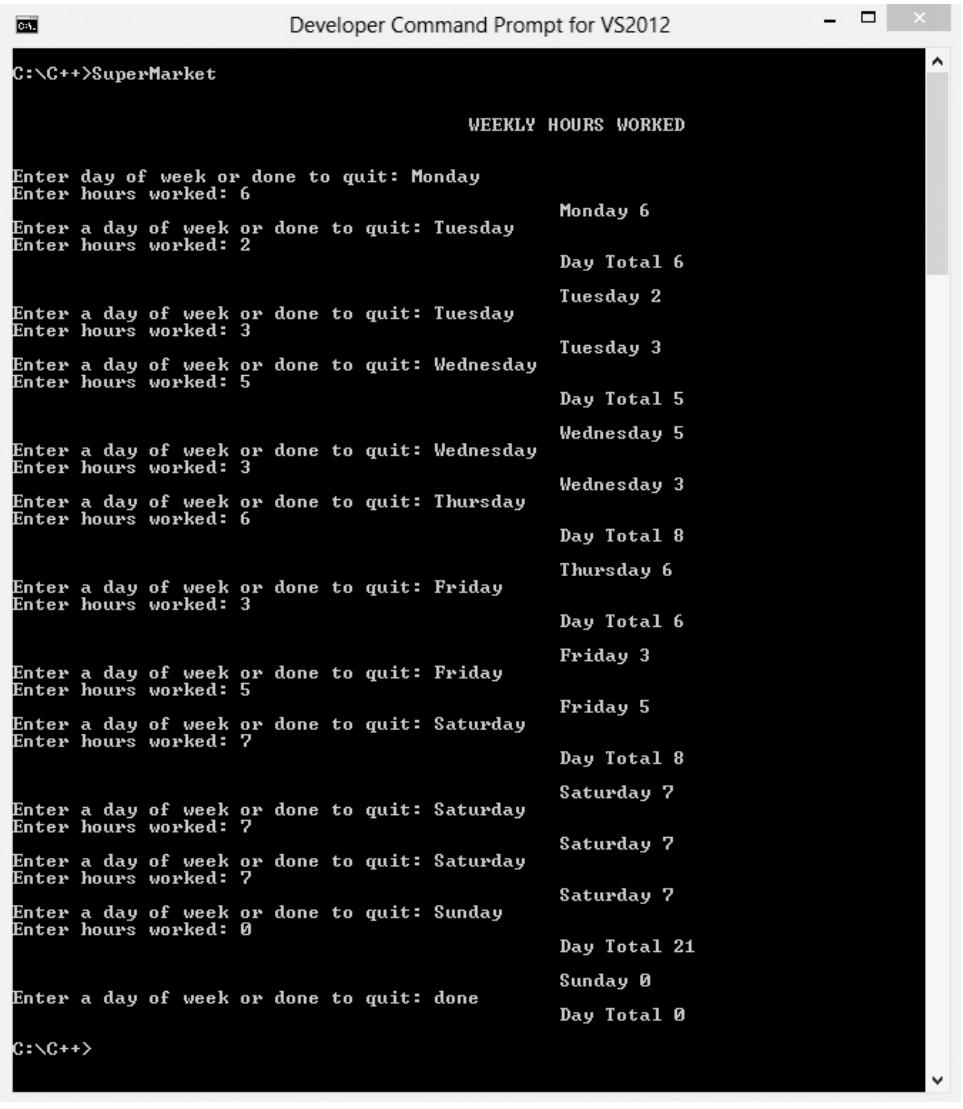


**Figure 7-6** Super Market program report

The student file provided for this lab includes the necessary variable declarations and input and output statements. You need to implement the code that recognizes when a control break should occur. You also need to complete the control break code. Be sure to accumulate the daily totals for all days in the week. Comments in the code tell you where to write your code. You can use the Client By State program in this chapter as a guide for this new program.

1. Open the source code file named `SuperMarket.cpp` using Notepad or the text editor of your choice.
2. Study the prewritten code to understand what has already been done.

3. Write the control break code, including the code for the `dayChange()` function, in the `main()` function.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file, `SuperMarket.cpp`.
6. Execute the program using the following input values:

   **`Monday`**—**`6`** hours (employee 1)

   **`Tuesday`**—**`2`** hours (employee 1), **`3`** hours (employee 2)

   **`Wednesday`**—**`5`** hours (employee 1), **`3`** hours (employee 2)

   **`Thursday`**—**`6`** hours (employee 1)

   **`Friday`**—**`3`** hours (employee 1), **`5`** hours (employee 2)

   **`Saturday`**—**`7`** hours (employee 1), **`7`** hours (employee 2), **`7`** hours (employee 3)

   **`Sunday`**—**`0`** hours