5.11 Using Files for Data Storage

CONCEPT: When a program needs to save data for later use, it writes the data in a file. The data can then be read from the file at a later time.

The programs you have written so far require the user to reenter data each time the program runs, because data kept in variables and control properties is stored in RAM and disappears once the program stops running. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. Data that is stored in a file can be then retrieved and used at a later time.

Most of the commercial software that you use on a day-to-day basis store data in files. The following are a few examples.

- **Word processors:** Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and printed.
- **Image editors:** Image editing programs are used to draw graphics and edit images such as the ones that you take with a digital camera. The images that you create or edit with an image editor are saved in files.
- **Spreadsheets:** Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved in a file for use later.
- **Games:** Many computer games keep data stored in files. For example, some games keep a list of player names with their scores stored in a file. These games typically display the players' names in order of their scores, from highest to lowest. Some games also allow you to save your current game status in a file so you can quit the game and then resume playing it later without having to start from the beginning.
- **Web browsers:** Sometimes when you visit a Web page, the browser stores a small file known as a *cookie* on your computer. Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

Programs that are used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company's products in files, accounting systems keep data about a company's financial operations in files, and so on.

Programmers usually refer to the process of saving data in a file as *writing data* to the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 5-12. An *output file* is a file that data is written to. It is called an output file because the program stores output in it.

The process of retrieving data from a file is known as *reading data* from the file. When a piece of data is read from a file, it is copied from the file into a variable in RAM. Figure 5-13 illustrates this process. An *input file* is a file that data is read from. It is called an input file because the program gets input from the file.

Figure 5-12 Writing data to a file

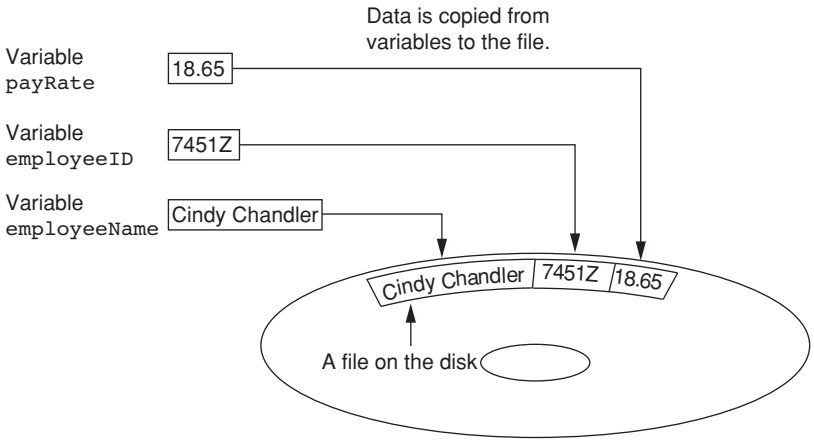
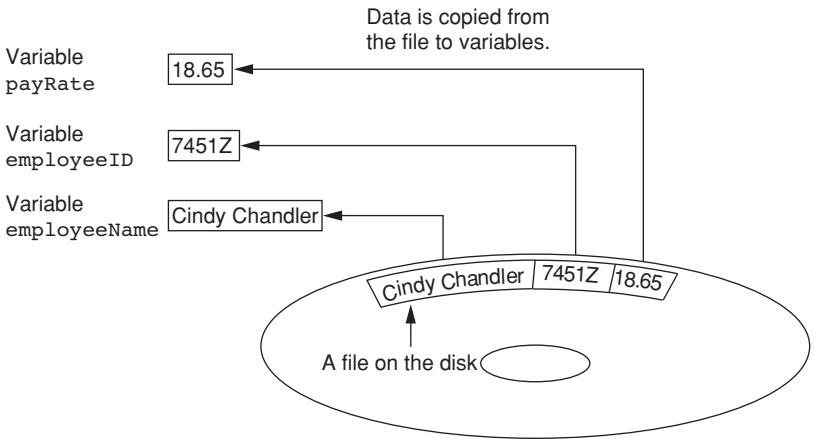


Figure 5-13 Reading data from a file



This section discusses ways to create programs that write data to files and read data from files. When a file is used by a program, three steps must be taken.

1. **Open the file**—Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file.
2. **Process the file**—Data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. **Close the file**—After the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.

Types of Files

In general, there are two types of files: text and binary. A *text file* contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A *binary file* contains data

that has not been converted to text. Thus, you cannot view the contents of a binary file with a text editor. In this chapter we work only with text files. In Chapter 12 you will learn to work with binary files.

File Access Methods

There are two general ways to access data stored in a file: sequential access and direct access. When you work with a *sequential access file*, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all of the data that comes before it—you cannot jump directly to the desired data. This is similar to the way cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

When you work with a *random access file* (also known as a *direct access file*), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song that you want to listen to.

This chapter focuses on sequential access files. Sequential access files are easy to work with, and you can use them to gain an understanding of basic file operations. In Chapter 12 you will learn to work with random access files.

Filenames and File Stream Objects

Files on a disk are identified by a *filename*. For example, when you create a document with a word processor and then save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Figure 5-14 shows how three files named cat.jpg, notes.txt, and resume .doc might be represented in Windows Explorer.

Figure 5-14 Three files

Each operating system has its own rules for naming files. Many systems, including Windows, support the use of *filename extensions*, which are short sequences of characters that appear at the end of a filename preceded by a period (known as a "dot"). For example, the files depicted in Figure 5-14 have the extensions .jpg, .txt, and .doc. The extension usually indicates the type of data stored in the file. For example, the .jpg extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The .txt extension usually indicates that the file contains text. The .doc extension usually indicates that the file contains a Microsoft Word document.

In order for a program to work with a file on the computer's disk, the program must create a file stream object in memory. A *file stream object* is an object that is associated with a specific file and provides a way for the program to work with that file. It is called a "stream" object because a file can be thought of as a stream of data.

File stream objects work very much like the `cin` and `cout` objects. A stream of data may be sent to `cout`, which causes values to be displayed on the screen. A stream of data may be read from the keyboard by `cin`, and stored in variables. Likewise, streams of data may be sent to a file stream object, which writes the data to a file. When data is read from a file, the data flows from the file stream object that is associated with the file, into variables.

Setting Up a Program for File Input/Output

Just as `cin` and `cout` require the `iostream` file to be included in the program, C++ file access requires another header file. The file `fstream` contains all the declarations necessary for file operations. It is included with the following statement:

```
#include <fstream>
```

The `fstream` header file defines the data types `ofstream`, `ifstream`, and `fstream`. Before a C++ program can work with a file, it must define an object of one of these data types. The object will be "linked" with an actual file on the computer's disk, and the operations that may be performed on the file depend on which of these three data types you pick for the file stream object. Table 5-1 lists and describes the file stream data types.

Table 5-1

File Stream Data Type	Description
`ofstream`	Output file stream. You create an object of this data type when you want to create a file and write data to it.
`ifstream`	Input file stream. You create an object of this data type when you want to open an existing file and read data from it.
`fstream`	File stream. Objects of this data type can be used to open files for reading, writing, or both.

NOTE: In this chapter we discuss only the `ofstream` and `ifstream` types. The `fstream` type is covered in Chapter 12.

Creating a File Object and Opening a File

Before data can be written to or read from a file, the following things must happen:

- A file stream object must be created
- The file must be opened and linked to the file stream object.

The following code shows an example of opening a file for input (reading).

```
ifstream inputFile;
inputFile.open("Customers.txt");
```

The first statement defines an `ifstream` object named `inputFile`. The second statement calls the object's `open` member function, passing the string `"Customers.txt"` as an argument. In this statement, the `open` member function opens the Customers.txt file and links it with the `inputFile` object. After this code executes, you will be able to use the `inputFile` object to read data from the Customers.txt file.

The following code shows an example of opening a file for output (writing).

```
ofstream outputFile;
outputFile.open("Employees.txt");
```

The first statement defines an `ofstream` object named `outputFile`. The second statement calls the object's `open` member function, passing the string `"Employees.txt"` as an argument. In this statement, the `open` member function creates the Employees.txt file and links it with the `outputFile` object. After this code executes, you will be able to use the `outputFile` object to write data to the Employees.txt file. It's important to remember that when you call an `ofstream` object's `open` member function, the specified file will be created. If the specified file already exists, it will be erased, and a new file with the same name will be created.

Often, when opening a file, you will need to specify its path as well as its name. For example, on a Windows system the following statement opens the file `C:\data\inventory.txt`:

```
inputFile.open("C:\\data\\inventory.txt")
```

In this statement, the file `C:\data\inventory.txt` is opened and linked with `inputFile`.

NOTE: Notice the use of two backslashes in the file's path. Two backslashes are needed to represent one backslash in a string literal.

It is possible to define a file stream object and open a file in one statement. Here is an example:

```
ifstream inputFile("Customers.txt");
```

This statement defines an `ifstream` object named `inputFile` and opens the Customer.txt file. Here is an example that defines an `ofstream` object named `outputFile` and opens the Employees.txt file:

```
ofstream outputFile("Employees.txt");
```

Closing a File

The opposite of opening a file is closing it. Although a program's files are automatically closed when the program shuts down, it is a good programming practice to write statements that close them. Here are two reasons a program should close files when it is finished using them:

- Most operating systems temporarily store data in a *file buffer* before it is written to a file. A file buffer is a small "holding section" of memory that file-bound data is first written to. When the buffer is filled, all the data stored there is written to the file. This technique improves the system's performance. Closing a file causes any unsaved data that may still be held in a buffer to be saved to its file. This means the data will be in the file if you need to read it later in the same program.
- Some operating systems limit the number of files that may be open at one time. When a program closes files that are no longer being used, it will not deplete more of the operating system's resources than necessary.

Calling the file stream object's `close` member function closes a file. Here is an example:

```
inputFile.close();
```

Writing Data to a File

You already know how to use the stream insertion operator (<<) with the `cout` object to write data to the screen. It can also be used with `ofstream` objects to write data to a file. Assuming `outputFile` is an `ofstream` object, the following statement demonstrates using the << operator to write a string literal to a file:

```
outputFile << "I love C++ programming\n";
```

This statement writes the string literal `"I love C++ programming\n"` to the file associated with `outputFile`. As you can see, the statement looks like a `cout` statement, except the name of the `ofstream` object name replaces `cout`. Here is a statement that writes both a string literal and the contents of a variable to a file:

```
outputFile << "Price: " << price << endl;
```

The statement above writes the stream of data to `outputFile` exactly as `cout` would write it to the screen: It writes the string `"Price: "`, followed by the value of the `price` variable, followed by a newline character.

Program 5-15 demonstrates opening a file, writing data to the file, and closing the file. After this code has executed, we can open the demofile.txt file using a text editor and look at its contents. Figure 5-15 shows how the file's contents will appear in Notepad.

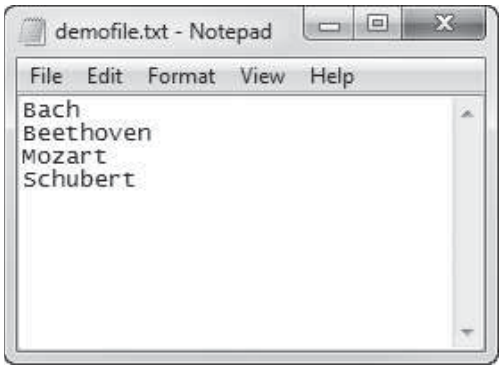
Program 5-15

```
1  // This program writes data to a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       ofstream outputFile;
9       outputFile.open("demofile.txt");
10
11      cout << "Now writing data to the file.\n";
12
13      // Write four names to the file.
14      outputFile << "Bach\n";
15      outputFile << "Beethoven\n";
16      outputFile << "Mozart\n";
17      outputFile << "Schubert\n";
18
19      // Close the file
20      outputFile.close();
21      cout << "Done.\n";
22      return 0;
23 }
```

Program Screen Output

```
Now writing data to the file.
Done.
```

Figure 5-15



Notice that in lines 14 through 17 of Program 5-15, each string that was written to the file ends with a newline escape sequence (`\n`). The newline specifies the end of a line of text. Because a newline is written at the end of each string, the strings appear on separate lines when viewed in a text editor, as shown in Figure 5-15.

Program 5-16 shows what happens if we write the same four names without the `\n` escape sequence. Figure 5-16 shows the contents of the file that Program 5-16 creates. As you can see, all of the names appear on the same line in the file.

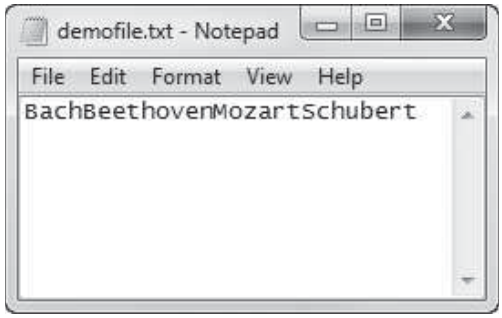
Program 5-16

```
1  // This program writes data to a single line in a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       ofstream outputFile;
9       outputFile.open("demofile.txt");
10
11      cout << "Now writing data to the file.\n";
12
13      // Write four names to the file.
14      outputFile << "Bach";
15      outputFile << "Beethoven";
16      outputFile << "Mozart";
17      outputFile << "Schubert";
18
19      // Close the file
20      outputFile.close();
21      cout << "Done.\n";
22      return 0;
23 }
```

Program Screen Output

```
Now writing data to the file.
Done.
```

Figure 5-16



Program 5-17 shows another example. This program reads three numbers from the keyboard as input and then saves those numbers in a file named Numbers.txt.

Program 5-17

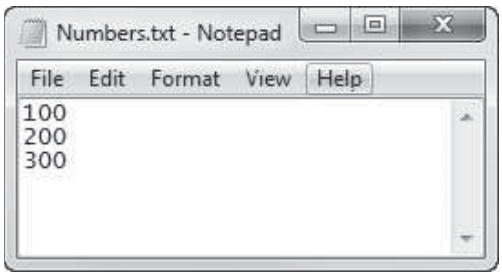
```
1  // This program writes user input to a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       ofstream outputFile;
9       int number1, number2, number3;
10
11      // Open an output file.
12      outputFile.open("Numbers.txt");
13
14      // Get three numbers from the user.
15      cout << "Enter a number: ";
16      cin >> number1;
17      cout << "Enter another number: ";
18      cin >> number2;
19      cout << "One more time. Enter a number: ";
20      cin >> number3;
21
22      // Write the numbers to the file.
23      outputFile << number1 << endl;
24      outputFile << number2 << endl;
25      outputFile << number3 << endl;
26      cout << "The numbers were saved to a file.\n";
27
28      // Close the file
29      outputFile.close();
30      cout << "Done.\n";
31      return 0;
32 }
```

Program Screen Output with Example Input Shown in Bold

```
Enter a number: 100 [Enter]
Enter another number: 200 [Enter]
One more time. Enter a number: 300 [Enter]
The numbers were saved to a file.
Done.
```

In Program 5-17, lines 23 through 25 write the contents of the `number1`, `number2`, and `number3` variables to the file. Notice that the `endl` manipulator is sent to the `outputFile` object immediately after each item. Sending the `endl` manipulator causes a newline to be written to the file. Figure 5-17 shows the file's contents displayed in Notepad, using the example input values 100, 200, and 300. As you can see, each item appears on a separate line in the file because of the `endl` manipulators.

Figure 5-17



Program 5-18 shows an example that reads strings as input from the keyboard and then writes those strings to a file. The program asks the user to enter the first names of three friends, and then it writes those names to a file named Friends.txt. Figure 5-18 shows an example of the Friends.txt file opened in Notepad.

Program 5-18

```
1  // This program writes user input to a file.
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9      ofstream outputFile;
10     string name1, name2, name3;
11
12     // Open an output file.
13     outputFile.open("Friends.txt");
14
```

(program continues)

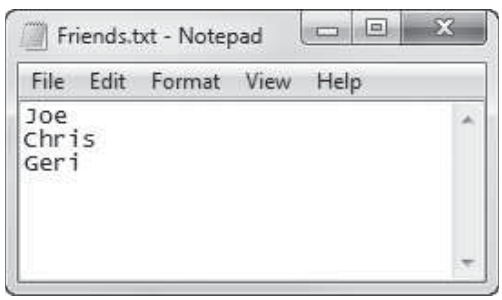
Program 5-18 *(continued)*

```
15       // Get the names of three friends.
16       cout << "Enter the names of three friends.\n";
17       cout << "Friend #1: ";
18       cin >> name1;
19       cout << "Friend #2: ";
20       cin >> name2;
21       cout << "Friend #3: ";
22       cin >> name3;
23
24       // Write the names to the file.
25       outputFile << name1 << endl;
26       outputFile << name2 << endl;
27       outputFile << name3 << endl;
28       cout << "The names were saved to a file.\n";
29
30       // Close the file
31       outputFile.close();
32       return 0;
33   }
```

Program Screen Output with Example Input Shown in Bold

```
Enter the names of three friends.
Friend #1: Joe [Enter]
Friend #2: Chris [Enter]
Friend #3: Geri [Enter]
The names were saved to a file.
```

Figure 5-18



Reading Data from a File

VideoNote
Reading Data from a File

The >> operator not only reads user input from the `cin` object, but also data from a file. Assuming `inputFile` is an `ifstream` object, the following statement shows the >> operator reading data from the file into the variable name:

```
inputFile >> name;
```

Let's look at an example. Assume the file Friends.txt exists, and it contains the names shown in Figure 5-18. Program 5-19 opens the file, reads the names and displays them on the screen, and then closes the file.

Program 5-19

```
1  // This program reads data from a file.
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9      ifstream inputFile;
10     string name;
11
12     inputFile.open("Friends.txt");
13     cout << "Reading data from the file.\n";
14
15     inputFile >> name;        // Read name 1 from the file
16     cout << name << endl;     // Display name 1
17
18     inputFile >> name;        // Read name 2 from the file
19     cout << name << endl;     // Display name 2
20
21     inputFile >> name;        // Read name 3 from the file
22     cout << name << endl;     // Display name 3
23
24     inputFile.close();        // Close the file
25     return 0;
26 }
```

Program Output

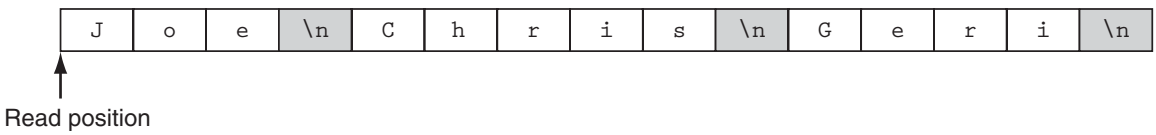
```
Reading data from the file.
Joe
Chris
Geri
```

The Read Position

When a file has been opened for input, the file stream object internally maintains a special value known as a *read position*. A file's read position marks the location of the next byte that will be read from the file. When an input file is opened, its read position is initially set to the first byte in the file. So, the first read operation extracts data starting at the first byte. As data is read from the file, the read position moves forward, toward the end of the file.

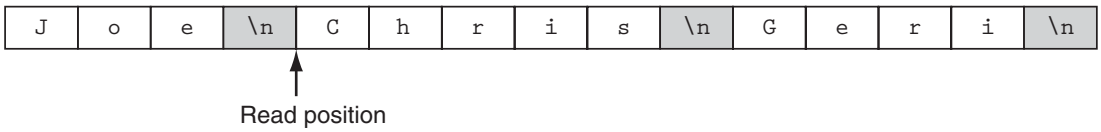
Let's see how this works with the example shown in Program 5-19. When the Friends.txt file is opened by the statement in line 12, the read position for the file will be positioned as shown in Figure 5-19.

Figure 5-19



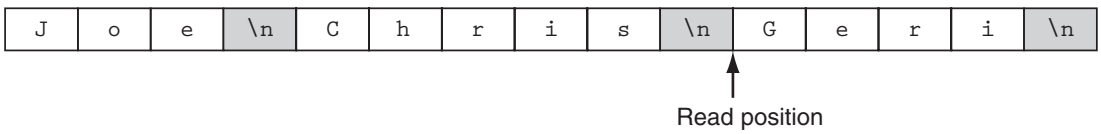
Keep in mind that when the >> operator extracts data from a file, it expects to read pieces of data that are separated by whitespace characters (spaces, tabs, or newlines). When the statement in line 15 executes, the >> operator reads data from the file's current read position, up to the `\n` character. The data that is read from the file is assigned to the `name` object. The `\n` character is also read from the file, but is not included as part of the data. So, the `name` object will hold the value `"Joe"` after this statement executes. The file's read position will then be at the location shown in Figure 5-20.

Figure 5-20



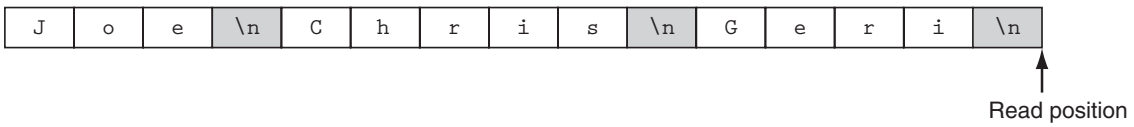
When the statement in line 18 executes, it reads the next item from the file, which is `"Chris"`, and assigns that value to the `name` object. After this statement executes, the file's read position will be advanced to the next item, as shown in Figure 5-21.

Figure 5-21



When the statement in line 21 executes, it reads the next item from the file, which is `"Geri"`, and assigns that value to the `name` object. After this statement executes, the file's read position will be advanced to the end of the file, as shown in Figure 5-22.

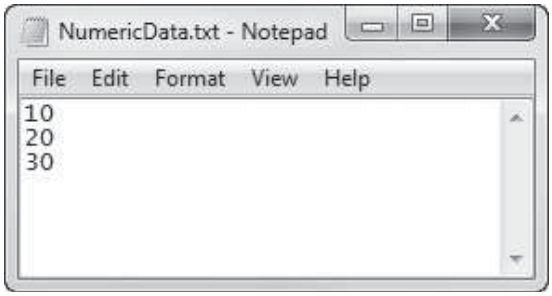
Figure 5-22



Reading Numeric Data From a Text File

Remember that when data is stored in a text file, it is encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. For example, suppose a text file contains numeric data, such as that shown in Figure 5-17. The numbers that you see displayed in the figure are stored in the file as the strings `"10"`, `"20"`, and `"30"`. Fortunately, you can use the >> operator to read data such as this from a text file, into a numeric variable, and the >> operator will automatically convert the data to a numeric data type. Program 5-20 shows an example. It opens the file shown in Figure 5-23, reads the three numbers from the file into `int` variables, and calculates their sum.

Figure 5-23



Program 5-20

```
1  // This program reads numbers from a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       ifstream inFile;
9       int value1, value2, value3, sum;
10
11      // Open the file.
12      inFile.open("NumericData.txt");
13
14      // Read the three numbers from the file.
15      inFile >> value1;
16      inFile >> value2;
17      inFile >> value3;
18
19      // Close the file.
20      inFile.close();
21
22      // Calculate the sum of the numbers.
23      sum = value1 + value2 + value3;
24
25      // Display the three numbers.
26      cout << "Here are the numbers:\n"
27           << value1 << " " << value2
28           << " " << value3 << endl;
29
30      // Display the sum of the numbers.
31      cout << "Their sum is: " << sum << endl;
32      return 0;
33 }
```

Program Output

```
Here are the numbers:
10 20 30
Their sum is: 60
```

Using Loops to Process Files

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in Program 5-21. This program gets sales amounts for a series of days from the user and writes those amounts to a file named Sales.txt. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days. Figure 5-24 shows the contents of the Sales.txt file containing the data entered by the user in the sample run.

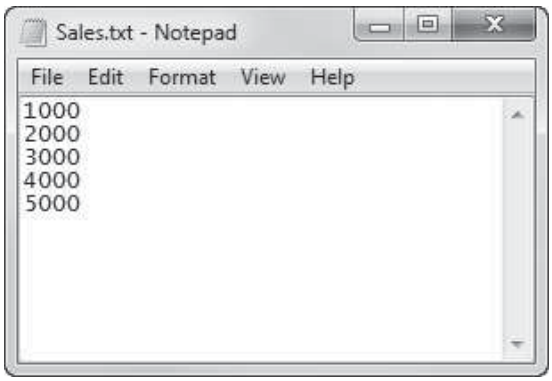
Program 5-21

```
1  // This program reads data from a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8      ofstream outputFile;  // File stream object
9      int numberOfDays;     // Number of days of sales
10     double sales;         // Sales amount for a day
11
12     // Get the number of days.
13     cout << "For how many days do you have sales? ";
14     cin >> numberOfDays;
15
16     // Open a file named Sales.txt.
17     outputFile.open("Sales.txt");
18
19     // Get the sales for each day and write it
20     // to the file.
21     for (int count = 1; count <= numberOfDays; count++)
22     {
23         // Get the sales for a day.
24         cout << "Enter the sales for day "
25              << count << ": ";
26         cin >> sales;
27
28         // Write the sales to the file.
29         outputFile << sales << endl;
30     }
31
32     // Close the file.
33     outputFile.close();
34     cout << "Data written to Sales.txt\n";
35     return 0;
36 }
```

Program Output (with Input Shown in Bold)

```
For how many days do you have sales? 5 [Enter]
Enter the sales for day 1: 1000.00 [Enter]
Enter the sales for day 2: 2000.00 [Enter]
Enter the sales for day 3: 3000.00 [Enter]
Enter the sales for day 4: 4000.00 [Enter]
Enter the sales for day 5: 5000.00 [Enter]
Data written to sales.txt.
```

Figure 5-24



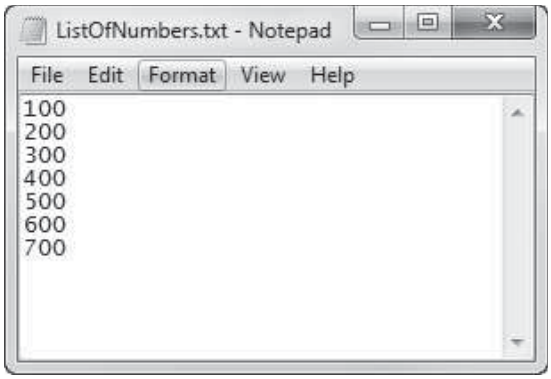
Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, suppose you need to write a program that displays all of the items in a file, but you do not know how many items the file contains. You can open the file and then use a loop to repeatedly read an item from the file and display it. However, an error will occur if the program attempts to read beyond the end of the file. The program needs some way of knowing when the end of the file has been reached so it will not try to read beyond it.

Fortunately, the >> operator not only reads data from a file, but also returns a true or false value indicating whether the data was successfully read or not. If the operator returns true, then a value was successfully read. If the operator returns false, it means that no value was read from the file.

Let's look at an example. A file named ListOfNumbers.txt, which is shown in Figure 5-25, contains a list of numbers. Without knowing how many numbers the file contains, Program 5-22 opens the file, reads all of the values it contains, and displays them.

Figure 5-25



Program 5-22

```
1  // This program reads data from a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       ifstream inputFile;
9       int number;
10
11      // Open the file.
12      inputFile.open("ListOfNumbers.txt");
13
14      // Read the numbers from the file and
15      // display them.
16      while (inputFile >> number)
17      {
18           cout << number << endl;
19      }
20
21      // Close the file.
22      inputFile.close();
23      return 0;
24 }
```

Program Output

```
100
200
300
400
500
600
700
```

Take a closer look at line 16:

```
while (inputFile >> number)
```

Notice that the statement that extracts data from the file is used as the Boolean expression in the `while` loop. It works like this:

- The expression `inputFile >> number` executes.
- If an item is successfully read from the file, the item is stored in the `number` variable, and the expression returns true to indicate that it succeeded. In that case, the statement in line 18 executes and the loop repeats.
- If there are no more items to read from the file, the expression `inputFile >> number` returns false, indicating that it did not read a value. In that case, the loop terminates.

Because the value returned from the >> operator controls the loop, it will read items from the file until the end of the file has been reached.

Testing for File Open Errors

Under certain circumstances, the `open` member function will not work. For example, the following code will fail if the file info.txt does not exist:

```
ifstream inputFile;
inputFile.open("info.txt");
```

There is a way to determine whether the `open` member function successfully opened the file. After you call the `open` member function, you can test the file stream object as if it were a Boolean expression. Program 5-23 shows an example.

Program 5-23

```
1  // This program tests for file open errors.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8      ifstream inputFile;
9      int number;
10
11     // Open the file.
12     inputFile.open("BadListOfNumbers.txt");
13
14     // If the file successfully opened, process it.
15     if (inputFile)
16     {
17         // Read the numbers from the file and
18         // display them.
19         while (inputFile >> number)
20         {
21             cout << number << endl;
22         }
23
```

(program continues)

Program 5-23 *(continued)*

```
24           // Close the file.
25           inputFile.close();
26      }
27      else
28      {
29           // Display an error message.
30           cout << "Error opening the file.\n";
31      }
32      return 0;
33  }
```

Program Output (Assume BadListOfNumbers.txt does not exist)

```
Error opening the file.
```

Let's take a closer look at certain parts of the code. Line 12 calls the `inputFile` object's `open` member function to open the file ListOfNumbers.txt. Then the `if` statement in line 15 tests the value of the `inputFile` object as if it were a Boolean expression. When tested this way, the `inputFile` object will give a true value if the file was successfully opened. Otherwise it will give a false value. The example output shows this program will display an error message if it could not open the file.

Another way to detect a failed attempt to open a file is with the `fail` member function, as shown in the following code:

```
ifstream inputFile;
inputFile.open("customers.txt");
if (inputFile.fail())
{
     cout << "Error opening file.\n";
}
else
{
     // Process the file.
}
```

The `fail` member function returns true when an attempted file operation is unsuccessful. When using file I/O, you should always test the file stream object to make sure the file was opened successfully. If the file could not be opened, the user should be informed and appropriate action taken by the program.

Letting the User Specify a Filename

In each of the previous examples, the name of the file that is opened is hard-coded as a string literal into the program. In many cases, you will want the user to specify the name of a file for the program to open. In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function. Program 5-24 shows an example. This is a modified version of Program 5-23. This version prompts the user to enter the name of the file. In line 15, the name that the user enters is stored in a `string` object named `filename`. In line 18, the `filename` object is passed as an argument to the `open` function.

Program 5-24

```
1  // This program lets the user enter a filename.
2  #include <iostream>
3  #include <string>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9       ifstream inputFile;
10      string filename;
11      int number;
12
13      // Get the filename from the user.
14      cout << "Enter the filename: ";
15      cin >> filename;
16
17      // Open the file.
18      inputFile.open(filename);
19
20      // If the file successfully opened, process it.
21      if (inputFile)
22      {
23           // Read the numbers from the file and
24           // display them.
25           while (inputFile >> number)
26           {
27                cout << number << endl;
28           }
29
30           // Close the file.
31           inputFile.close();
32      }
33      else
34      {
35           // Display an error message.
36           cout << "Error opening the file.\n";
37      }
38      return 0;
39 }
```

Program Output with Example Input Shown in Bold

```
Enter the filename: ListOfNumbers.txt [Enter]
100
200
300
400
500
600
700
```

Using the c_str Member Function in Older Versions of C++

In older versions of the C++ language (prior to C++ 11), a file stream object's open member function will not accept a string object as an argument. The open member function requires that you pass the name of the file as a null-terminated string, which is also known as a *C-string*. String literals are stored in memory as null-terminated C-strings, but string objects are not.

Fortunately, string objects have a member function named c_str that returns the contents of the object formatted as a null-terminated C-string. Here is the general format of how you call the function:

```
stringObject.c_str()
```

In the general format, *stringObject* is the name of a string object. The c_str function returns the string that is stored in *stringObject* as a null-terminated C-string.

For example, line 18 in Program 5-24 could be rewritten in the following manner to make the program compatible with an older version of C++:

```
inputFile.open(filename.c_str());
```

In this version of the statement, the value that is returned from filename.c_str() is passed as an argument to the open function.

Checkpoint

5.16 What is an output file? What is an input file?

5.17 What three steps must be taken when a file is used by a program?

5.18 What is the difference between a text file and a binary file?

5.19 What is the difference between sequential access and random access?

5.20 What type of file stream object do you create if you want to write data to a file?

5.21 What type of file stream object do you create if you want to read data from a file?

5.22 Write a short program that uses a for loop to write the numbers 1 through 10 to a file.

5.23 Write a short program that opens the file created by the program you wrote for Checkpoint 5.22, reads all of the numbers from the file, and displays them.

5.12 Optional Topics: Breaking and Continuing a Loop

CONCEPT: **The break statement causes a loop to terminate early. The continue statement causes a loop to stop its current iteration and begin the next one.**

WARNING! Use the break and continue statements with great caution. Because they bypass the normal condition that controls the loop's iterations, these statements make code difficult to understand and debug. For this reason, you should avoid using break and continue whenever possible. However, because they are part of the C++ language, we discuss them briefly in this section.