CHAPTER 12

Advanced File Operations

TOPICS

12.1 File Operations

CONCEPT: A file is a collection of data that is usually stored on a computer's disk. Data can be saved to files and then later reused.

Almost all real-world programs use files to store and retrieve data. Here are a few examples of familiar software packages that use files extensively.

- **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 reprinted.
- **Database Management Systems:** DBMSs are used to create and maintain databases. Databases are files that contain large collections of data, such as payroll records, inventories, sales statistics, and customer records.
- **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 to a file for use later.
- **Compilers:** Compilers translate the source code of a program, which is saved in a file, into an executable file. Throughout the previous chapters of this book you have created many C++ source files and compiled them to executable files.

Chapter 5 provided enough information for you to write programs that perform simple file operations. This chapter covers more advanced file operations and focuses primarily

on the fstream data type. As a review, Table 12-1 compares the ifstream, ofstream, and fstream data types. All of these data types require the fstream header file.

Table 12-1 File Stream

Data Type	Description
ifstream	Input File Stream. This data type can be used only to read data from files into memory.
ofstream	Output File Stream. This data type can be used to create files and write data to them.
fstream	File Stream. This data type can be used to create files, write data to them, and read data from them.

Using the fstream Data Type

You define an fstream object just as you define objects of other data types. The following statement defines an fstream object named dataFile.

```
fstream dataFile;
```

As with ifstream and ofstream objects, you use an fstream object's open function to open a file. An fstream object's open function requires two arguments, however. The first argument is a string containing the name of the file. The second argument is a file access flag that indicates the mode in which you wish to open the file. Here is an example.

```
dataFile.open("info.txt", ios::out);
```

The first argument in this function call is the name of the file, info.txt. The second argument is the file access flag ios::out. This tells C++ to open the file in output mode. Output mode allows data to be written to a file. The following statement uses the ios::in access flag to open a file in input mode, which allows data to be read from the file.

```
dataFile.open("info.txt", ios::in);
```

There are many file access flags, as listed in Table 12-2.

Table 12-2

File Access Flag	Meaning
ios::app	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
ios::ate	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
ios::binary	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
ios::in	Input mode. Data will be read from the file. If the file does not exist, it will not be created and the open function will fail.
ios::out	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
ios::trunc	If the file already exists, its contents will be deleted (truncated). This is the default mode used by ios::out.

Several flags may be used together if they are connected with the | operator. For example, assume `dataFile` is an `fstream` object in the following statement:

```
dataFile.open("info.txt", ios::in | ios::out);
```

This statement opens the file `info.txt` in both input and output modes. This means data may be written to and read from the file.

NOTE: When used by itself, the `ios::out` flag causes the file's contents to be deleted if the file already exists. When used with the `ios::in` flag, however, the file's existing contents are preserved. If the file does not already exist, it will be created.

The following statement opens the file in such a way that data will only be written to its end:

```
dataFile.open("info.txt", ios::out | ios::app);
```

By using different combinations of access flags, you can open files in many possible modes.

Program 12-1 uses an `fstream` object to open a file for output, and then writes data to the file.

Program 12-1

```
1  // This program uses an fstream object to write data to a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       fstream dataFile;
9
10      cout << "Opening file...\n";
11      dataFile.open("demofile.txt", ios::out);     // Open for output
12      cout << "Now writing data to the file.\n";
13      dataFile << "Jones\n";                       // Write line 1
14      dataFile << "Smith\n";                       // Write line 2
15      dataFile << "Willis\n";                      // Write line 3
16      dataFile << "Davis\n";                       // Write line 4
17      dataFile.close();                            // Close the file
18      cout << "Done.\n";
19      return 0;
20 }
```

Program Output

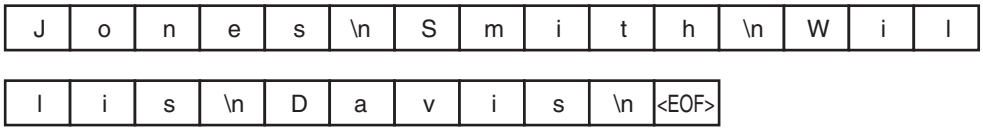
```
Opening file...
Now writing data to the file.
Done.
```

Output to File `demofile.txt`

```
Jones
Smith
Willis
Davis
```

The file output is shown for Program 12-1 the way it would appear if the file contents were displayed on the screen. The \n characters cause each name to appear on a separate line. The actual file contents, however, appear as a stream of characters as shown in Figure 12-1.

Figure 12-1



As you can see from the figure, \n characters are written to the file along with all the other characters. The characters are added to the file sequentially, in the order they are written by the program. The very last character is an *end-of-file marker*. It is a character that marks the end of the file and is automatically written when the file is closed. (The actual character used to mark the end of a file depends upon the operating system being used. It is always a nonprinting character. For example, some systems use control-Z.)

Program 12-2 is a modification of Program 12-1 that further illustrates the sequential nature of files. The file is opened, two names are written to it, and it is closed. The file is then reopened by the program in append mode (with the `ios::app` access flag). When a file is opened in append mode, its contents are preserved, and all subsequent output is appended to the file's end. Two more names are added to the file before it is closed and the program terminates.

Program 12-2

```
1  // This program writes data to a file, closes the file,
2  // then reopens the file and appends more data.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9      ofstream dataFile;
10
11     cout << "Opening file...\n";
12     // Open the file in output mode.
13     dataFile.open("demofile.txt", ios::out);
14     cout << "Now writing data to the file.\n";
15     dataFile << "Jones\n";                    // Write line 1
16     dataFile << "Smith\n";                    // Write line 2
17     cout << "Now closing the file.\n";
18     dataFile.close();                         // Close the file
19
20     cout << "Opening the file again...\n";
21     // Open the file in append mode.
22     dataFile.open("demofile.txt", ios::out | ios::app);
23     cout << "Writing more data to the file.\n";
24     dataFile << "Willis\n";                   // Write line 3
25     dataFile << "Davis\n";                    // Write line 4
26     cout << "Now closing the file.\n";
27     dataFile.close();                         // Close the file
```

```
28
29       cout << "Done.\n";
30       return 0;
31   }
```

Output to File `demofile.txt`

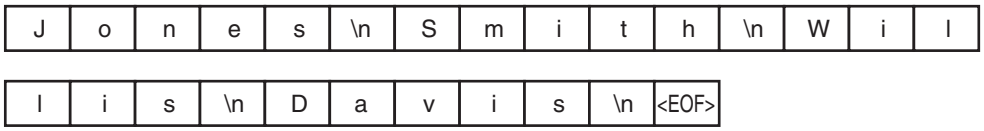
```
Jones
Smith
Willis
Davis
```

The first time the file is opened, the names are written as shown in Figure 12-2.

Figure 12-2

The file is closed, and an end-of-file character is automatically written. When the file is reopened, the new output is appended to the end of the file, as shown in Figure 12-3.

Figure 12-3



NOTE: If the `ios::out` flag had been alone, without `ios::app` the second time the file was opened, the file's contents would have been deleted. If this had been the case, the names Jones and Smith would have been erased, and the file would only have contained the names Willis and Davis.

File Open Modes with `ifstream` and `ofstream` Objects

The `ifstream` and `ofstream` data types each have a default mode in which they open files. This mode determines the operations that may be performed on the file and what happens if the file that is being opened already exists. Table 12-3 describes each data type's default open mode.

Table 12-3

File Type	Default Open Mode
`ofstream`	The file is opened for output only. Data may be written to the file, but not read from the file. If the file does not exist, it is created. If the file already exists, its contents are deleted (the file is truncated).
`ifstream`	The file is opened for input only. Data may be read from the file, but not written to it. The file's contents will be read from its beginning. If the file does not exist, the open function fails.

You cannot change the fact that ifstream files may only be read from and ofstream files may only be written to. You can, however, vary the way operations are carried out on these files by providing a file access flag as an optional second argument to the open function. The following code shows an example using an ofstream object.

```
ofstream outputFile;
outputFile.open("values.txt", ios::out|ios::app);
```

The ios::app flag specifies that data written to the values.txt file should be appended to its existing contents.

Checking for a File's Existence Before Opening It

Sometimes you want to determine whether a file already exists before opening it for output. You can do this by first attempting to open the file for input. If the file does not exist, the open operation will fail. In that case, you can create the file by opening it for output. The following code gives an example.

```
fstream dataFile;
dataFile.open("values.txt", ios::in);
if (dataFile.fail())
{
        // The file does not exist, so create it.
        dataFile.open("values.txt", ios::out);
        //
        // Continue to process the file...
        //
}

else    // The file already exists.
{
        dataFile.close();
        cout << "The file values.txt already exists.\n";
}
```

Opening a File with the File Stream Object Definition Statement

An alternative to using the open member function is to use the file stream object definition statement to open the file. Here is an example:

```
fstream dataFile("names.txt", ios::in | ios::out);
```

This statement defines an fstream object named dataFile and uses it to open the file names.txt. The file is opened in both input and output modes. This technique eliminates the need to call the open function when your program knows the name and access mode of the file at the time the object is defined. You may also use this technique with ifstream and ofstream objects, as shown in the following examples.

```
ifstream inputFile("info.txt");
ofstream outputFile("addresses.txt");
ofstream dataFile("customers.txt", ios::out|ios::app);
```

You may also test for errors after you have opened a file with this technique. The following code shows an example.

```
ifstream inputFile("SalesData.txt");
if (!inputFile)
     cout << "Error opening SalesData.txt.\n";
```

Checkpoint

12.1 Which file access flag would you use if you want all output to be written to the end of an existing file?

12.2 How do you use more than one file access flag?

12.3 Assuming that `diskInfo` is an `fstream` object, write a statement that opens the file `names.dat` for output.

12.4 Assuming that `diskInfo` is an `fstream` object, write a statement that opens the file `customers.txt` for output, where all output will be written to the end of the file.

12.5 Assuming that `diskInfo` is an `fstream` object, write a statement that opens the file `payable.txt` for both input and output.

12.6 Write a statement that defines an `fstream` object named `dataFile` and opens a file named `salesfigures.txt` for input. (*Note:* The file should be opened with the definition statement, not an `open` function call.)

12.2 File Output Formatting

CONCEPT: File output may be formatted in the same way that screen output is formatted.

The same output formatting techniques that are used with `cout`, which are covered in Chapter 3, may also be used with file stream objects. For example, the `setprecision` and `fixed` manipulators may be called to establish the number of digits of precision that floating point values are rounded to. Program 12-3 demonstrates this.

Program 12-3

```
1  // This program uses the setprecision and fixed
2  // manipulators to format file output.
3  #include <iostream>
4  #include <iomanip>
5  #include <fstream>
6  using namespace std;
7
8  int main()
9  {
10     fstream dataFile;
11     double num = 17.816392;
12
13     dataFile.open("numfile.txt", ios::out);     // Open in output mode
14
```

(program continues)

Program 12-3 *(continued)*

```
15       dataFile << fixed;             // Format for fixed-point notation
16       dataFile << num << endl;       // Write the number
17
18       dataFile << setprecision(4);   // Format for 4 decimal places
19       dataFile << num << endl;       // Write the number
20
21       dataFile << setprecision(3);   // Format for 3 decimal places
22       dataFile << num << endl;       // Write the number
23
24       dataFile << setprecision(2);   // Format for 2 decimal places
25       dataFile << num << endl;       // Write the number
26
27       dataFile << setprecision(1);   // Format for 1 decimal place
28       dataFile << num << endl;       // Write the number
29
30       cout << "Done.\n";
31       dataFile.close();              // Close the file
32       return 0;
33   }
```

Contents of File `numfile.txt`

```
17.816392
17.8164
17.816
17.82
17.8
```

Notice the file output is formatted just as `cout` would format screen output. Program 12-4 shows the `setw` stream manipulator being used to format file output into columns.

Program 12-4

```
1   // This program writes three rows of numbers to a file.
2   #include <iostream>
3   #include <fstream>
4   #include <iomanip>
5   using namespace std;
6
7   int main()
8   {
9        const int ROWS = 3;    // Rows to write
10       const int COLS = 3;    // Columns to write
11       int nums[ROWS][COLS] = { 2897, 5, 837,
12                                34, 7, 1623,
13                                390, 3456, 12 };
14       fstream outFile("table.txt", ios::out);
15
```

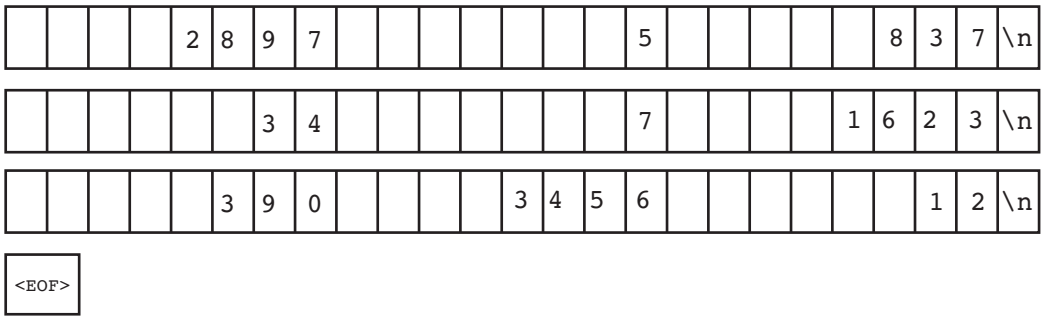
```
16      // Write the three rows of numbers with each
17      // number in a field of 8 character spaces.
18      for (int row = 0; row < ROWS; row++)
19      {
20          for (int col = 0; col < COLS; col++)
21          {
22              outFile << setw(8) << nums[row][col];
23          }
24          outFile << endl;
25      }
26      outFile.close();
27      cout << "Done.\n";
28      return 0;
29  }
```

Contents of File `table.txt`

```
    2897        5      837
      34        7     1623
     390     3456       12
```

Figure 12-4 shows the way the characters appear in the file.

Figure 12-4



12.3 Passing File Stream Objects to Functions

CONCEPT: File stream objects may be passed by reference to functions.

When writing actual programs, you'll want to create modularized code for handling file operations. File stream objects may be passed to functions, but they should always be passed by reference. The `openFile` function shown below uses an `fstream` reference object parameter:

VideoNote
Passing File Stream Objects to Functions

```
bool openFileIn(fstream &file, string name)
{
   bool status;
```

```
        file.open(name, ios::in);
        if (file.fail())
           status = false;
        else
           status = true;
        return status;
     }
```

The internal state of file stream objects changes with most every operation. They should always be passed to functions by reference to ensure internal consistency. Program 12-5 shows an example of how file stream objects may be passed as arguments to functions.

Program 12-5

```
1  // This program demonstrates how file stream objects may
2  // be passed by reference to functions.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  // Function prototypes
9  bool openFileIn(fstream &, string);
10 void showContents(fstream &);
11
12 int main()
13 {
14     fstream dataFile;
15
16     if (openFileIn(dataFile, "demofile.txt"))
17     {
18          cout << "File opened successfully.\n";
19          cout << "Now reading data from the file.\n\n";
20          showContents(dataFile);
21          dataFile.close();
22          cout << "\nDone.\n";
23     }
24     else
25          cout << "File open error!" << endl;
26
27     return 0;
28 }
29
30 //*********************************************************
31 // Definition of function openFileIn. Accepts a reference  *
32 // to an fstream object as an argument. The file is opened *
33 // for input. The function returns true upon success, false *
34 // upon failure.                                           *
35 //*********************************************************
36
```

```
37  bool openFileIn(fstream &file, string name)
38  {
39       file.open(name, ios::in);
40       if (file.fail())
41            return false;
42       else
43            return true;
44  }
45
46  //*********************************************************
47  // Definition of function showContents. Accepts an fstream  *
48  // reference as its argument. Uses a loop to read each name *
49  // from the file and displays it on the screen.             *
50  //*********************************************************
51
52  void showContents(fstream &file)
53  {
54       string line;
55
56       while (file >> line)
57       {
58            cout << line << endl;
59       }
60  }
```

Program Output

```
File opened successfully.
Now reading data from the file.

Jones
Smith
Willis
Davis

Done.
```

12.4 More Detailed Error Testing

CONCEPT: **All stream objects have error state bits that indicate the condition of the stream.**

All stream objects contain a set of bits that act as flags. These flags indicate the current state of the stream. Table 12-4 lists these bits.

These bits can be tested by the member functions listed in Table 12-5. (You've already learned about the `fail()` function.) One of the functions listed in the table, `clear()`, can be used to set a status bit.

Table 12-4

Bit	Description
`ios::eofbit`	Set when the end of an input stream is encountered.
`ios::failbit`	Set when an attempted operation has failed.
`ios::hardfail`	Set when an unrecoverable error has occurred.
`ios::badbit`	Set when an invalid operation has been attempted.
`ios::goodbit`	Set when all the flags above are not set. Indicates the stream is in good condition.

Table 12-5

Function	Description
`eof()`	Returns true (nonzero) if the `eofbit` flag is set, otherwise returns false.
`fail()`	Returns true (nonzero) if the `failbit` or `hardfail` flags are set, otherwise returns false.
`bad()`	Returns true (nonzero) if the `badbit` flag is set, otherwise returns false.
`good()`	Returns true (nonzero) if the `goodbit` flag is set, otherwise returns false.
`clear()`	When called with no arguments, clears all the flags listed above. Can also be called with a specific flag as an argument.

The function `showState`, shown here, accepts a file stream reference as its argument. It shows the state of the file by displaying the return values of the `eof()`, `fail()`, `bad()`, and `good()` member functions:

```
void showState(fstream &file)
{
   cout << "File Status:\n";
   cout << " eof bit: " << file.eof() << endl;
   cout << " fail bit: " << file.fail() << endl;
   cout << " bad bit: " << file.bad() << endl;
   cout << " good bit: " << file.good() << endl;
   file.clear();      // Clear any bad bits
}
```

Program 12-6 uses the `showState` function to display `testFile`'s status after various operations. First, the file is created and the integer value 10 is stored in it. The file is then closed and reopened for input. The integer is read from the file, and then a second read operation is performed. Because there is only one item in the file, the second read operation will result in an error.

Program 12-6

```
1  // This program demonstrates the return value of the stream
2  // object error testing member functions.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
```

```
// Function prototype
void showState(fstream &);

int main()
{
    int num = 10;

    // Open the file for output.
    fstream testFile("stuff.dat", ios::out);
    if (testFile.fail())
    {
        cout << "ERROR: Cannot open the file.\n";
        return 0;
    }

    // Write a value to the file.
    cout << "Writing the value " << num << " to the file.\n";
    testFile << num;

    // Show the bit states.
    showState(testFile);

    // Close the file.
    testFile.close();

    // Reopen the file for input.
    testFile.open("stuff.dat", ios::in);
    if (testFile.fail())
    {
        cout << "ERROR: Cannot open the file.\n";
        return 0;
    }

    // Read the only value from the file.
    cout << "Reading from the file.\n";
    testFile >> num;
    cout << "The value " << num << " was read.\n";

    // Show the bit states.
    showState(testFile);

    // No more data in the file, but force an invalid read operation.
    cout << "Forcing a bad read operation.\n";
    testFile >> num;

    // Show the bit states.
    showState(testFile);

    // Close the file.
    testFile.close();
    return 0;
}
```

(program continues)

Program 12-6 *(continued)*

```
60  //*************************************************************
61  // Definition of function showState. This function uses       *
62  // an fstream reference as its parameter. The return values of *
63  // the eof(), fail(), bad(), and good() member functions are   *
64  // displayed. The clear() function is called before the function *
65  // returns.                                                    *
66  //*************************************************************
67
68  void showState(fstream &file)
69  {
70       cout << "File Status:\n";
71       cout << "  eof bit: " << file.eof() << endl;
72       cout << "  fail bit: " << file.fail() << endl;
73       cout << "  bad bit: " << file.bad() << endl;
74       cout << "  good bit: " << file.good() << endl;
75       file.clear(); // Clear any bad bits
76  }
```

Program Output

```
Writing the value 10 to the file.
File Status:
     eof bit: 0
     fail bit: 0
     bad bit: 0
     good bit: 1
Reading from the file.
The value 10 was read.
File Status:
     eof bit: 1
     fail bit: 0
     bad bit: 0
     good bit: 1
Forcing a bad read operation.
File Status:
     eof bit: 1
     fail bit: 1
     bad bit: 0
     good bit: 0
```

12.5 Member Functions for Reading and Writing Files

CONCEPT: File stream objects have member functions for more specialized file reading and writing.

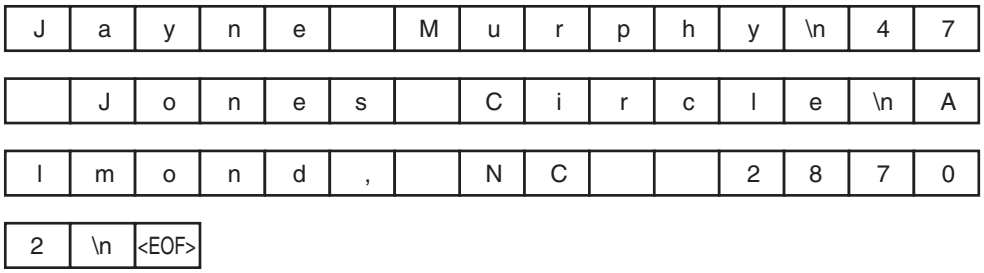
If whitespace characters are part of the data in a file, a problem arises when the file is read by the >> operator. Because the operator considers whitespace characters as delimiters, it

does not read them. For example, consider the file `murphy.txt`, which contains the following data:

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702
```

Figure 12-5 shows the way the data is recorded in the file.

Figure 12-5



The problem that arises from the use of the >> operator is evident in the output of Program 12-7.

Program 12-7

```
1  // This program demonstrates how the >> operator should not
2  // be used to read data that contain whitespace characters
3  // from a file.
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  using namespace std;
8
9  int main()
10 {
11     string input;     // To hold file input
12     fstream nameFile; // File stream object
13
14     // Open the file in input mode.
15     nameFile.open("murphy.txt", ios::in);
16
17     // If the file was successfully opened, continue.
18     if (nameFile)
19     {
20         // Read the file contents.
21         while (nameFile >> input)
22         {
23             cout << input;
24         }
25
```

(program continues)

Program 12-7 *(continued)*

```
26           // Close the file.
27           nameFile.close();
28       }
29       else
30       {
31           cout << "ERROR: Cannot open file.\n";
32       }
33       return 0;
34   }
```

Program Output

```
JayneMurphy47JonesCircleAlmond,NC28702
```

The `getline` Function

The problem with Program 12-7 can be solved by using the `getline` function. The function reads a "line" of data, including whitespace characters. Here is an example of the function call:

```
getline(dataFile, str,'\n');
```

The three arguments in this statement are explained as follows:

`dataFile`	This is the name of the file stream object. It specifies the stream object from which the data is to be read.
`str`	This is the name of a `string` object. The data read from the file will be stored here.
`'\n'`	This is a delimiter character of your choice. If this delimiter is encountered, it will cause the function to stop reading. (This argument is optional. If it's left out, `'\n'` is the default.)

The statement is an instruction to read a line of characters from the file. The function will read until it encounters a `\n`. The line of characters will be stored in the `str` object.

Program 12-8 is a modification of Program 12-7. It uses the `getline` function to read whole lines of data from the file.

Program 12-8

```
1  // This program uses the getline function to read a line of
2  // data from the file.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  int main()
```

```
{
     string input;     // To hold file input
     fstream nameFile; // File stream object

     // Open the file in input mode.
     nameFile.open("murphy.txt", ios::in);

     // If the file was successfully opened, continue.
     if (nameFile)
     {
          // Read an item from the file.
          getline(nameFile, input);

          // While the last read operation
          // was successful, continue.
          while (nameFile)
          {
               // Display the last item read.
               cout << input << endl;

               // Read the next item.
               getline(nameFile, input);
          }

          // Close the file.
          nameFile.close();
     }
     else
     {
          cout << "ERROR: Cannot open file.\n";
     }
     return 0;
}
```

Program Output

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702
```

Because the third argument of the `getline` function was left out in Program 12-8, its default value is `\n`. Sometimes you might want to specify another delimiter. For example, consider a file that contains multiple names and addresses and that is internally formatted in the following manner:

Contents of `names2.txt`

```
Jayne Murphy$47 Jones Circle$Almond, NC 28702\n$Bobbie Smith$
217 Halifax Drive$Canton, NC 28716\n$Bill Hammet$PO Box 121$
Springfield, NC 28357\n$
```

Think of this file as consisting of three records. A record is a complete set of data about a single item. Also, the records in the file above are made of three fields. The first field is the person's name. The second field is the person's street address or PO box number. The third field contains the person's city, state, and ZIP code. Notice that each field ends with a `$` character, and each record ends with a `\n` character. Program 12-9 demonstrates how a `getline` function can be used to detect the `$` characters.

Program 12-9

```
// This file demonstrates the getline function with
// a specified delimiter.
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    string input; // To hold file input

    // Open the file for input.
    fstream dataFile("names2.txt", ios::in);

    // If the file was successfully opened, continue.
    if (dataFile)
    {
        // Read an item using $ as a delimiter.
        getline(dataFile, input, '$');

        // While the last read operation
        // was successful, continue.
        while (dataFile)
        {
            // Display the last item read.
            cout << input << endl;

            // Read an item using $ as a delimiter.
            getline(dataFile, input, '$');
        }

        // Close the file.
        dataFile.close();
    }
    else
    {
        cout << "ERROR: Cannot open file.\n";
    }
    return 0;
}
```

Program Output

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702

Bobbie Smith
217 Halifax Drive
Canton, NC 28716

Bill Hammet
PO Box 121
Springfield, NC 28357
```

Notice that the `\n` characters, which mark the end of each record, are also part of the output. They cause an extra blank line to be printed on the screen, separating the records.

NOTE: When using a printable character, such as `$`, to delimit data in a file, be sure to select a character that will not actually appear in the data itself. Since it's doubtful that anyone's name or address contains a `$` character, it's an acceptable delimiter. If the file contained dollar amounts, however, another delimiter would have been chosen.

The `get` Member Function

The file stream object's `get` member function is also useful. It reads a single character from the file. Here is an example of its usage:

```
inFile.get(ch);
```

In this example, `ch` is a `char` variable. A character will be read from the file and stored in `ch`. Program 12-10 shows the function used in a complete program. The user is asked for the name of a file. The file is opened and the `get` function is used in a loop to read the file's contents, one character at a time.

Program 12-10

```
1  // This program asks the user for a file name. The file is
2  // opened and its contents are displayed on the screen.
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  int main()
9  {
10     string fileName;  // To hold the file name
11     char ch;          // To hold a character
12     fstream file;     // File stream object
13
14     // Get the file name
15     cout << "Enter a file name: ";
16     cin >> fileName;
17
18     // Open the file.
19     file.open(filename, ios::in);
20
21     // If the file was successfully opened, continue.
22     if (file)
23     {
24          // Get a character from the file.
25          file.get(ch);
26
27          // While the last read operation was
28          // successful, continue.
29          while (file)
30          {
```

(program continues)

Program 12-10 *(continued)*

```
31               // Display the last character read.
32               cout << ch;
33
34               // Read the next character
35               file.get(ch);
36           }
37
38           // Close the file.
39           file.close();
40       }
41       else
42           cout << fileName << " could not be opened.\n";
43       return 0;
44   }
```

Program 12-10 will display the contents of any file. The `get` function even reads whitespaces, so all the characters will be shown exactly as they appear in the file.

The `put` Member Function

The `put` member function writes a single character to the file. Here is an example of its usage:

```
outFile.put(ch);
```

In this statement, the variable `ch` is assumed to be a `char` variable. Its contents will be written to the file associated with the file stream object `outFile`. Program 12-11 demonstrates the `put` function.

Program 12-11

```
1  // This program demonstrates the put member function.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       char ch;  // To hold a character
9
10      // Open the file for output.
11      fstream dataFile("sentence.txt", ios::out);
12
13      cout << "Type a sentence and be sure to end it with a ";
14      cout << "period.\n";
15
16      // Get a sentence from the user one character at a time
17      // and write each character to the file.
18      cin.get(ch);
19      while (ch != '.')
20      {
21           dataFile.put(ch);
22           cin.get(ch);
23      }
```

```
24       dataFile.put(ch); // Write the period.
25
26       // Close the file.
27       dataFile.close();
28       return 0;
29   }
```

Program Output with Example Input Shown in Bold

```
Type a sentence and be sure to end it with a period.
I am on my way to becoming a great programmer. [Enter]
```

Resulting Contents of the File `sentence.txt`:

```
I am on my way to becoming a great programmer.
```

Checkpoint

12.7 Assume the file `input.txt` contains the following characters:

R	u	n		S	p	o	t		r	u	n	\n	S	e

e		S	p	o	t		r	u	n	\n	<EOF>

What will the following program display on the screen?

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
     fstream inFile("input.txt", ios::in);
     string item;
     inFile >> item;
     while (inFile)
     {
          cout << item << endl;
          inFile >> item;
     }
     inFile.close();
     return 0;
}
```

12.8 Describe the difference between reading a file with the >> operator and the `getline` function.

12.9 What will be stored in the file `out.txt` after the following program runs?

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
```

```
int main()
{
     const int SIZE = 5;
     ofstream outFile("out.txt");
     double nums[SIZE] = {100.279, 1.719, 8.602, 7.777, 5.099};

     outFile << fixed << setprecision(2);
     for (int count = 0; count < 5; count++)
     {
          outFile << setw(8) << nums[count];
     }
     outFile.close();
     return 0;
}
```

12.6 Focus on Software Engineering: Working with Multiple Files

CONCEPT: It's possible to have more than one file open at once in a program.

VideoNote
Working with Multiple Files

Quite often you will need to have multiple files open at once. In many real-world applications, data about a single item are categorized and written to several different files. For example, a payroll system might keep the following files:

`emp.dat`	A file that contains the following data about each employee: name, job title, address, telephone number, employee number, and the date hired.
`pay.dat`	A file that contains the following data about each employee: employee number, hourly pay rate, overtime rate, and number of hours worked in the current pay cycle.
`withhold.dat`	A file that contains the following data about each employee: employee number, dependents, and extra withholdings.

When the system is writing paychecks, you can see that it will need to open each of the files listed above and read data from them. (Notice that each file contains the employee number. This is how the program can locate a specific employee's data.)

In C++, you open multiple files by defining multiple file stream objects. For example, if you need to read from three files, you can define three file stream objects, such as:

```
ifstream file1, file2, file3;
```

Sometimes you will need to open one file for input and another file for output. For example, Program 12-12 asks the user for a file name. The file is opened and read. Each character is converted to uppercase and written to a second file called `out.txt`. This type of program can be considered a *filter*. Filters read the input of one file, changing the data in some fashion, and write it out to a second file. The second file is a modified version of the first file.

Program 12-12

```
// This program demonstrates reading from one file and writing
// to a second file.
#include <iostream>
#include <fstream>
#include <string>
#include <cctype> // Needed for the toupper function.
using namespace std;

int main()
{
    string fileName;    // To hold the file name
    char ch;            // To hold a character
    ifstream inFile;    // Input file

    // Open a file for output.
    ofstream outFile("out.txt");

    // Get the input file name.
    cout << "Enter a file name: ";
    cin >> fileName;

    // Open the file for input.
    inFile.open(filename);

    // If the input file opened successfully, continue.
    if (inFile)
    {
        // Read a char from file 1.
        inFile.get(ch);

        // While the last read operation was
        // successful, continue.
        while (inFile)
        {
            // Write uppercase char to file 2.
            outFile.put(toupper(ch));

            // Read another char from file 1.
            inFile.get(ch);
        }

        // Close the two files.
        inFile.close();
        outFile.close();
        cout << "File conversion done.\n";
    }
    else
        cout << "Cannot open " << fileName << endl;
    return 0;
}
```

(program output continues)

Program 12-12 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter a file name: hownow.txt [Enter]
File conversion done.
```

Contents of `hownow.txt`

```
how now brown cow.
How Now?
```

Resulting Contents of `out.txt`

```
HOW NOW BROWN COW.
HOW NOW?
```

12.7 Binary Files

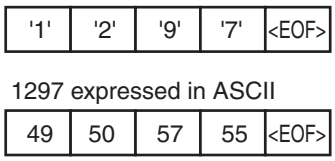
CONCEPT: **Binary files contain data that is not necessarily stored as ASCII text.**

All the files you've been working with so far have been text files. That means the data stored in the files has been formatted as ASCII text. Even a number, when stored in a file with the << operator, is converted to text. For example, consider the following program segment:

```
ofstream file("num.dat");
short x = 1297;
file << x;
```

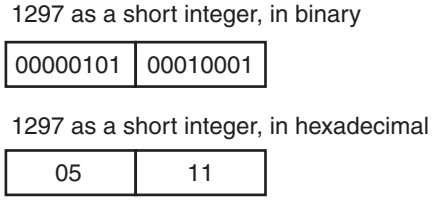
The last statement writes the contents of `x` to the file. When the number is written, however, it is stored as the characters `'1'`, `'2'`, `'9'`, and `'7'`. This is illustrated in Figure 12-6.

Figure 12-6



The number 1297 isn't stored in memory (in the variable `x`) in the fashion depicted in the figure above, however. It is formatted as a binary number, occupying two bytes on a typical PC. Figure 12-7 shows how the number is represented in memory, using binary or hexadecimal.

Figure 12-7



The representation of the number shown in Figure 12-7 is the way the "raw" data is stored in memory. Data can be stored in a file in its pure, binary format. The first step is to open the file in binary mode. This is accomplished by using the `ios::binary` flag. Here is an example:

```
file.open("stuff.dat", ios::out | ios::binary);
```

Notice the `ios::out` and `ios::binary` flags are joined in the statement with the | operator. This causes the file to be opened in both output and binary modes.

NOTE: By default, files are opened in text mode.

The `write` and `read` Member Functions

The file stream object's `write` member function is used to write binary data to a file. The general format of the `write` member function is

```
fileObject.write(address, size);
```

Let's look at the parts of this function call format.

- *`fileObject`* is the name of a file stream object.
- *`address`* is the starting address of the section of memory that is to be written to the file. This argument is expected to be the address of a `char` (or a pointer to a `char`).
- *`size`* is the number of bytes of memory to write. This argument must be an integer value.

For example, the following code uses a file stream object named `file` to write a character to a binary file.

```
char letter = 'A';
file.write(&letter, sizeof(letter));
```

The first argument passed to the `write` function is the address of the `letter` variable. This tells the `write` function where the data that is to be written to the file is located. The second argument is the size of the `letter` variable, which is returned from the `sizeof` operator. This tells the `write` function the number of bytes of data to write to the file. Because the sizes of data types can vary among systems, it is best to use the `sizeof` operator to determine the number of bytes to write. After this function call executes, the contents of the `letter` variable will be written to the binary file associated with the `file` object.

The following code shows another example. This code writes an entire `char` array to a binary file.

```
char data[] = {'A', 'B', 'C', 'D'};
file.write(data, sizeof(data));
```

In this code, the first argument is the name of the `data` array. By passing the name of the array we are passing a pointer to the beginning of the array. Because `data` is an array of `char` values, the name of the array is a pointer to a `char`. The second argument passes the name of the array to the `sizeof` operator. When the name of an array is passed to the `sizeof` operator, the operator returns the number of bytes allocated to the array. After this function call executes, the contents of the `data` array will be written to the binary file associated with the `file` object.

The `read` member function is used to read binary data from a file into memory. The general format of the `read` member function is

```
fileObject.read(address, size);
```

Here are the parts of this function call format:

- *`fileObject`* is the name of a file stream object.
- *`address`* is the starting address of the section of memory where the data being read from the file is to be stored. This is expected to be the address of a `char` (or a pointer to a `char`).
- *`size`* is the number of bytes of memory to read from the file. This argument must be an integer value.

For example, suppose we want to read a single character from a binary file and store that character in the `letter` variable. The following code uses a file stream object named `file` to do just that.

```
char letter;
file.read(&letter, sizeof(letter));
```

The first argument passed to the `read` function is the address of the `letter` variable. This tells the `read` function where to store the value that is read from the file. The second argument is the size of the `letter` variable. This tells the `read` function the number of bytes to read from the file. After this function executes, the `letter` variable will contain a character that was read from the file.

The following code shows another example. This code reads enough data from a binary file to fill an entire `char` array.

```
char data[4];
file.read(data, sizeof(data));
```

In this code, the first argument is the address of the `data` array. The second argument is the number of bytes allocated to the array. On a system that uses 1-byte characters, this function will read four bytes from the file and store them in the `data` array.

Program 12-13 demonstrates writing a `char` array to a file and then reading the data from the file back into memory.

Program 12-13

```
1  // This program uses the write and read functions.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       const int SIZE = 4;
9       char data[SIZE] = {'A', 'B', 'C', 'D'};
10      fstream file;
11
```

```
12      // Open the file for output in binary mode.
13      file.open("test.dat", ios::out | ios::binary);
14
15      // Write the contents of the array to the file.
16      cout << "Writing the characters to the file.\n";
17      file.write(data, sizeof(data));
18
19      // Close the file.
20      file.close();
21
22      // Open the file for input in binary mode.
23      file.open("test.dat", ios::in | ios::binary);
24
25      // Read the contents of the file into the array.
26      cout << "Now reading the data back into memory.\n";
27      file.read(data, sizeof(data));
28
29      // Display the contents of the array.
30      for (int count = 0; count < SIZE; count++)
31          cout << data[count] << " ";
32      cout << endl;
33
34      // Close the file.
35      file.close();
36      return 0;
37  }
```

Program Output

```
Writing the characters to the file.
Now reading the data back into memory.
A B C D
```

Writing Data Other Than `char` to Binary Files

Because the `write` and `read` member functions expect their first argument to be a pointer to a `char`, you must use a type cast when writing and reading items that are of other data types. To convert a pointer from one type to another you should use the `reinterpret_cast` type cast. The general format of the type cast is

```
reinterpret_cast<dataType>(value)
```

where *`dataType`* is the data type that you are converting to, and *`value`* is the value that you are converting. For example, the following code uses the type cast to store the address of an `int` in a `char` pointer variable.

```
int x = 65;
char *ptr = nullptr;
ptr = reinterpret_cast<char *>(&x);
```

The following code shows how to use the type cast to pass the address of an integer as the first argument to the `write` member function.

```
int x = 27;
file.write(reinterpret_cast<char *>(&x), sizeof(x));
```

After the function executes, the contents of the variable `x` will be written to the binary file associated with the `file` object. The following code shows an `int` array being written to a binary file.

```
const int SIZE = 10;
int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
```

After this function call executes, the contents of the `numbers` array will be written to the binary file. The following code shows values being read from the file and stored into the `numbers` array.

```
const int SIZE = 10;
int numbers[SIZE];
file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));
```

Program 12-14 demonstrates writing an `int` array to a file and then reading the data from the file back into memory.

Program 12-14

```
1  // This program uses the write and read functions.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8       const int SIZE = 10;
9       fstream file;
10      int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12      // Open the file for output in binary mode.
13      file.open("numbers.dat", ios::out | ios::binary);
14
15      // Write the contents of the array to the file.
16      cout << "Writing the data to the file.\n";
17      file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
18
19      // Close the file.
20      file.close();
21
22      // Open the file for input in binary mode.
23      file.open("numbers.dat", ios::in | ios::binary);
24
25      // Read the contents of the file into the array.
26      cout << "Now reading the data back into memory.\n";
27      file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));
28
29      // Display the contents of the array.
30      for (int count = 0; count < SIZE; count++)
31           cout << numbers[count] << " ";
32      cout << endl;
33
34      // Close the file.
35      file.close();
36      return 0;
37 }
```

Program Output

```
Writing the data to the file.
Now reading the data back into memory.
1 2 3 4 5 6 7 8 9 10
```

12.8 Creating Records with Structures

CONCEPT: **Structures may be used to store fixed-length records to a file.**

Earlier in this chapter the concept of fields and records was introduced. A field is an individual piece of data pertaining to a single item. A record is made up of fields and is a complete set of data about a single item. For example, a set of fields might be a person's name, age, address, and phone number. Together, all those fields that pertain to one person make up a record.

In C++, structures provide a convenient way to organize data into fields and records. For example, the following code could be used to create a record containing data about a person.

```
const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;

struct Info
{
     char name[NAME_SIZE];
     int age;
     char address1[ADDR_SIZE];
     char address2[ADDR_SIZE];
     char phone[PHONE_SIZE];
};
```

Besides providing an organizational structure for data, structures also package data into a single unit. For example, assume the structure variable `person` is defined as

```
Info person;
```

Once the members (or fields) of `person` are filled with data, the entire variable may be written to a file using the `write` function:

```
file.write(reinterpret_cast<char *>(&person), sizeof(person));
```

The first argument is the address of the `person` variable. The `reinterpret_cast` operator is used to convert the address to a `char` pointer. The second argument is the `sizeof` operator with `person` as its `argument`. This returns the number of bytes used by the `person` structure. Program 12-15 demonstrates this technique.

NOTE: Because structures can contain a mixture of data types, you should always use the `ios::binary` mode when opening a file to store them.

Program 12-15

```
// This program uses a structure variable to store a record to a file.
#include <iostream>
#include <fstream>
using namespace std;

// Array sizes
const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;

// Declare a structure for the record.
struct Info
{
    char name[NAME_SIZE];
    int age;
    char address1[ADDR_SIZE];
    char address2[ADDR_SIZE];
    char phone[PHONE_SIZE];
};

int main()
{
    Info person;   // To hold info about a person
    char again;    // To hold Y or N

    // Open a file for binary output.
    fstream people("people.dat", ios::out | ios::binary);

    do
    {
        // Get data about a person.
        cout << "Enter the following data about a "
             << "person:\n";
        cout << "Name: ";
        cin.getline(person.name, NAME_SIZE);
        cout << "Age: ";
        cin >> person.age;
        cin.ignore(); // Skip over the remaining newline.
        cout << "Address line 1: ";
        cin.getline(person.address1, ADDR_SIZE);
        cout << "Address line 2: ";
        cin.getline(person.address2, ADDR_SIZE);
        cout << "Phone: ";
        cin.getline(person.phone, PHONE_SIZE);

        // Write the contents of the person structure to the file.
        people.write(reinterpret_cast<char *>(&person),
                     sizeof(person));

        // Determine whether the user wants to write another record.
        cout << "Do you want to enter another record? ";
        cin >> again;
        cin.ignore(); // Skip over the remaining newline.
    } while (again == 'Y' || again == 'y');
```

```
53
54       // Close the file.
55       people.close();
56       return 0;
57   }
```

Program Output with Example Input Shown in Bold

```
Enter the following data about a person:
Name: Charlie Baxter [Enter]
Age: 42 [Enter]
Address line 1: 67 Kennedy Blvd. [Enter]
Address line 2: Perth, SC 38754 [Enter]
Phone: (803)555-1234 [Enter]
Do you want to enter another record? Y [Enter]
Enter the following data about a person:
Name: Merideth Murney [Enter]
Age: 22 [Enter]
Address line 1: 487 Lindsay Lane [Enter]
Address line 2: Hazelwood, NC 28737 [Enter]
Phone: (828)555-9999 [Enter]
Do you want to enter another record? N [Enter]
```

Program 12-15 allows you to build a file by filling the members of the `person` variable, and then writing the variable to the file. Program 12-16 opens the file and reads each record into the `person` variable, then displays the data on the screen.

Program 12-16

```
1   // This program uses a structure variable to read a record from a file.
2   #include <iostream>
3   #include <fstream>
4   using namespace std;
5
6   const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
7
8   // Declare a structure for the record.
9   struct Info
10  {
11      char name[NAME_SIZE];
12      int age;
13      char address1[ADDR_SIZE];
14      char address2[ADDR_SIZE];
15      char phone[PHONE_SIZE];
16  };
17
18  int main()
19  {
20      Info person;     // To hold info about a person
21      char again;      // To hold Y or N
22      fstream people;  // File stream object
23
24      // Open the file for input in binary mode.
25      people.open("people.dat", ios::in | ios::binary);
26
```

(program continues)

Program 12-16 *(continued)*

```
27      // Test for errors.
28      if (!people)
29      {
30           cout << "Error opening file. Program aborting.\n";
31           return 0;
32      }
33
34
35      cout << "Here are the people in the file:\n\n";
36      // Read the first record from the file.
37      people.read(reinterpret_cast<char *>(&person),
38                  sizeof(person));
39
40      // While not at the end of the file, display
41      // the records.
42      while (!people.eof())
43      {
44           // Display the record.
45           cout << "Name: ";
46           cout << person.name << endl;
47           cout << "Age: ";
48           cout << person.age << endl;
49           cout << "Address line 1: ";
50           cout << person.address1 << endl;
51           cout << "Address line 2: ";
52           cout << person.address2 << endl;
53           cout << "Phone: ";
54           cout << person.phone << endl;
55
56           // Wait for the user to press the Enter key.
57           cout << "\nPress the Enter key to see the next record.\n";
58           cin.get(again);
59
60           // Read the next record from the file.
61           people.read(reinterpret_cast<char *>(&person),
62                       sizeof(person));
63      }
64
65      cout << "That's all the data in the file!\n";
66      people.close();
67      return 0;
68  }
```

Program Output (Using the same file created by Program 12-15 as input)

```
Here are the people in the file:

Name: Charlie Baxter
Age: 42
Address line 1: 67 Kennedy Blvd.
Address line 2: Perth, SC 38754
Phone: (803)555-1234
```

```
Press the Enter key to see the next record.

Name: Merideth Murney
Age: 22
Address line 1: 487 Lindsay Lane
Address line 2: Hazelwood, NC 28737
Phone: (828)555-9999

Press the Enter key to see the next record.

That's all the data in the file!
```

NOTE: Structures containing pointers cannot be correctly stored to disk using the techniques of this section. This is because if the structure is read into memory on a subsequent run of the program, it cannot be guaranteed that all program variables will be at the same memory locations. Because `string` class objects contain implicit pointers, they cannot be a part of a structure that has to be stored.

12.9 Random-Access Files

CONCEPT: **Random access means nonsequentially accessing data in a file.**

All of the programs created so far in this chapter have performed *sequential file access*. When a file is opened, the position where reading and/or writing will occur is at the file's beginning (unless the `ios::app` mode is used, which causes data to be written to the end of the file). If the file is opened for output, bytes are written to it one after the other. If the file is opened for input, data is read beginning at the first byte. As the reading or writing continues, the file stream object's read/write position advances sequentially through the file's contents.

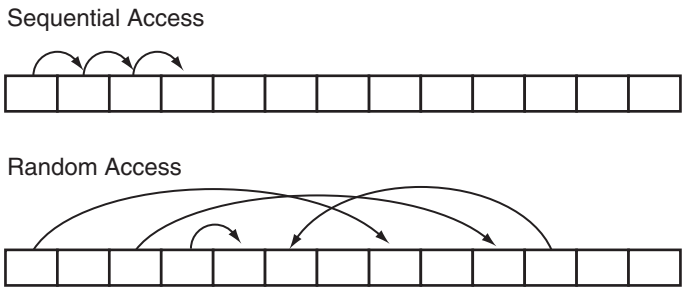
The problem with sequential file access is that in order to read a specific byte from the file, all the bytes that precede it must be read first. For instance, if a program needs data stored at the hundredth byte of a file, it will have to read the first 99 bytes to reach it. If you've ever listened to a cassette tape player, you understand sequential access. To listen to a song at the end of the tape, you have to listen to all the songs that come before it, or fast-forward over them. There is no way to immediately jump to that particular song.

Although sequential file access is useful in many circumstances, it can slow a program down tremendously. If the file is very large, locating data buried deep inside it can take a long time. Alternatively, C++ allows a program to perform *random file access*. In random file access, a program may immediately jump to any byte in the file without first reading the preceding bytes. The difference between sequential and random file access is like the difference between a cassette tape and a compact disc. When listening to a CD, there is no need to listen to or fast forward over unwanted songs. You simply jump to the track that you want to listen to. This is illustrated in Figure 12-8.

The `seekp` and `seekg` Member Functions

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are `seekp` and `seekg`. The `seekp` function is used with

Figure 12-8



files opened for output, and `seekg` is used with files opened for input. (It makes sense if you remember that "p" stands for "put" and "g" stands for "get." `seekp` is used with files that you put data into, and `seekg` is used with files you get data out of.)

Here is an example of `seekp`'s usage:

```
file.seekp(20L, ios::beg);
```

The first argument is a `long` integer representing an offset into the file. This is the number of the byte you wish to move to. In this example, `20L` is used. (Remember, the L suffix forces the compiler to treat the number as a long integer.) This statement moves the file's write position to byte number 20. (All numbering starts at 0, so byte number 20 is actually the twenty-first byte.)

The second argument is called the mode, and it designates where to calculate the offset *from*. The flag `ios::beg` means the offset is calculated from the beginning of the file. Alternatively, the offset can be calculated from the end of the file or the current position in the file. Table 12-6 lists the flags for all three of the random-access modes.

Table 12-6

Mode Flag	Description
`ios::beg`	The offset is calculated from the beginning of the file.
`ios::end`	The offset is calculated from the end of the file.
`ios::cur`	The offset is calculated from the current position.

Table 12-7 shows examples of `seekp` and `seekg` using the various mode flags.

Notice that some of the examples in Table 12-7 use a negative offset. Negative offsets result in the read or write position being moved backward in the file, while positive offsets result in a forward movement.

Assume the file `letters.txt` contains the following data:

```
abcdefghijklmnopqrstuvwxyz
```

Program 12-17 uses the `seekg` function to jump around to different locations in the file, retrieving a character after each stop.

Table 12-7

Statement	How It Affects the Read/Write Position
`file.seekp(32L, ios::beg);`	Sets the write position to the 33rd byte (byte 32) from the beginning of the file.
`file.seekp(-10L, ios::end);`	Sets the write position to the 10th byte from the end of the file.
`file.seekp(120L, ios::cur);`	Sets the write position to the 121st byte (byte 120) from the current position.
`file.seekg(2L, ios::beg);`	Sets the read position to the 3rd byte (byte 2) from the beginning of the file.
`file.seekg(-100L, ios::end);`	Sets the read position to the 100th byte from the end of the file.
`file.seekg(40L, ios::cur);`	Sets the read position to the 41st byte (byte 40) from the current position.
`file.seekg(0L, ios::end);`	Sets the read position to the end of the file.

Program 12-17

```
1  // This program demonstrates the seekg function.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8      char ch; // To hold a character
9
10     // Open the file for input.
11     fstream file("letters.txt", ios::in);
12
13     // Move to byte 5 from the beginning of the file
14     // (the 6th byte) and read the character there.
15     file.seekg(5L, ios::beg);
16     file.get(ch);
17     cout << "Byte 5 from beginning: " << ch << endl;
18
19     // Move to the 10th byte from the end of the file
20     // and read the character there.
21     file.seekg(-10L, ios::end);
22     file.get(ch);
23     cout << "10th byte from end: " << ch << endl;
24
25     // Move to byte 3 from the current position
26     // (the 4th byte) and read the character there.
27     file.seekg(3L, ios::cur);
28     file.get(ch);
29     cout << "Byte 3 from current: " << ch << endl;
30
31     file.close();
32     return 0;
33 }
```

(program output continues)

Program 12-17 *(continued)*

Program Screen Output

```
Byte 5 from beginning: f
10th byte from end: q
Byte 3 from current: u
```

Program 12-18 shows a more robust example of the `seekg` function. It opens the `people.dat` file created by Program 12-15. The file contains two records. Program 12-18 displays record 1 (the second record) first, then displays record 0.

The program has two important functions other than `main`. The first, `byteNum`, takes a record number as its argument and returns that record's starting byte. It calculates the record's starting byte by multiplying the record number by the size of the `Info` structure. This returns the offset of that record from the beginning of the file. The second function, `showRec`, accepts an `Info` structure as its argument and displays its contents on the screen.

Program 12-18

```
1  // This program randomly reads a record of data from a file.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
7
8  // Declare a structure for the record.
9  struct Info
10 {
11     char name[NAME_SIZE];
12     int age;
13     char address1[ADDR_SIZE];
14     char address2[ADDR_SIZE];
15     char phone[PHONE_SIZE];
16 };
17
18 // Function Prototypes
19 long byteNum(int);
20 void showRec(Info);
21
22 int main()
23 {
24     Info person;     // To hold info about a person
25     fstream people;  // File stream object
26
27     // Open the file for input in binary mode.
28     people.open("people.dat", ios::in | ios::binary);
29
```

```
30      // Test for errors.
31      if (!people)
32      {
33          cout << "Error opening file. Program aborting.\n";
34          return 0;
35      }
36
37      // Read and display record 1 (the second record).
38      cout << "Here is record 1:\n";
39      people.seekg(byteNum(1), ios::beg);
40      people.read(reinterpret_cast<char *>(&person), sizeof(person));
41      showRec(person);
42
43      // Read and display record 0 (the first record).
44      cout << "\nHere is record 0:\n";
45      people.seekg(byteNum(0), ios::beg);
46      people.read(reinterpret_cast<char *>(&person), sizeof(person));
47      showRec(person);
48
49      // Close the file.
50      people.close();
51      return 0;
52  }
53
54  //*********************************************************
55  // Definition of function byteNum. Accepts an integer as    *
56  // its argument. Returns the byte number in the file of the *
57  // record whose number is passed as the argument.           *
58  //*********************************************************
59
60  long byteNum(int recNum)
61  {
62      return sizeof(Info) * recNum;
63  }
64
65  //*********************************************************
66  // Definition of function showRec. Accepts an Info structure *
67  // as its argument, and displays the structure's contents.  *
68  //*********************************************************
69
70  void showRec(Info record)
71  {
72      cout << "Name: ";
73      cout << record.name << endl;
74      cout << "Age: ";
75      cout << record.age << endl;
76      cout << "Address line 1: ";
77      cout << record.address1 << endl;
78      cout << "Address line 2: ";
79      cout << record.address2 << endl;
```

(program continues)

Program 12-18 *(continued)*

```
80       cout << "Phone: ";
81       cout << record.phone << endl;
82   }
```

Program Output (Using the same file created by Program 12–15 as input)

```
Here is record 1:
Name: Merideth Murney
Age: 22
Address line 1: 487 Lindsay Lane
Address line 2: Hazelwood, NC 28737
Phone: (828)555-9999

Here is record 0:
Name: Charlie Baxter
Age: 42
Address line 1: 67 Kennedy Blvd.
Address line 2: Perth, SC 38754
Phone: (803)555-1234
```

WARNING! If a program has read to the end of a file, you must call the file stream object's `clear` member function before calling `seekg` or `seekp`. This clears the file stream object's `eof` flag. Otherwise, the `seekg` or `seekp` function will not work.

The `tellp` and `tellg` Member Functions

File stream objects have two more member functions that may be used for random file access: `tellp` and `tellg`. Their purpose is to return, as a long integer, the current byte number of a file's read and write position. As you can guess, `tellp` returns the write position and `tellg` returns the read position. Assuming `pos` is a long integer, here is an example of the functions' usage:

```
pos = outFile.tellp();
pos = inFile.tellg();
```

One application of these functions is to determine the number of bytes that a file contains. The following example demonstrates how to do this using the `tellg` function.

```
file.seekg(0L, ios::end);
numBytes = file.tellg();
cout << "The file has " << numBytes << " bytes.\n";
```

First the `seekg` member function is used to move the read position to the last byte in the file. Then the `tellg` function is used to get the current byte number of the read position.

Program 12-19 demonstrates the `tellg` function. It opens the `letters.txt` file, which was also used in Program 12-17. The file contains the following characters:

```
abcdefghijklmnopqrstuvwxyz
```

Program 12-19

```
// This program demonstrates the tellg function.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    long offset;    // To hold an offset amount
    long numBytes;  // To hold the file size
    char ch;        // To hold a character
    char again;     // To hold Y or N

    // Open the file for input.
    fstream file("letters.txt", ios::in);

    // Determine the number of bytes in the file.
    file.seekg(0L, ios::end);
    numBytes = file.tellg();
    cout << "The file has " << numBytes << " bytes.\n";

    // Go back to the beginning of the file.
    file.seekg(0L, ios::beg);

    // Let the user move around within the file.
    do
    {
        // Display the current read position.
        cout << "Currently at position " << file.tellg() << endl;

        // Get a byte number from the user.
        cout << "Enter an offset from the beginning of the file: ";
        cin >> offset;

        // Move the read position to that byte, read the
        // character there, and display it.
        if (offset >= numBytes)   // Past the end of the file?
            cout << "Cannot read past the end of the file.\n";
        else
        {
            file.seekg(offset, ios::beg);
            file.get(ch);
            cout << "Character read: " << ch << endl;
        }

        // Does the user want to try this again?
        cout << "Do it again? ";
        cin >> again;
    } while (again == 'Y' || again == 'y');

    // Close the file.
    file.close();
    return 0;
}
```

(program output continues)

Program 12-19 *(continued)*

Program Output with Example Input Shown in Bold

```
The file has 26 bytes.
Currently at position 0
Enter an offset from the beginning of the file: 5 [Enter]
Character read: f
Do it again? y [Enter]
Currently at position 6
Enter an offset from the beginning of the file: 0 [Enter]
Character read: a
Do it again? y [Enter]
Currently at position 1
Enter an offset from the beginning of the file: 26 [Enter]
Cannot read past the end of the file.
Do it again? n [Enter]
```

Rewinding a Sequential-Access File with `seekg`

Sometimes when processing a sequential file, it is necessary for a program to read the contents of the file more than one time. For example, suppose a program searches a file for an item specified by the user. The program must open the file, read its contents, and determine if the specified item is in the file. If the user needs to search the file again for another item, the program must read the file's contents again.

One simple approach for reading a file's contents more than once is to close and reopen the file, as shown in the following code example.

```
dataFile.open("file.txt", ios::in);      // Open the file.

//
// Read and process the file's contents.
//

dataFile.close();                        // Close the file.
dataFile.open("file.txt", ios::in);      // Open the file again.

//
// Read and process the file's contents again.
//

dataFile.close();                        // Close the file.
```

Each time the file is reopened, its read position is located at the beginning of the file. The read position is the byte in the file that will be read with the next read operation.

Another approach is to "rewind" the file. This means moving the read position to the beginning of the file without closing and reopening it. This is accomplished with the file stream object's `seekg` member function to move the read position back to the beginning of the file. The following example code demonstrates.

```
dataFile.open("file.txt", ios::in);      // Open the file.

//
// Read and process the file's contents.
//
```

```
dataFile.clear();                       // Clear the eof flag.
dataFile.seekg(0L, ios::beg);           // Rewind the read position.

//
// Read and process the file's contents again.
//

dataFile.close();                       // Close the file.
```

Notice that prior to calling the `seekg` member function, the `clear` member function is called. As previously mentioned this clears the file object's `eof` flag and is necessary only if the program has read to the end of the file. This approach eliminates the need to close and reopen the file each time the file's contents are processed.

12.10 Opening a File for Both Input and Output

CONCEPT: You may perform input and output on an `fstream` file without closing it and reopening it.

Sometimes you'll need to perform both input and output on a file without closing and reopening it. For example, consider a program that allows you to search for a record in a file and then make changes to it. A read operation is necessary to copy the data from the file to memory. After the desired changes have been made to the data in memory, a write operation is necessary to replace the old data in the file with the new data in memory.

Such operations are possible with `fstream` objects. The `ios::in` and `ios::out` file access flags may be joined with the | operator, as shown in the following statement.

```
fstream file("data.dat", ios::in | ios::out)
```

The same operation may be accomplished with the `open` member function:

```
file.open("data.dat", ios::in | ios::out);
```

You may also specify the `ios::binary` flag if binary data is to be written to the file. Here is an example:

```
file.open("data.dat", ios::in | ios::out | ios::binary);
```

When an `fstream` file is opened with both the `ios::in` and `ios::out` flags, the file's current contents are preserved, and the read/write position is initially placed at the beginning of the file. If the file does not exist, it is created.

Programs 12-20, 12-21, and 12-22 demonstrate many of the techniques we have discussed. Program 12-20 sets up a file with five blank inventory records. Each record is a structure with members for holding a part description, quantity on hand, and price. Program 12-21 displays the contents of the file on the screen. Program 12-22 opens the file in both input and output modes and allows the user to change the contents of a specific record.

Program 12-20

```
1  // This program sets up a file of blank inventory records.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  // Constants
7  const int DESC_SIZE = 31;   // Description size
8  const int NUM_RECORDS = 5;  // Number of records
9
10 // Declaration of InventoryItem structure
11 struct InventoryItem
12 {
13     char desc[DESC_SIZE];
14     int qty;
15     double price;
16 };
17
18 int main()
19 {
20     // Create an empty InventoryItem structure.
21     InventoryItem record = { "", 0, 0.0 };
22
23     // Open the file for binary output.
24     fstream inventory("Inventory.dat", ios::out | ios::binary);
25
26     // Write the blank records
27     for (int count = 0; count < NUM_RECORDS; count++)
28     {
29         cout << "Now writing record " << count << endl;
30         inventory.write(reinterpret_cast<char *>(&record),
31                         sizeof(record));
32     }
33
34     // Close the file.
35     inventory.close();
36     return 0;
37 }
```

Program Output

```
Now writing record 0
Now writing record 1
Now writing record 2
Now writing record 3
Now writing record 4
```

Program 12-21 simply displays the contents of the inventory file on the screen. It can be used to verify that Program 12-20 successfully created the blank records, and that Program 12-22 correctly modified the designated record.

Program 12-21

```
// This program displays the contents of the inventory file.
#include <iostream>
#include <fstream>
using namespace std;

const int DESC_SIZE = 31; // Description size

// Declaration of InventoryItem structure
struct InventoryItem
{
    char desc[DESC_SIZE];
    int qty;
    double price;
};

int main()
{
    InventoryItem record; // To hold an inventory record

    // Open the file for binary input.
    fstream inventory("Inventory.dat", ios::in | ios::binary);

    // Now read and display the records
    inventory.read(reinterpret_cast<char *>(&record),
                   sizeof(record));
    while (!inventory.eof())
    {
        cout << "Description: ";
        cout << record.desc << endl;
        cout << "Quantity: ";
        cout << record.qty << endl;
        cout << "Price: ";
        cout << record.price << endl << endl;
        inventory.read(reinterpret_cast<char *>(&record),
                       sizeof(record));
    }

    // Close the file.
    inventory.close();
    return 0;
}
```

Here is the screen output of Program 12-21 if it is run immediately after Program 12-20 sets up the file of blank records.

Program 12-21

Program Output

```
Description:
Quantity: 0
Price: 0.0
```

(program output continues)

Program 12-21 *(continued)*

```
Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0
```

Program 12-22 allows the user to change the contents of an individual record in the inventory file.

Program 12-22

```
1  // This program allows the user to edit a specific record.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  const int DESC_SIZE = 31; // Description size
7
8  // Declaration of InventoryItem structure
9  struct InventoryItem
10 {
11     char desc[DESC_SIZE];
12     int qty;
13     double price;
14 };
15
16 int main()
17 {
18     InventoryItem record;  // To hold an inventory record
19     long recNum;           // To hold a record number
20
21     // Open the file in binary mode for input and output
22     fstream inventory("Inventory.dat",
23                       ios::in | ios::out | ios::binary);
24
25     // Get the record number of the desired record.
26     cout << "Which record do you want to edit? ";
27     cin >> recNum;
28
29     // Move to the record and read it.
30     inventory.seekg(recNum * sizeof(record), ios::beg);
31     inventory.read(reinterpret_cast<char *>(&record),
32                    sizeof(record));
```

```
33
34      // Display the record contents.
35      cout << "Description: ";
36      cout << record.desc << endl;
37      cout << "Quantity: ";
38      cout << record.qty << endl;
39      cout << "Price: ";
40      cout << record.price << endl;
41
42      // Get the new record data.
43      cout << "Enter the new data:\n";
44      cout << "Description: ";
45      cin.ignore();
46      cin.getline(record.desc, DESC_SIZE);
47      cout << "Quantity: ";
48      cin >> record.qty;
49      cout << "Price: ";
50      cin >> record.price;
51
52      // Move back to the beginning of this record's position.
53      inventory.seekp(recNum * sizeof(record), ios::beg);
54
55      // Write the new record over the current record.
56      inventory.write(reinterpret_cast<char *>(&record),
57                      sizeof(record));
58
59      // Close the file.
60      inventory.close();
61      return 0;
62  }
```

Program Output with Example Input Shown in Bold

```
Which record do you want to edit? 2 [Enter]
Description:
Quantity: 0
Price: 0.0
Enter the new data:
Description: Wrench [Enter]
Quantity: 10 [Enter]
Price: 4.67 [Enter]
```

Checkpoint

12.10 Describe the difference between the `seekg` and the `seekp` functions.

12.11 Describe the difference between the `tellg` and the `tellp` functions.

12.12 Describe the meaning of the following file access flags:

```
ios::beg
ios::end
ios::cur
```

12.13 What is the number of the first byte in a file?

12.14 Briefly describe what each of the following statements does:

```
file.seekp(100L, ios::beg);
file.seekp(-10L, ios::end);
file.seekg(-25L, ios::cur);
file.seekg(30L, ios::cur);
```

12.15 Describe the mode that each of the following statements causes a file to be opened in:

```
file.open("info.dat", ios::in | ios::out);
file.open("info.dat", ios::in | ios::app);
file.open("info.dat", ios::in | ios::out | ios::ate);
file.open("info.dat", ios::in | ios::out | ios::binary);
```

For another example of this chapter's topics, see the High Adventure Travel Part 3 Case Study, available on the book's companion Web site at www.pearsonhighered.com/gaddis.

Review Questions and Exercises

Short Answer

1. What capability does the `fstream` data type provide that the `ifstream` and `ofstream` data types do not?
2. Which file access flag do you use to open a file when you want all output written to the end of the file's existing contents?
3. Assume that the file `data.txt` already exists, and the following statement executes. What happens to the file?

   ```
   fstream file("data.txt", ios::out);
   ```

4. How do you combine multiple file access flags when opening a file?
5. Should file stream objects be passed to functions by value or by reference? Why?
6. Under what circumstances is a file stream object's `ios::hardfail` bit set? What member function reports the state of this bit?
7. Under what circumstances is a file stream object's `ios::eofbit` bit set? What member function reports the state of this bit?
8. Under what circumstances is a file stream object's `ios::badbit` bit set? What member function reports the state of this bit?
9. How do you read the contents of a text file that contains whitespace characters as part of its data?
10. What arguments do you pass to a file stream object's `write` member function?
11. What arguments do you pass to a file stream object's `read` member function?
12. What type cast do you use to convert a pointer from one type to another?
13. What is the difference between the `seekg` and `seekp` member functions?
14. How do you get the byte number of a file's current read position? How do you get the byte number of a file's current write position?
15. If a program has read to the end of a file, what must you do before using either the `seekg` or `seekp` member functions?

16. How do you determine the number of bytes that a file contains?
17. How do you rewind a sequential-access file?

Fill-in-the-Blank

18. The _____ file stream data type is for output files, input files, or files that perform both input and output.
19. If a file fails to open, the file stream object will be set to _____.
20. The same formatting techniques used with _____ may also be used when writing data to a file.
21. The _____ function reads a line of text from a file.
22. The _____ member function reads a single character from a file.
23. The _____ member function writes a single character to a file.
24. _____ files contain data that is unformatted and not necessarily stored as ASCII text.
25. _____ files contain data formatted as _____.
26. A(n) _____ is a complete set of data about a single item and is made up of _____.
27. In C++, _____ provide a convenient way to organize data into fields and records.
28. The _____ member function writes "raw" binary data to a file.
29. The _____ member function reads "raw" binary data from a file.
30. The _____ operator is necessary if you pass anything other than a pointer-to-char as the first argument of the two functions mentioned in questions 26 and 27.
31. In _____ file access, the contents of the file are read in the order they appear in the file, from the file's start to its end.
32. In _____ file access, the contents of a file may be read in any order.
33. The _____ member function moves a file's read position to a specified byte in the file.
34. The _____ member function moves a file's write position to a specified byte in the file.
35. The _____ member function returns a file's current read position.
36. The _____ member function returns a file's current write position.
37. The _____ mode flag causes an offset to be calculated from the beginning of a file.
38. The _____ mode flag causes an offset to be calculated from the end of a file.
39. The _____ mode flag causes an offset to be calculated from the current position in the file.
40. A negative offset causes the file's read or write position to be moved _____ in the file from the position specified by the mode.

Algorithm Workbench

41. Write a statement that defines a file stream object named `places`. The object will be used for both output and input.

42. Write two statements that use a file stream object named `people` to open a file named `people.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for output.
43. Write two statements that use a file stream object named `pets` to open a file named `pets.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for input.
44. Write two statements that use a file stream object named `places` to open a file named `places.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for both input and output.
45. Write a program segment that defines a file stream object named `employees`. The file should be opened for both input and output (in binary mode). If the file fails to open, the program segment should display an error message.
46. Write code that opens the file `data.txt` for both input and output, but first determines if the file exists. If the file does not exist, the code should create it, then open it for both input and output.
47. Write code that determines the number of bytes contained in the file associated with the file stream object `dataFile`.
48. The `infoFile` file stream object is used to sequentially access data. The program has already read to the end of the file. Write code that rewinds the file.

True or False

49. T F Different operating systems have different rules for naming files.
50. T F `fstream` objects are only capable of performing file output operations.
51. T F `ofstream` objects, by default, delete the contents of a file if it already exists when opened.
52. T F `ifstream` objects, by default, create a file if it doesn't exist when opened.
53. T F Several file access flags may be joined by using the | operator.
54. T F A file may be opened in the definition of the file stream object.
55. T F If a file is opened in the definition of the file stream object, no mode flags may be specified.
56. T F A file stream object's `fail` member function may be used to determine if the file was successfully opened.
57. T F The same output formatting techniques used with `cout` may also be used with file stream objects.
58. T F The >> operator expects data to be delimited by whitespace characters.
59. T F The `getline` member function can be used to read text that contains whitespaces.
60. T F It is not possible to have more than one file open at once in a program.
61. T F Binary files contain unformatted data, not necessarily stored as text.
62. T F Binary is the default mode in which files are opened.
63. T F The `tellp` member function tells a file stream object which byte to move its write position to.
64. T F It is possible to open a file for both input and output.

Find the Error

Each of the following programs or program segments has errors. Find as many as you can.

65.
```
fstream file(ios::in | ios::out);
file.open("info.dat");
if (!file)
{
   cout << "Could not open file.\n";
}
```

66.
```
ofstream file;
file.open("info.dat", ios::in);
if (file)
{
   cout << "Could not open file.\n";
}
```

67.
```
fstream file("info.dat");
if (!file)
{
   cout << "Could not open file.\n";
}
```

68.
```
fstream dataFile("info.dat", ios:in | ios:binary);
int x = 5;
dataFile << x;
```

69.
```
fstream dataFile("info.dat", ios:in);
char stuff[81];
dataFile.get(stuff);
```

70.
```
fstream dataFile("info.dat", ios:in);
char stuff[81] = "abcdefghijklmnopqrstuvwxyz";
dataFile.put(stuff);
```

71.
```
fstream dataFile("info.dat", ios:out);
struct Date
{
   int month;
   int day;
   int year;
};
Date dt = { 4, 2, 98 };
dataFile.write(&dt, sizeof(int));
```

72.
```
fstream inFile("info.dat", ios:in);
int x;
inFile.seekp(5);
inFile >> x;
```

Programming Challenges

1. **File Head Program**

   Write a program that asks the user for the name of a file. The program should display the first 10 lines of the file on the screen (the "head" of the file). If the file has fewer

than 10 lines, the entire file should be displayed, with a message indicating the entire file has been displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

2. File Display Program

Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, and then pause. Each time the program pauses, it should wait for the user to strike a key before the next 24 lines are displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

3. Punch Line

Write a program that reads and prints a joke and its punch line from two different files. The first file contains a joke, but not its punch line. The second file has the punch line as its last line, preceded by "garbage." The `main` function of your program should open the two files and then call two functions, passing each one the file it needs. The first function should read and display each line in the file it is passed (the joke file). The second function should display only the last line of the file it is passed (the punch line file). It should find this line by seeking to the end of the file and then backing up to the beginning of the last line. Data to test your program can be found in the `joke.txt` and `punchline.txt` files.

4. Tail Program

Write a program that asks the user for the name of a file. The program should display the last 10 lines of the file on the screen (the "tail" of the file). If the file has fewer than 10 lines, the entire file should be displayed, with a message indicating the entire file has been displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

5. Line Numbers

(This assignment could be done as a modification of the program in Programming Challenge 2.) Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. Each line of screen output should be preceded with a line number, followed by a colon. The line numbering should start at 1. Here is an example:

```
1:George Rolland
2:127 Academy Street
3:Brasstown, NC  28706
```

If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, and then pause. Each time the program pauses, it should wait for the user to strike a key before the next 24 lines are displayed.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

6. String Search

Write a program that asks the user for a file name and a string to search for. The program should search the file for every occurrence of a specified string. When the string is found, the line that contains it should be displayed. After all the occurrences have been located, the program should report the number of times the string appeared in the file.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

7. Sentence Filter

Write a program that asks the user for two file names. The first file will be opened for input and the second file will be opened for output. (It will be assumed that the first file contains sentences that end with a period.) The program will read the contents of the first file and change all the letters to lowercase except the first letter of each sentence, which should be made uppercase. The revised contents should be stored in the second file.

NOTE: Using an editor, you should create a simple text file that can be used to test this program.

8. Array/File Functions

Write a function named `arrayToFile`. The function should accept three arguments: the name of a file, a pointer to an `int` array, and the size of the array. The function should open the specified file in binary mode, write the contents of the array to the file, and then close the file.

Write another function named `fileToArray`. This function should accept three arguments: the name of a file, a pointer to an `int` array, and the size of the array. The function should open the specified file in binary mode, read its contents into the array, and then close the file.

Write a complete program that demonstrates these functions by using the `arrayToFile` function to write an array to a file, and then using the `fileToArray` function to read the data from the same file. After the data are read from the file into the array, display the array's contents on the screen.

VideoNote
Solving the File Encryption Filter Problem

9. File Encryption Filter

File encryption is the science of writing the contents of a file in a secret code. Your encryption program should work like a filter, reading the contents of one file, modifying the data into a code, and then writing the coded contents out to a second file. The second file will be a version of the first file, but written in a secret code.

Although there are complex encryption techniques, you should come up with a simple one of your own. For example, you could read the first file one character at a time, and add 10 to the ASCII code of each character before it is written to the second file.

10. File Decryption Filter

Write a program that decrypts the file produced by the program in Programming Challenge 9. The decryption program should read the contents of the coded file, restore the data to its original state, and write it to another file.

11. Corporate Sales Data Output

Write a program that uses a structure to store the following data on a company division:

Division Name (such as East, West, North, or South)
Quarter (1, 2, 3, or 4)
Quarterly Sales

The user should be asked for the four quarters' sales figures for the East, West, North, and South divisions. The data for each quarter for each division should be written to a file.

Input Validation: Do not accept negative numbers for any sales figures.

12. Corporate Sales Data Input

Write a program that reads the data in the file created by the program in Programming Challenge 11. The program should calculate and display the following figures:

- Total corporate sales for each quarter
- Total yearly sales for each division
- Total yearly corporate sales
- Average quarterly sales for the divisions
- The highest and lowest quarters for the corporation

13. Inventory Program

Write a program that uses a structure to store the following inventory data in a file:

Item Description
Quantity on Hand
Wholesale Cost
Retail Cost
Date Added to Inventory

The program should have a menu that allows the user to perform the following tasks:

- Add new records to the file.
- Display any record in the file.
- Change any record in the file.

Input Validation: The program should not accept quantities, or wholesale or retail costs, less than 0. The program should not accept dates that the programmer determines are unreasonable.

14. Inventory Screen Report

Write a program that reads the data in the file created by the program in Programming Challenge 13. The program should calculate and display the following data:

- The total wholesale value of the inventory
- The total retail value of the inventory
- The total quantity of all items in the inventory

15. Average Number of Words

If you have downloaded this book's source code from the companion Web site, you will find a file named `text.txt` in the Chapter 12 folder. (The companion Web site is at www.pearsonhighered.com/gaddis.) The text that is in the file is stored as one sentence per line. Write a program that reads the file's contents and calculates the average number of words per sentence.

Group Project

16. Customer Accounts

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call other program functions. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so each student is given about the same workload.

Write a program that uses a structure to store the following data about a customer account:

Name
Address
City, State, and ZIP
Telephone Number
Account Balance
Date of Last Payment

The structure should be used to store customer account records in a file. The program should have a menu that lets the user perform the following operations:

- Enter new records into the file.
- Search for a particular customer's record and display it.
- Search for a particular customer's record and delete it.
- Search for a particular customer's record and change it.
- Display the contents of the entire file.

Input Validation: When the data for a new account is entered, be sure the user enters data for all the fields. No negative account balances should be entered.