Chapter 12 Advanced File Operations

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.

- Most real-world programs utilize files for data storage and retrieval.
- Common examples include:
  - **Word Processors:** Save documents to files for later editing.
  - **Database Management Systems (DBMSs):** Manage large data collections like payroll and customer records in files.
  - **Spreadsheets:** Store numerical data and formulas in files.
  - **Compilers:** Translate source code from a file into an executable file.
- This chapter expands on simple file operations, focusing on the `fstream` data type.
- The `ifstream`, `ofstream`, and `fstream` data types all require the `<fstream>` header file.

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 Types

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.

DATA TYPE	DESCRIPTION
`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

- An `fstream` object is defined like any other data type object.

```
fstream dataFile;
```

- The `open` function is used to open a file and requires two arguments:
  - **File Name:** A string with the name of the file.
  - **File Access Flag:** Specifies the mode for opening the file.

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

- The `ios::out` flag opens a file in **output mode**, allowing data to be written to it.
- The `ios::in` flag opens a file in **input mode**, allowing data to be read from it.

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

- See Table 12-2 for a list of file access flags.

Table 12-2 File Access Flags

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.

FILE ACCESS FLAG	MEANING
`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`.

- Multiple flags can be combined using the `|` operator.

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

- The combination `ios::in | ios::out` opens the file for both reading and writing.

> **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 combination `ios::out | ios::app` opens the file for output, and new data is appended to the end of the file.

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

- Various combinations of access flags allow for multiple file-opening modes.

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

Program 12-1

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

int main()
{
    fstream dataFile;

    cout << "Opening file...\n";
    dataFile.open("demofile.txt", ios::out);
    cout << "Now writing data to the file.\n";
    dataFile << "Jones\n";
    dataFile << "Smith\n";
    dataFile << "Willis\n";
    dataFile << "Davis\n";
    dataFile.close();
    cout << "Done.\n";
    return 0;
}
```

Program Output

Output to File demofile.txt

```
Jones
Smith
Willis
Davis
```

- The `\n` character creates new lines in the file's visual representation.
- All characters are written sequentially in the order they are sent.
- An **end-of-file marker**, a nonprinting character, is automatically added when the file is closed.

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   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main()
6   {
7       ofstream dataFile;
8
9       cout << "Opening file...\n";
10      dataFile.open("demofile.txt", ios::out);
11      cout << "Now writing data to the file.\n";
12      dataFile << "Jones\n";
13      dataFile << "Smith\n";
14      cout << "Now closing the file.\n";
15      dataFile.close();
16
17      cout << "Opening the file again...\n";
18      dataFile.open("demofile.txt", ios::out | ios::app);
19      cout << "Writing more data to the file.\n";
20      dataFile << "Willis\n";
21      dataFile << "Davis\n";
22      cout << "Now closing the file.\n";
23      dataFile.close();
24
25      cout << "Done.\n";
26      return 0;
27  }
```

Output to File demofile.txt

```
Jones
Smith
Willis
Davis
```

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

- After the file is closed, an end-of-file character is added. Reopening the file in append mode adds new data to the end, as shown in 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

- `ifstream` and `ofstream` objects have default modes for opening files.
- These modes define which operations are allowed and what happens if the file already exists.

Table 12-3 Default Open Mode

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.

- While `ifstream` is read-only and `ofstream` is write-only, their behavior can be modified by providing optional file access flags in the `open` function.

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

- For example, using `ios::app` with an `ofstream` object appends data instead of overwriting the file.

Checking for a File's Existence Before Opening It

- To avoid overwriting an existing file, you can check for its existence first.
- Attempt to open the file for input using `ios::in`.
- If the `open` operation fails (checked with `dataFile.fail()`), the file doesn't exist and can be safely created by opening it for output (`ios::out`).
- If it succeeds, the file already exists, and you can close it and inform the user.

```
fstream dataFile;
dataFile.open("values.txt", ios::in);
if (dataFile.fail())
{
   dataFile.open("values.txt", ios::out);
}
else
{
   dataFile.close();
   cout << "The file values.txt already exists.\n";
}
```

Opening a File with the File Stream Object Definition Statement

- Files can be opened directly within the file stream object's definition statement, bypassing a separate `open` call.

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

- This is useful when the file name and access mode are known at compile time.
- This method works for `fstream`, `ifstream`, and `ofstream` objects.

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

- Error checking can still be performed after opening a file this way.

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

12.4 Assuming `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 `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.

- Formatting techniques used with `cout` (from Chapter 3) also apply to file stream objects.
- Manipulators like `setprecision` and `fixed` can be used to format floating-point numbers written to a file, as shown in Program 12-3.

Program 12-3

```
1   #include <iostream>
2   #include <iomanip>
3   #include <fstream>
4   using namespace std;
5
6   int main()
7   {
```

```
 8       fstream dataFile;
 9       double num = 17.816392;
10
11       dataFile.open("numfile.txt", ios::out);
12
13       dataFile << fixed;
14       dataFile << num << endl;
15
16       dataFile << setprecision(4);
17       dataFile << num << endl;
18
19       dataFile << setprecision(3);
20       dataFile << num << endl;
21
22       dataFile << setprecision(2);
23       dataFile << num << endl;
24
25       dataFile << setprecision(1);
26       dataFile << num << endl;
27
28       cout << "Done.\n";
29       dataFile.close();
30       return 0;
31   }
```

Contents of File numfile.txt

```
17.816392
17.8164
17.816
17.82
17.8
```

- The `setw` stream manipulator can format file output into columns, just as with screen output. This is demonstrated in Program 12-4.

Program 12-4

```
1   #include <iostream>
2   #include <fstream>
3   #include <iomanip>
4   using namespace std;
5
```

```
 6   int main()
 7   {
 8       const int ROWS = 3;
 9       const int COLS = 3;
10       int nums[ROWS][COLS] = { 2897, 5, 837,
11                                34, 7, 1623,
12                                390, 3456, 12 };
13       fstream outFile("table.txt", ios::out);
14
15       for (int row = 0; row < ROWS; row++)
16       {
17           for (int col = 0; col < COLS; col++)
18           {
19               outFile << setw(8) << nums[row][col];
20           }
21           outFile << endl;
22       }
23       outFile.close();
24       cout << "Done.\n";
25       return 0;
26   }
```

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.

12.3 Passing File Stream Objects to Functions

Concept:

File stream objects may be passed by reference to functions.

- To create modular code for file handling, file stream objects can be passed to functions.
- **Crucially, they must always be passed by reference.**

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;
}
```

- Passing by reference is necessary because file stream operations change the object's internal state. This ensures consistency.
- Program 12-5 provides an example of passing file stream objects to functions.

Program 12-5

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   using namespace std;
5
6   bool openFileIn(fstream &, string);
7   void showContents(fstream &);
8
9   int main()
10  {
11      fstream dataFile;
12
13      if (openFileIn(dataFile, "demofile.txt"))
14      {
15          cout << "File opened successfully.\n";
16          cout << "Now reading data from the file.\n\n";
17          showContents(dataFile);
18          dataFile.close();
19          cout << "\nDone.\n";
20      }
21      else
22          cout << "File open error!" << endl;
23
24      return 0;
25  }
```

```
26
27
28  bool openFileIn(fstream &file, string name)
29  {
30      file.open(name, ios::in);
31      if (file.fail())
32          return false;
33      else
34          return true;
35  }
36
37
38  void showContents(fstream &file)
39  {
40      string line;
41
42      while (file >> line)
43      {
44          cout << line << endl;
45      }
46  }
```

Program Output

12.4 More Detailed Error Testing

Concept:

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

- All stream objects have a set of bit flags that signal the current stream state.
- See Table 12-4 for a list of these state bits.

Table 12-4 Stream Bits

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.

- Member functions, listed in Table 12-5, can be used to test these bits.
- The `clear()` function can be used to reset these state flags.

Table 12-5 Functions to Test the State of Stream Bits

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.

FUNCTION	DESCRIPTION
`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();
}
```

- Program 12-6 demonstrates how to use these error-testing functions to check a file stream's status after different operations, such as a successful write and a failed read attempt.

Program 12-6

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   void showState(fstream &);
6
7   int main()
8   {
9       int num = 10;
10
11      fstream testFile("stuff.dat", ios::out);
12      if (testFile.fail())
13      {
14          cout << "ERROR: Cannot open the file.\n";
15          return 0;
16      }
```

```
17
18      cout << "Writing the value " << num << " to the file.\n";
19      testFile << num;
20
21      showState(testFile);
22
23      testFile.close();
24
25      testFile.open("stuff.dat", ios::in);
26      if (testFile.fail())
27      {
28          cout << "ERROR: Cannot open the file.\n";
29          return 0;
30      }
31
32      cout << "Reading from the file.\n";
33      testFile >> num;
34      cout << "The value " << num << " was read.\n";
35
36      showState(testFile);
37
38      cout << "Forcing a bad read operation.\n";
39      testFile >> num;
40
41      showState(testFile);
42
43      testFile.close();
44      return 0;
45  }
46
47
48  void showState(fstream &file)
49  {
50      cout << "File Status:\n";
51      cout << "  eof bit: " << file.eof() << endl;
52      cout << "  fail bit: " << file.fail() << endl;
53      cout << "  bad bit: " << file.bad() << endl;
54      cout << "  good bit: " << file.good() << endl;
55      file.clear();
56  }
```

Program Output

12.5 Member Functions for Reading and Writing Files

Concept:

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

- Reading a file with the `>>` operator can be problematic if the data contains whitespace.
- The `>>` operator uses whitespace as a delimiter and therefore skips over it instead of reading it.
- For example, consider the file murphy.txt, which contains the following data:
- Jayne Murphy
- 47 Jones Circle
- Almond, NC 28702
- Program 12-7 shows how this can lead to incorrectly reading formatted data, like addresses, from a file.

Program 12-7

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

int main()
{
    string input;
    fstream nameFile;

    nameFile.open("murphy.txt", ios::in);

    if (nameFile)
    {
        while (nameFile >> input)
        {
            cout << input;
        }

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

💻 **Program Output**

The `getline` Function

- The `getline` function solves the whitespace problem by reading an entire line of data, including spaces.

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

- The function takes three arguments:
  - **File Stream Object:** The source from which to read (e.g., `dataFile`).
  - **String Object:** The destination to store the data (e.g., `str`).

- **Delimiter Character:** (Optional) The character that stops the reading process. The default is the newline character `\n`.

`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 `getline` function reads until it encounters the delimiter, then stores the read characters in the string object.
- Program 12-8 modifies Program 12-7 to use `getline`, correctly reading each line from the file.

Program 12-8

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   using namespace std;
5
6   int main()
7   {
8       string input;
9       fstream nameFile;
10
11      nameFile.open("murphy.txt", ios::in);
12
13      if (nameFile)
14      {
15          getline(nameFile, input);
16
17          while (nameFile)
18          {
```

```
            cout << input << endl;

            getline(nameFile, input);
        }

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

Program Output

- You can specify a custom delimiter. This is useful for parsing files where data fields are separated by a special character.
- For instance, a file might use `$` to separate fields and `\n` to separate records.

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$
```

- A **record** is a complete set of data for one item, while a **field** is a single piece of data within that record.
- Program 12-9 shows how to use `getline` with a custom `$` delimiter to read fields from a file.

Program 12-9

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

```
 5
 6   int main()
 7   {
 8       string input;
 9
10       fstream dataFile("names2.txt", ios::in);
11
12       if (dataFile)
13       {
14           getline(dataFile, input, '$');
15
16           while (dataFile)
17           {
18               cout << input << endl;
19
20               getline(dataFile, input, '$');
21           }
22
23           dataFile.close();
24       }
25       else
26       {
27           cout << "ERROR: Cannot open file.\n";
28       }
29       return 0;
30   }
```

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
```

- Using a custom delimiter like `$` will read the `\n` characters as part of the data, which can result in extra blank lines in the output.

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 `get` member function reads a single character from a file, including whitespace.

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

- In the example above, a single character is read from the `inFile` stream and stored in the `char` variable `ch`.
- Program 12-10 demonstrates using `get` in a loop to read and display the entire contents of a file character by character.

Program 12-10

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   using namespace std;
5
6   int main()
7   {
8       string fileName;
9       char ch;
10      fstream file;
11
12      cout << "Enter a file name: ";
13      cin >> fileName;
14
15      file.open(filename, ios::in);
16
17      if (file)
18      {
19          file.get(ch);
20
21          while (file)
```

```
        {
            cout << ch;

            file.get(ch);
        }

        file.close();
    }
    else
        cout << fileName << " could not be opened.\n";
    return 0;
}
```

The `put` Member Function

- The `put` member function writes a single character to a file.

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

- This statement writes the character stored in the `char` variable `ch` to the file associated with `outFile`.
- Program 12-11 shows how to use `put` to write a user-entered sentence to a file one character at a time.

Program 12-11

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

int main()
{
    char ch;

    fstream dataFile("sentence.txt", ios::out);

    cout << "Type a sentence and be sure to end it with a ";
    cout << "period.\n";

    cin.get(ch);
    while (ch != '.')
```

```
16        {
17            dataFile.put(ch);
18            cin.get(ch);
19        }
20        dataFile.put(ch);
21
22        dataFile.close();
23        return 0;
24    }
```

Program Output

Checkpoint

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

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.

Working with Multiple Files

- Programs can have multiple files open simultaneously.
- Real-world applications often use multiple files to categorize data for a single item, such as in a payroll system.

`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.

- To open multiple files, define a separate file stream object for each one.

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

- A common task is opening one file for input and another for output.
- A program that reads data from one file, modifies it, and writes it to another file is known as a **filter**.
- Program 12-12 demonstrates this by reading a file, converting its contents to uppercase, and writing the result to a new file.

Program 12-12

```
 1   #include <iostream>
 2   #include <fstream>
 3   #include <string>
 4   #include <cctype>
 5   using namespace std;
 6
 7   int main()
 8   {
 9       string fileName;
10       char ch;
11       ifstream inFile;
12
13       ofstream outFile("out.txt");
14
15       cout << "Enter a file name: ";
16       cin >> fileName;
17
18       inFile.open(filename);
19
```

```
20        if (inFile)
21        {
22            inFile.get(ch);
23
24            while (inFile)
25            {
26                outFile.put(toupper(ch));
27
28                inFile.get(ch);
29            }
30
31            inFile.close();
32            outFile.close();
33            cout << "File conversion done.\n";
34        }
35        else
36            cout << "Cannot open " << fileName << endl;
37        return 0;
38    }
```

💻 **Program Output**

12.7 Binary Files

Concept:

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

- Files can be categorized as **text files** or **binary files**.
- In text files, all data, including numbers, is stored as human-readable ASCII text. For example, the number `1297` is stored as the characters '1', '2', '9', '7'.

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

- In memory, data is stored in its “raw” binary format, not as text.
- Binary files store data in this same raw, binary format, exactly as it appears in memory.
- To work with a binary file, open it using the `ios::binary` flag, often combined with other flags like `ios::out`.

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

> **Note:**
> By default, files are opened in text mode.

The `write` and `read` Member Functions

- The `write` member function is used to write binary data to a file.
- Its format is `fileObject.write(address, size)`.
  - `address`: The starting memory address of the data to be written (expected to be a `char*`).
  - `size`: The number of bytes to write.
- The `read` member function reads binary data from a file into memory.
- Its format is `fileObject.read(address, size)`.
  - `address`: The starting memory address where the read data will be stored (expected to be a `char*`).
  - `size`: The number of bytes to read.

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

- The following code writes an entire `char` array to a binary file.

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

- The following code uses the `read` function to read a single character from a binary file.

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

- The following code reads enough data from a binary file to fill an entire `char` array.

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

- Program 12-13 shows how to write a `char` array to a binary file and then read it back.

Program 12-13

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main()
6   {
7       const int SIZE = 4;
8       char data[SIZE] = {'A', 'B', 'C', 'D'};
9       fstream file;
10
11      file.open("test.dat", ios::out | ios::binary);
12
13      cout << "Writing the characters to the file.\n";
14      file.write(data, sizeof(data));
15
16      file.close();
17
18      file.open("test.dat", ios::in | ios::binary);
19
20      cout << "Now reading the data back into memory.\n";
21      file.read(data, sizeof(data));
22
23      for (int count = 0; count < SIZE; count++)
24          cout << data[count] << " ";
25      cout << endl;
```

```
26
27        file.close();
28        return 0;
29    }
```

Program Output

Writing Data other than `char` to Binary Files

- The `write` and `read` functions require their first argument to be a pointer to a `char`.
- To work with other data types (like `int`, `double`, etc.), you must cast the pointer to the data's address.
- The `reinterpret_cast<char *>(address)` is used to perform this conversion. Its general format is:

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

- 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));
```

- 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));
```

- 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 and reading an `int` array to and from a binary file using `reinterpret_cast`.

Program 12-14

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

int main()
{
    const int SIZE = 10;
    fstream file;
    int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

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

    cout << "Writing the data to the file.\n";
    file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));

    file.close();

    file.open("numbers.dat", ios::in | ios::binary);

    cout << "Now reading the data back into memory.\n";
    file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));

    for (int count = 0; count < SIZE; count++)
        cout << numbers[count] << " ";
    cout << endl;

    file.close();
    return 0;
}
```

Program Output

12.8 Creating Records with Structures

Concept:

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

- A **field** is a single piece of data; a **record** is a complete set of fields about one item.
- Structures are an excellent way to organize fields into a record.

```
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];
};
```

- An entire structure variable can be written to a file in a single operation using the `write` function.

```
Info person;
```

```
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

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

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];
};

int main()
{
    Info person;
    char again;

    fstream people("people.dat", ios::out | ios::binary);

    do
    {
        cout << "Enter the following data about a "
             << "person:\n";
        cout << "Name: ";
        cin.getline(person.name, NAME_SIZE);
        cout << "Age: ";
        cin >> person.age;
        cin.ignore();
        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);

```

```
39          people.write(reinterpret_cast<char *>(&person),
40                       sizeof(person));
41
42          cout << "Do you want to enter another record? ";
43          cin >> again;
44          cin.ignore();
45       } while (again == 'Y' || again == 'y');
46
47       people.close();
48       return 0;
49    }
```

Program Output

- Program 12-15 allows you to build a file by filling the members of the `person` variable, 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   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
6
7   struct Info
8   {
9       char name[NAME_SIZE];
```

```
10      int age;
11      char address1[ADDR_SIZE];
12      char address2[ADDR_SIZE];
13      char phone[PHONE_SIZE];
14  };
15
16  int main()
17  {
18      Info person;
19      char again;
20      fstream people;
21
22      people.open("people.dat", ios::in | ios::binary);
23
24      if (!people)
25      {
26          cout << "Error opening file. Program aborting.\n";
27          return 0;
28      }
29
30      cout << "Here are the people in the file:\n\n";
31      people.read(reinterpret_cast<char *>(&person),
32                  sizeof(person));
33
34      while (!people.eof())
35      {
36          cout << "Name: ";
37          cout << person.name << endl;
38          cout << "Age: ";
39          cout << person.age << endl;
40          cout << "Address line 1: ";
41          cout << person.address1 << endl;
42          cout << "Address line 2: ";
43          cout << person.address2 << endl;
44          cout << "Phone: ";
45          cout << person.phone << endl;
46
47          cout << "\nPress the Enter key to see the next record.\n";
48          cin.get(again);
49
50          people.read(reinterpret_cast<char *>(&person),
51                      sizeof(person));
52      }
53
```

```
54        cout << "That's all the data in the file!\n";
55        people.close();
56        return 0;
57    }
```

💻 **Program Output** 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.

- **Sequential access** involves reading or writing data from the beginning of a file to the end, one byte after another. To access data in the middle, you must first process everything before it.
- **Random access** allows a program to jump directly to any byte in the file without processing the preceding bytes.
- This is analogous to the difference between a cassette tape (sequential) and a CD (random access).

The `seekp` and `seekg` Member Functions

- The `seekp` and `seekg` member functions are used to move the read/write position within a file.
- `seekp` ("seek put") is used for output files.
- `seekg` ("seek get") is used for input files.

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

- The functions take two arguments: a `long` integer offset and a mode flag.
- The **offset** specifies how many bytes to move. It can be positive (forward) or negative (backward).
- The **mode** specifies the starting point for the offset calculation:

Table 12-6 Offset Modes

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 Mode Examples

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.

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

```
abcdefghijklmnopqrstuvwxyz
```

- Program 12-17 demonstrates using `seekg` to jump to different locations in a text file.

Program 12-17

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main()
6   {
7       char ch;
8
9       fstream file("letters.txt", ios::in);
10
11      file.seekg(5L, ios::beg);
```

```
12        file.get(ch);
13        cout << "Byte 5 from beginning: " << ch << endl;
14
15        file.seekg(-10L, ios::end);
16        file.get(ch);
17        cout << "10th byte from end: " << ch << endl;
18
19        file.seekg(3L, ios::cur);
20        file.get(ch);
21        cout << "Byte 3 from current: " << ch << endl;
22
23        file.close();
24        return 0;
25    }
```

Program Screen Output

- Program 12-18 shows how to use `seekg` to randomly access and display specific records in a binary file of structures.

Program 12-18

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
6
7   struct Info
8   {
9       char name[NAME_SIZE];
10      int age;
11      char address1[ADDR_SIZE];
12      char address2[ADDR_SIZE];
13      char phone[PHONE_SIZE];
14  };
15
16  long byteNum(int);
17  void showRec(Info);
18
```

```
19  int main()
20  {
21      Info person;
22      fstream people;
23
24      people.open("people.dat", ios::in | ios::binary);
25
26      if (!people)
27      {
28          cout << "Error opening file. Program aborting.\n";
29          return 0;
30      }
31
32      cout << "Here is record 1:\n";
33      people.seekg(byteNum(1), ios::beg);
34      people.read(reinterpret_cast<char *>(&person), sizeof(person));
35      showRec(person);
36
37      cout << "\nHere is record 0:\n";
38      people.seekg(byteNum(0), ios::beg);
39      people.read(reinterpret_cast<char *>(&person), sizeof(person));
40      showRec(person);
41
42      people.close();
43      return 0;
44  }
45
46
47  long byteNum(int recNum)
48  {
49      return sizeof(Info) * recNum;
50  }
51
52
53  void showRec(Info record)
54  {
55      cout << "Name: ";
56      cout << record.name << endl;
57      cout << "Age: ";
58      cout << record.age << endl;
59      cout << "Address line 1: ";
60      cout << record.address1 << endl;
61      cout << "Address line 2: ";
62      cout << record.address2 << endl;
```

```
63        cout << "Phone: ";
64        cout << record.phone << endl;
65    }
```

💻 **Program Output** 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

- The `tellp` and `tellg` functions return the current byte number of the file's read/write position.
- `tellp` returns the write position.
- `tellg` returns the read position.

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

- These functions can be used to determine the total size of a file by seeking to the end (`file.seekg(0L, ios::end)`) and then getting the position (`file.tellg()`).

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

- Program 12-19 demonstrates using `tellg` to show the current position and file size. The file `letters.txt` contains:

```
abcdefghijklmnopqrstuvwxyz
```

Program 12-19

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main()
6   {
7       long offset;
8       long numBytes;
9       char ch;
10      char again;
11
12      fstream file("letters.txt", ios::in);
13
14      file.seekg(0L, ios::end);
15      numBytes = file.tellg();
16      cout << "The file has " << numBytes << " bytes.\n";
17
18      file.seekg(0L, ios::beg);
19
20      do
21      {
22          cout << "Currently at position " << file.tellg() << endl;
23
24          cout << "Enter an offset from the beginning of the file: ";
25          cin >> offset;
26
27          if (offset >= numBytes)
28              cout << "Cannot read past the end of the file.\n";
29          else
30          {
31              file.seekg(offset, ios::beg);
```

```
            file.get(ch);
            cout << "Character read: " << ch << endl;
        }

        cout << "Do it again? ";
        cin >> again;
    } while (again == 'Y' || again == 'y');

    file.close();
    return 0;
}
```

Program Output

Rewinding a Sequential-Access File with `seekg`

- To process a file's contents more than once, you can either close and reopen it or "rewind" it.

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

- Rewinding moves the read position back to the beginning of the file without closing it.
- This is done by calling `dataFile.seekg(0L, ios::beg)`.

- If you have already read to the end of the file, you must call `dataFile.clear()` first to clear the `eof` flag.

```
dataFile.open("file.txt", ios::in);
dataFile.clear();
dataFile.seekg(0L, ios::beg);
dataFile.close();
```

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.

- An `fstream` object can be opened for both input and output simultaneously.
- This is done by combining the `ios::in` and `ios::out` file access flags with the `|` operator.

```
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 needed.

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

- When opened this way, the file's existing contents are preserved, and the initial read/write position is at the beginning. If the file doesn't exist, it is created.
- This is useful for applications that need to find a record and then modify it in place.
- Program 12-20 creates a file of blank inventory records.
- Program 12-21 reads and displays the records from the file.
- Program 12-22 opens the file for both input and output, allowing the user to seek to a specific record, display it, get new data, and overwrite the old record.

Program 12-20

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

const int DESC_SIZE = 31;
const int NUM_RECORDS = 5;

struct InventoryItem
{
    char desc[DESC_SIZE];
    int qty;
    double price;
};

int main()
{
    InventoryItem record = { "", 0, 0.0 };

    fstream inventory("Inventory.dat", ios::out | ios::binary);

    for (int count = 0; count < NUM_RECORDS; count++)
    {
        cout << "Now writing record " << count << endl;
        inventory.write(reinterpret_cast<char *>(&record),
                        sizeof(record));
    }

    inventory.close();
    return 0;
}
```

Program Output

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

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

const int DESC_SIZE = 31;

struct InventoryItem
{
    char desc[DESC_SIZE];
    int qty;
    double price;
};

int main()
{
    InventoryItem record;

    fstream inventory("Inventory.dat", ios::in | ios::binary);

    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));
    }

    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

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

Program 12-22

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   const int DESC_SIZE = 31;
6
7   struct InventoryItem
8   {
9       char desc[DESC_SIZE];
10      int qty;
11      double price;
12  };
13
14  int main()
15  {
16      InventoryItem record;
17      long recNum;
18
19      fstream inventory("Inventory.dat",
```

```
                        ios::in | ios::out | ios::binary);

      cout << "Which record do you want to edit? ";
      cin >> recNum;

      inventory.seekg(recNum * sizeof(record), ios::beg);
      inventory.read(reinterpret_cast<char *>(&record),
                     sizeof(record));

      cout << "Description: ";
      cout << record.desc << endl;
      cout << "Quantity: ";
      cout << record.qty << endl;
      cout << "Price: ";
      cout << record.price << endl;

      cout << "Enter the new data:\n";
      cout << "Description: ";
      cin.ignore();
      cin.getline(record.desc, DESC_SIZE);
      cout << "Quantity: ";
      cin >> record.qty;
      cout << "Price: ";
      cin >> record.price;

      inventory.seekp(recNum * sizeof(record), ios::beg);

      inventory.write(reinterpret_cast<char *>(&record),
                      sizeof(record));

      inventory.close();
      return 0;
   }
```

Program Output

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);
```

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 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 function?
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

1. The _____ file stream data type is for output files, input files, or files that perform both input and output.
2. If a file fails to open, the file stream object will be set to _____.
3. The same formatting techniques used with _____ may also be used when writing data to a file.

4. The _____ function reads a line of text from a file.
5. The _____ member function reads a single character from a file.
6. The _____ member function writes a single character to a file.
7. _____ files contain data that is unformatted and not necessarily stored as ASCII text.
8. _____ files contain data formatted as _____.
9. A(n) _____ is a complete set of data about a single item and is made up of _____.
10. In C++, _____ provide a convenient way to organize data into fields and records.
11. The _____ member function writes "raw" binary data to a file.
12. The _____ member function reads "raw" binary data from a file.
13. 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.
14. 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.
15. In _____ file access, the contents of a file may be read in any order.
16. The _____ member function moves a file's read position to a specified byte in the file.
17. The _____ member function moves a file's write position to a specified byte in the file.
18. The _____ member function returns a file's current read position.
19. The _____ member function returns a file's current write position.
20. The _____ mode flag causes an offset to be calculated from the beginning of a file.
21. The _____ mode flag causes an offset to be calculated from the end of a file.

22. The _____ mode flag causes an offset to be calculated from the current position in the file.

23. 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

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

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. Write code that determines the number of bytes contained in the file associated with the file stream object `dataFile`.

8. 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

1. T F Different operating systems have different rules for naming files.

2. T F `fstream` objects are only capable of performing file output operations.

3. T F `ofstream` objects, by default, delete the contents of a file if it already exists when opened.
4. T F `ifstream` objects, by default, create a file if it doesn't exist when opened.
5. T F Several file access flags may be joined by using the | operator.
6. T F A file may be opened in the definition of the file stream object.
7. T F If a file is opened in the definition of the file stream object, no mode flags may be specified.
8. T F A file stream object's `fail` member function may be used to determine if the file was successfully opened.
9. T F The same output formatting techniques used with `cout` may also be used with file stream objects.
10. T F The `>>` operator expects data to be delimited by whitespace characters.
11. T F The `getline` member function can be used to read text that contains whitespaces.
12. T F It is not possible to have more than one file open at once in a program.
13. T F Binary files contain unformatted data, not necessarily stored as text.
14. T F Binary is the default mode in which files are opened.
15. T F The `tellp` member function tells a file stream object which byte to move its write position to.
16. 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.

1.
```
fstream file(ios::in | ios::out);
file.open("info.dat");
if (!file)
{
    cout << "Could not open file.\n";
}
```

2.
```
ofstream file;
file.open("info.dat", ios::in);
if (file)
{
    cout << "Could not open file.\n";
}
```

3.
```
fstream file("info.dat");
if (!file)
{
    cout << "Could not open file.\n";
}
```

4.
```
fstream dataFile("info.dat", ios:in | ios:binary);
int x = 5;
dataFile << x;
```

5.
```
fstream dataFile("info.dat", ios:in);
char stuff[81];
dataFile.get(stuff);
```

6.
```
fstream dataFile("info.dat", ios:in);
char stuff[81] = "abcdefghijklmnopqrstuvwxyz";
dataFile.put(stuff);
```

7.
```
fstream dataFile("info.dat", ios:out);
struct Date
{
   int month;
   int day;
   int year;
};
Date dt = { 4, 2, 98 };
dataFile.write(&dt, sizeof(int));
```

8.
```
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 ten lines of the file on the screen (the "head" of the file). If the file has fewer than ten 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, 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 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 ten lines of the file on the screen (the "tail" of the file). If the file has fewer than ten 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 for which to search. 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 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, 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.

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, 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.

   

   Solving the File Encryption Filter Problem

   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, you will find a file named text.txt in the Chapter 12 folder. 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

1. 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 that 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.