11 Data File Basics

OBJECTIVES

- *Understand the uses for data files.*
- *Understand the difference between sequential-access and random-access data files.*
- *Open and close data files.*
- *Write to data files.*
- *Read from data files.*
- *Add to the end of a data file.*
- *Detect the end of a file.*
- *Use multiple data files at the same time.*
- *Prompt the user for file names.*

Overview

Many useful programs collect all input from the keyboard and print all output to the screen. However, the ability to input data from a disk file and send output to a disk file opens the door to many more possibilities. A program that organizes names and addresses, for example, must store the data somewhere other than RAM. Otherwise, the data is lost when the program ends.

In this chapter, you will learn about sequential-access and random-access data files. You will learn how to open and close a sequential-access file, how to write data to a file, how to read data from a file, and how to add data to the end of an existing file. You will also learn how to detect the end of a file, use multiple files at the same time, and prompt the user for file names.

CHAPTER 11, SECTION 1

File Concepts

Data files are not difficult to understand. Storing data in a file simply involves taking data from your computer's RAM and copying it to a disk. Retrieving data from a file is just the opposite. Data stored on disk is copied into variables or other data structures in RAM.

WHY USE DATA FILES?

Recall that your computer's memory (RAM) holds data only as long as the computer is on. Furthermore, data in RAM is lost as soon as your program ends. Disks and other forms of secondary storage hold data even after the computer is turned off. Therefore, any data that your program needs again should be stored on disk so that it can be reloaded into your program.

For example, suppose you have a program that prints mailing addresses on labels and envelopes. Unless a data file is used to store the addresses, the user has to enter the data from the keyboard every time the program runs.

Another reason to use data files is the amount of space available on a disk as compared to RAM. In the example of the program that prints addresses, a data file could store hundreds or even thousands of addresses—many times the number that could fit in RAM.

SEQUENTIAL-ACCESS VS. RANDOM-ACCESS FILES

There are two types of data files: sequential access and random access. The difference between the two is in how they access the data that is stored in them.

SEQUENTIAL-ACCESS DATA FILES

A ***sequential-access file*** works like an audio cassette tape. When you put a cassette tape in a stereo, you must fast forward or rewind to get to a specific song. You must move through the songs *sequentially* until you reach the one you

want to hear. Data stored in a sequential-access data file is placed one after the other like songs on a cassette tape. To retrieve specific data from a sequential-access data file, you must start at the beginning of the file and search for the data or records you want while moving through the file.

Sequential-access files are the most widely used data files. Word processors save documents in sequential-access files. Spreadsheets, graphic files, and some databases are stored as sequential-access files.

A sequential-access file can contain data of mixed types and sizes. For example, a word processor file may begin with general information about the document, followed by the text of the document itself. There may even be codes that control formatting mixed in with the document's text. The programmers that developed the word processor program place data in the file using their own rules. When the program loads the document from disk, those same rules are followed in order to correctly interpret the data file.

Figure 11-1 represents a sequential-access file storing a list of names. Notice that the names vary in length. To find the fourth name in the list (Beau Chenoweth), the three names that precede it must be read first.

FIGURE 11-1
A sequential-access file requires that data be read from the beginning of the file each time it is accessed.

Shelley Neff	James MacCloskey	Kim Fan	Beau Chenoweth	Sarah Boyd	Britney Sooter

RANDOM-ACCESS DATA FILES

A ***random-access file*** works like an audio compact disc (CD). With the touch of a button, you can immediately access any song on the CD. You can play the songs on a CD in any order, regardless of the order they appear on the CD. A random-access data file allows to you move directly to any data in the file.

Random-access files are most often used to store databases. A database with a large number of records is more efficiently managed with a random-access file because you can move quickly to any desired record in the database.

The secret to a random-access file is that data is written to the file in blocks (records) of equal size. In reality, the file appears on the disk as a sequential-access file. Because the file is made up of equally-sized blocks, a program can predict how far to move forward from the beginning of the file in order to get to the desired data.

Figure 11-2 represents a random-access file. Regardless of the length of the name, the same amount of disk space is occupied by the data. While the random-access file allows almost instant access to any data in the file, the disadvantage is that random-access files often occupy more disk space than sequential-access files.

In this book, you will write programs that use only sequential-access data files. However, the concept of a random-access file is important.

FIGURE 11-2
A random-access file allows any data to be accessed directly.

Shelley Neff	James MacCloskey	Kim Fan	Beau Chenowe

On the Net

Many modern operating systems include a file manager or their own set of file handling functions that programmers use to create consistency among programs. For more information on the resources operating systems provide for handling files, see http://www.ProgramCPP.com. See topic 11.1.1.

SECTION 11.1 QUESTIONS

1. List three reasons to use data files.
2. What are the two types of data files?
3. List some types of files stored as sequential-access files.
4. Describe an advantage of using random-access data files.
5. Describe a drawback of using random-access data files rather than sequential-access files?

CHAPTER 11, SECTION 2

Using Sequential-Access Files

Using a sequential-access file requires that you complete a few simple (but important) steps. First, the file must be opened. Data can then be stored or retrieved from the file. Finally, the file must be closed.

OPENING AND CLOSING FILES

A file is a container for data. Whether you think of a file as a drawer, a folder, or a box, the concept of opening and closing the file makes sense. Before you can put something in a file or take something out, the file must be opened. When you have finished accessing the file, it must be closed.

WARNING

In a paper filing system, a folder or drawer left open may result in important information getting misplaced or lost. Closing a computer file may be even more important. A data file left open by a program can be destroyed if a power failure occurs or if a program crash occurs. A closed data file is almost always protected from such occurrences.

Note

If you have the need to work with more than one file at a time, you can declare more than one file pointer and assign each file pointer to a different file on the disk.

DECLARING A FILE POINTER

C++ uses a special type of pointer called a ***file pointer*** to work with files. Recall that a pointer is a constant or variable that points to a location in memory. A file pointer is a variable that points to a position in a data file. You will assign an actual disk file to the file pointer, and then use the file pointer to access the file.

How you declare a file pointer varies depending on how you intend to use the file. If you will be storing data to the file (called ***writing***), you will declare a file pointer of type **ofstream** as shown below.

```
ofstream outfile;  // declares a file pointer for writing to file
```

If you will be getting data from a file (called ***reading***), you will declare a file pointer of type **ifstream** as shown below.

```
ifstream infile;  // declares a file pointer for reading a file
```

Note

The **ofstream** *and* **ifstream** *types are made available by including the header file named* **fstream.h**.

To help you remember the file pointer types, understand that **ofstream** is short for *output file stream* and **ifstream** is short for *input file stream*. Recall from Chapter 6 that a stream is data flowing from one place to another. Storing and retrieving data from files also involves input and output streams.

You can use any valid identifier for file pointer variables. The examples above used the names **outfile** and **infile** because they describe the purpose of the file pointer. In some situations, you may want to use a name that describes the data in the file, such as **customers_in** or **high_scores**.

Like any other variable, a file pointer must be declared outside the main function if it is to be accessed globally. There is nothing wrong with declaring file pointers globally. Some programmers prefer to have all their file pointers declared as global variables. In this book, we will declare file pointers globally when more than one function needs access to the file pointer.

After you have declared a file pointer, the next step is to open the file.

OPENING A FILE

When you ***open*** a file, a physical disk file is associated with the file pointer you declared. You must provide the name of the file you want to open and you must specify whether you want to put data in the file (write) or get data from the file (read). Consider the statements below.

```
ofstream high_scores;  // declares a file pointer for writing to file
high_scores.open("SCORES.DAT", ios::out); // create the output file
```

WARNING

You must be careful when opening a file for output. You will receive no warning if a file of the same name as what you are opening already exists. Many times you will want the existing file to be erased, but if not, it is up to you to use filenames that will not harm other data.

The statements above declare a file pointer named **high_scores** and opens a file on disk with the filename *SCORES.DAT*. After the filename, you must specify the way you want to access the file, called ***stream operation modes.*** There are several modes available, most of which you will learn about as you need them. For now, you need to know only two: **ios::out** and **ios::in**. Use **ios::out** when creating an output file and **ios::in** when opening a file for input. Your compiler may not require that you specify the stream operation mode when using these basic modes. It is a good idea, however, to specify the mode in all cases.

If a file named *SCORES.DAT* already exists in the same location, the existing file will be replaced with the new one. If no such file exists, one is created and made ready for data.

On the Net

See http://www.ProgramCPP.com topic 11.2.1 for information about slight differences in the way some compilers implement the opening of a file.

Let's look at another example. The statements below will open a file named *MYDATA.DAT* for input using a file pointer named **infile**.

```
ifstream infile;  // declares a file pointer for reading a file
infile.open("MYDATA.DAT", ios::in); // open the file for input
```

WARNING

Because opening a file associates a file pointer to an actual file on your computer, the filename you provide must be a filename that is legal for your operating system.

After a file has been opened, you can begin writing to it or reading from it (depending on the stream operation mode). When you complete your work with the file, you must ***close*** it. The statement below closes the file pointed to by the file pointer named **infile**.

```
infile.close(); // close the input file
```

The statement format is the same whether you are closing an input or output file.

WRITING DATA TO FILES

Writing data to a sequential-access data file employs the extraction operator (<<) that you use when printing data to the screen. Instead of using the output operator to direct data to the standard output device (**cout**), you direct the data to the file using the file pointer. For example, the program listed in Figure 11-3 prompts the user for his or her name and age, opens a file for output, and writes the data to the output file.

PITFALLS

By this time you are sending output to `cout` *by habit. When outputting to a file, make sure you use the file pointer name in place of* `cout`.

Notice the user is prompted to provide input in a manner similar to previous programs with which you have worked. Next, the statement **if (outfile)** appears. This code is the functional equivalent of **if (outfile != 0)**. Thus, if an error results in the attempt to open the file, the file pointer (**outfile**) is assigned the value of zero.

There are a number of conditions that can cause an error to occur when opening a file. The disk could be protected from new data being written to it. If you are attempting to open a file on a floppy disk, there is always the possibility that the disk is in the wrong drive. Because a disk drive is a hardware device, there may also be mechanical problems. Whatever the case, you must check to be sure that the file was opened successfully before sending data to the file.

Note

In most compilers, the `#include<fstream.h>` *directive makes the use of* `iostream.h` *unnecessary because* `fstream.h` *contains everything* `iostream.h` *has and more. In this chapter we include both for compatibility.*

```
#include<iostream.h>
#include<fstream.h>  // necessary for file I/O

int main()
{
 char user_name[25];
 int age;
 ofstream outfile; // declares file pointer named outfile

 cout << "Enter your name: "; // get name from user
 cin.get(user_name, 25);
 cout << "Enter your age: ";  // get age from user
 cin >> age;

 outfile.open("NAME_AGE.DAT",ios::out);  // open file for output

 if (outfile)  // If no error occurs while opening file
  {            // write the data to the file.
   outfile << user_name << endl;   // write the name to the file
   outfile << age << endl;         // write the age to the file
   outfile.close();  // close the output file
  }
 else          // If error occurred, display message.
  {
   cout << "An error occurred while opening the file.\n";
  }
 return 0;
}
```

FIGURE 11-3 This program stores the user's name and age to a sequential-access data file.

EXERCISE 11-1 WRITING SEQUENTIAL FILES

1. Enter the program shown in Figure 11-3. Save the source code as *FILEWRIT.CPP*.
2. Compile and run the program. Enter your name and age at the prompts.
3. When the program ends, open *NAME_AGE.DAT* using your compiler's text editor. The data you entered is in readable form in the file.
4. Close *NAME_AGE.DAT* and *FILEWRIT.CPP*.

When writing to files using the technique of Exercise 11-1, the output file receives data in text form in the same way output sent to **cout** appears on the screen. The output must be separated with spaces or end-of-line characters or the data will run together in the file. The reason you write data to a file is so that it can be read in again later. Therefore, you must separate data with spaces or end-of-line characters when you write it to a text file.

The program in Figure 11-4 uses a loop to ask the user for a series of numbers. Each number the user enters is stored in a file named *OUTFILE.DAT*. When the user enters a zero, the program ends.

```
#include<iostream.h>
#include<fstream.h>  // necessary for file I/O

int main()
{
 float x;           // variable for user input
 ofstream outfile; // declares file pointer named outfile

 outfile.open("FLOATS.DAT",ios::out);  // open file for output

 if (outfile)
  {
   cout << "Enter a series of floating-point numbers.\n"
        << "Enter a zero to end the series.\n";
   do   // repeat the loop until user enters zero
    {
     cin >> x;              // get number from user
     outfile << x << endl;  // write the number to the file
    } while (x != 0.0);
  }
 else
  {
   cout << "Error opening file.\n";
  }
 outfile.close();  // close the output file
 return 0;
}
```

FIGURE 11-4
This program stores numbers entered by the user until a zero is entered.

EXERCISE 11-2 MORE WRITING SEQUENTIAL FILES

1. Enter the program shown in Figure 11-4. Save the source code as *LOOPWRIT.CPP*.
2. Compile and run the program. Enter at least five or six floating point numbers and then be sure to end by entering a zero.
3. When the program ends, open *FLOATS.DAT* using your compiler's text editor. The numbers you entered are in readable form in the file.
4. Close *FLOATS.DAT* and *LOOPWRIT.CPP*.

READING DATA FROM FILES

The main reason data is written to a file is so that it can be later read and used again. In other cases, your program may read a data file created by another program.

As you learned earlier in this chapter, before you can read data from a file you must open the file for input using a file pointer. The statements below are an ex-

ample of declaring an input file pointer and opening a file named *MYDATA.DAT* for input.

```
ifstream infile;  // declares a file pointer for reading a file
infile.open("MYDATA.DAT", ios::in); // open the file for input
```

WARNING

When you open a file using `ios::in`*, the file must already exist. Some compilers will create an empty file and give unpredictable results. Other compilers may give an error message.*

Once the file is open, you can read the data using methods familiar to you. However, reading from a data file can be a little more complicated than getting input from the keyboard. In this chapter you will learn two methods of reading from data files. Other methods can be used, but these methods should give the desired results without too much complication.

First you will learn a method that can be used when the file contains only numeric data. Then you will learn how to read data that includes strings or a combination of strings and data.

READING NUMERIC DATA

When reading strictly numeric data, you can use the insertion operator (>>) as if you were getting input from the keyboard. Instead of using **`cin`** as the input stream, use your file pointer name. The program in Figure 11-5 reads the numbers you wrote to disk in Exercise 11-2, prints the numbers to the screen, and calculates the sum and average of the numbers.

EXERCISE 11-3 READING NUMERIC DATA

1. Retrieve the file *NUMREAD.CPP*. The program shown in Figure 11-5 appears.
2. Compile and run the program. The numbers you entered in Exercise 11-2 should appear, along with the sum of the values, and the average.
3. Close the source code file.

READING STRING DATA AND MIXED STRING AND NUMERIC DATA

When reading string data, use the **`get`** function as you do to read strings from the keyboard. Like using **`get`** with the keyboard, using **`get`** to read from files requires that you flush the stream to remove the end-of-line character.

When you have a file that includes both string data and numeric data (such as the name and age saved in Exercise 11-1), you should read all of the data as string data and convert the strings that contain numbers to numeric variables. This is done using a function called **`atoi`** or other related functions. The **`atoi`** function converts a number which is represented in a string to an actual integer value which can be stored in an integer variable. You will learn more about the functions which convert strings to numeric values in Chapter 13.

```
#include<fstream.h>  // necessary for file I/O
#include<iostream.h>
#include<iomanip.h>

int main()
{
 float x, sum, average;
 int count;
 ifstream infile;     // declares file pointer named infile

 infile.open("FLOATS.DAT",ios::in);  // open file for input

 sum = 0.0;  // initialize sum
 count = 0;  // initialize count
 if (infile)  // If no error occurred while opening file
  {           // input the data from the file.
  cout << "The numbers in the data file are as follows:\n"
       << setprecision(1); // set display to one decimal point
  cout.setf(ios::fixed);  // prevent E-notation

  do  // read numbers until 0.0 is encountered
   {
     infile >> x;          // get number from file
     cout << x << endl;    // print number to screen
     sum = sum + x;        // add number to sum
     count++;              // increment count of how many numbers read
   } while(x != 0.0);
   // Output sum and average.
   cout << "The sum of the numbers is " << sum << endl;
   cout << "The average of the numbers (excluding zero) is "
        << sum / (count - 1) << endl;
  }
 else           // If error occurred, display message.
  {
   cout << "An error occurred while opening the file.\n";
  }
 infile.close();  // close the input file
 return 0;
}
```

FIGURE 11-5
This program reads numbers from a data file until zero is encountered. Then the program reports the sum and average of the numbers.

The program completed in Exercise 11-1 stores the variable name as a string and age as an integer. However, the program in Figure 11-6 illustrates how both values can be read as strings and the age can be converted to an integer prior to printing the values to the screen.

EXERCISE 11-4 READING STRING DATA

1. Retrieve *STRREAD.CPP*. The program in Figure 11-6 appears on your screen.
2. Before running the program, analyze the source code to see the purpose of each statement.
3. Compile and run the program. Your name and age saved in Exercise 11-1 should appear in the output.
4. Close the source code file.

```
#include<fstream.h>  // necessary for file I/O
#include<iostream.h>
#include<stdlib.h>   // necessary for atoi function

int main()
{
 char user_name[25];
 char user_age[4];
 int age;
 ifstream infile;    // declares file pointer named infile

 infile.open("NAME_AGE.DAT",ios::in);  // open file for input

 if (infile)  // If no error occurred while opening file
  {           // input the data from the file.
   infile.get(user_name,25); // read the name from the file
   infile.ignore(80,'\n');
   infile.get(user_age,4); // read the age from the file as a string
   infile.ignore(80,'\n');
   age = atoi(user_age);
   cout << "The name read from the file is " << user_name << ".\n";
   cout << "The age read from the file is " << age << ".\n";
  }
 else          // If error occurred, display message.
  {
   cout << "An error occurred while opening the file.\n";
  }
 infile.close();  // close the input file
 return 0;
}
```

FIGURE 11-6
This program reads both the name and age from the file as strings and converts the age to an integer.

SECTION 11.2 QUESTIONS

1. What is the danger of leaving a file open?
2. What happens if you open a file for output that already exists?
3. Describe a condition that may cause an error when opening a file.
4. How can you test to see if an error occurred while opening a file?
5. How should you read numbers from a file when the file also contains string data?

PROBLEM 11.2.1

Write a program that asks the user's name, address, city, state, and ZIP code. The program should then save the data to a data file. Save the source code as *ADDRFILE.CPP*.

PROBLEM 11.2.2

Modify *HIGHTEMP.CPP* or *HTEMP2.CPP* so that it gets its temperatures from a data file, rather than from the keyboard. Create a text file with sample temperatures in order to test the program. Save the new source code as *HTEMP3.CPP*.

CHAPTER 11, SECTION 3

Sequential File Techniques

You now know how to write and read sequential-access data files. In this section you will learn techniques that will help you work more efficiently with files. You will learn how to add data to the end of a file, detect the end of a file, how to use multiple data files at the same time, and how to prompt the user for the name of a data file.

ADDING DATA TO THE END OF A FILE

One of the limitations of sequential-access files is that most changes require that you rewrite the file. For example, to insert data somewhere in a file, the file must be rewritten. You can, however, add data to the end of a file without rewriting the file. Adding data to the end of an existing file is called ***appending***. To append data to an existing file, open the file using the **`ios::app`** stream operation mode, as shown in the statement below.

```
outfile.open("MYDATA.DAT",ios::app);  // open file for appending
```

Note

If the file you open for appending does not exist, the operating system creates one just as if you had opened it using **`ios::out`** *mode.*

The program in Figure 11-7 opens the *NAME_AGE.DAT* data file and allows you to add more names and ages to the file. The only difference between the program in Figure 11-7 and the program you ran in Exercise 11-1 is the stream operation mode in the line that opens the file.

EXERCISE 11-5 APPENDING DATA TO A FILE

1. Open the program *FILEWRIT.CPP* that you saved in Exercise 11-1.
2. Run the program without any modification. Enter your name and age as input.
3. Open the output file (*NAME_AGE.DAT*) to see that the file's only contents are what you just entered. Close *NAME_AGE.DAT*.
4. Run the program again. This time enter *Scott McGrew* for the name and *18* for the age.

```
#include<fstream.h>  // necessary for file I/O
#include<iostream.h>

int main()
{
 char user_name[25];
 int age;
 ofstream outfile; // declares file pointer named outfile

 cout << "Enter your name: "; // get name from user
 cin.get(user_name, 25);
 cout << "Enter your age: ";  // get age from user
 cin >> age;

 outfile.open("NAME_AGE.DAT",ios::app);  // open file for appending

 if (outfile)  // If no error occurred while opening file
  {            // output the data to the file.
   outfile << user_name << endl;   // write the name to the file
   outfile << age << endl;         // write the age to the file
  }
 else          // If error occurred, display message.
  {
   cout << "An error occurred while opening the file.\n";
  }

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

FIGURE 11-7
This program adds data to the end of the data file, rather than replacing whatever data was in the file.

5. Open *NAME_AGE.DAT* again to see that Scott McGrew's name and age have replaced yours in the file. Close *NAME_AGE.DAT*.
6. Change the statement that opens the file for output:

```
outfile.open("NAME_AGE.DAT",ios::out);  // open file for output
```

to

```
outfile.open("NAME_AGE.DAT",ios::app);  // open file for appending
```

7. Run the program again. This time enter your name and age again.
8. Open *NAME_AGE.DAT* again to see that Scott McGrew's name and age have not been replaced by yours. Because the file was opened for appending, your name and age were added to the end of the file.
9. Close *NAME_AGE.DAT*.
10. Save the source code as *FILEAPP.CPP* and close the source code file.

On the Net

Some compilers handle appending to a file differently from what is shown here. For more information, see http://www.ProgramCPP.com topic 11.3.1.

DETECTING THE END OF A FILE

Often the length of a file is unknown to the programmer. In Exercise 11-3, the series of numbers you read into your program ended with a zero. Because you knew the data ended with a zero, you knew when to stop reading. In other cases, however, you may not have a value in the file that signals the end of the file. In those cases, there is a technique for detecting the end of the file.

When an attempt is made to read past the end of a file, the results vary depending on the data you are reading. If you try to read a number once the end of the file is reached, the last number in the file will probably be read again. In the case of string data, an empty string is returned. To know for sure whether the end of the file has been reached, use the **eof** function.

The **eof** function returns 1 (true) if an attempt has been made to read past the end of the file. To use the **eof** function, use the name of the file pointer, a period, and **eof()**, as shown in the code segment below.

```
infile >> x;              // Get number from file.
if (!infile.eof())
 {                        // If not the end of file,
  cout << x << endl;      // print the number to the screen.
 }
```

In the example above, the not operator (!) is used so that the statement in the if structure will be executed if the end of the file has *not* been reached. Figure 11-8 shows a program that uses the code segment above.

Note

You can use the **eof** *function whether working with string or numeric data.*

EXERCISE 11-6 DETECTING THE END OF A FILE

1. Retrieve the source code file *READEOF.CPP*. The program shown in Figure 11-8 appears.
2. Compile and run the program. If the program reports that an error occurred while opening the file, make sure that the *PRICES.DAT* data file is in the default directory, or provide the path to the correct directory in the statement that opens the file. Your instructor can assist you if necessary.
3. When the program runs correctly, close the source code file.

```
#include<fstream.h>  // necessary for file I/O
#include<iostream.h>
#include<iomanip.h>  // necessary for setprecision manipulator

int main()
{
 float x;             // Declare variable used for input.
 ifstream infile;     // Declare file pointer named infile.

 infile.open("PRICES.DAT",ios::in);  // Open file for input.

 if (infile)  // If no error occurred while opening file,
  {           // input the data from the file.
  cout << "The prices in the file are: \n" << setprecision(2);
  do  // Loop while not the end of the file.
   {
     infile >> x;           // Get number from file.
     if (!infile.eof())
      {                     // If not the end of file,
       cout << x << endl;   // print the number to the screen.
      }
   } while (!infile.eof());
  }
 else           // If error occurred, display message.
  {
   cout << "An error occurred while opening the file.\n";
  }
 infile.close();  // Close the input file.
 return 0;
}
```

FIGURE 11-8
This program continues to read floating point numbers from a file until the end of the file is reached.

On the Net

Some compilers require slightly different syntax to detect the end of a file. For more information, see http://www.ProgramCPP.com topic 11.3.2.

USING MULTIPLE FILES

As mentioned earlier, you can have more than one file open at a time. Just declare and use a separate file pointer for each file. Why would you want more than one file open at a time? There are many reasons. Let's look at a few of them.

Suppose you need to add some data to the middle of a file. Since you cannot insert data in a file, you must read the data from the original file and write it to a new file. At the position where the new data is to be inserted, you write the data to the new file and then continue writing the rest of the data from the original file.

Large database programs, called ***relational database systems,*** use multiple database files. For example, a program for an animal clinic might use one database file to store the information about the owners of pets and another file for the information about the pets themselves. The database of pets would include a field that linked the pet to its owner in the other database file. The term *relational*

```
#include<fstream.h>  // necessary for file I/O
#include<iostream.h>

int main()
{
 char ch;             // Declare character variable used for input.
 ifstream infile;     // Declare file pointer for input file.
 ofstream outfile;    // Declare file pointer for output file.

 infile.open("LOWER.TXT",ios::in);   // Open file for input.
 outfile.open("UPPER.TXT",ios::out); // Open file for output.

 if ((!infile) || (!outfile))  // If file error on either file,
  {                            // print message and stop program.
    cout << "Error opening file.\n";
    return 0;
  }

 infile.unsetf(ios::skipws); // prevents spaces from being skipped

 while (!infile.eof())  // Loop while not the end of the file.
  {
   infile >> ch;         // Get character from file.
   if (!infile.eof())
    {
     if ((ch > 96) && (ch < 123))  // if character is lowercase a-z,
      {                            // subtract 32 to make uppercase.
       ch = ch - 32;
      }
     outfile << ch;      // Write character to output file.
    }
  } // end of while loop

 infile.close();  // Close the input file.
 outfile.close(); // Close the output file.
 return 0;
}
```

FIGURE 11-9
This program converts all lowercase characters in a file to uppercase.

database comes from the fact that multiple database files are related or linked by certain fields.

Another example of when more than one file may be necessary is when performing a conversion process on a file. Suppose you need to convert all of the lowercase alphabetic characters in a file to uppercase letters. The program in Figure 11-9 reads the text in one file one character at a time, converts where necessary, and writes the converted characters to another file.

To convert the lowercase characters to uppercase, the ASCII value of the character is tested to see if it falls within the range of 97 to 122, which are the ASCII values of the lowercase letters. If the character falls within that range, the value in the character variable is reduced by 32 which converts it to the uppercase equivalent of the letter. This works because the uppercase letters have ASCII values from 65 to 90, which is 32 less than 97 to 122. To see the ASCII values in a table, refer to Appendix A.

Note

The code segment

```
if ((ch > 96) && (ch < 123))  // if character is lowercase a-z,
 {                            // subtract 32 to make uppercase.
  ch = ch - 32;
 }
```

could be rewritten as

```
if ((ch >= 'a') && (ch <= 'z'))  // if character is lowercase a-z,
 {                               // subtract 32 to make uppercase.
  ch = ch - 32;
 }
```

The statement **`infile.unsetf(ios::skipws);`** is probably new to you. Without this statement, the **`infile >> ch;`** statement will not read the spaces, tabs, or end-of-line characters. By default, white space (spaces, tabs, and end-of-line characters) are ignored when reading from a file. The **`skipws`** (which is short for *skip white space*) setting allows you to override the default.

The statement **`infile.unsetf(ios::skipws);`** is necessary in this program because the spaces, tabs, and end-of-file characters that are in the input file must be included in the output file. If the white space is skipped when the input file is read, the white space will not appear in the converted file.

EXERCISE 11-7 USING MULTIPLE FILES

1. Open *CONVERT.CPP*. The program in Figure 11-9 appears.
2. Create a text file that includes a variety of characters, both uppercase and lowercase. Save the file as ASCII text and name it *LOWER.TXT*. Make sure the file is in the current directory or supply the path in the statement that opens the file.
3. Compile and run the program. The program produces no output on the screen.
4. Open *UPPER.TXT* to see that all of the lowercase letters have been converted to uppercase.
5. Close *UPPER.TXT* and *LOWER.TXT*. Leave *CONVERT.CPP* open for the next exercise.

WARNING

Open files only as you need them and close them as soon as possible to avoid data loss in the event of power failure or program crash.

PROMPTING FOR FILE NAMES

Up to now we have used filenames that are ***hard coded*** into the program, meaning the names cannot be changed when the program runs. In programs with hard-coded filenames, the filenames appear in the source code as string literals. To make a program such as the one in Exercise 11-7 more flexible, you can prompt the user for the filenames. The code segment in Figure 11-10 prompts the user for two filenames and opens the files.

EXERCISE 11-8 PROMPTING FOR FILENAMES

1. Replace the two statements that open the files with the code from Figure 11-10. Save the modified source code as *CONVERT2.CPP*.
2. Run the program. Enter *LOWER.TXT* as the input file and *UPPER2.TXT* as the output file.
3. Check the output file (*UPPER2.TXT*) to make sure the conversion took place.
4. Close the source code file and the output file.

```
char inputfile[13];
char outputfile[13];

cout << "Enter the name of the input file: ";
cin.get(inputfile,13);
cin.ignore(80, '\n'};
cout << "Enter the name of the output file: ";
cin.get(outfile,13);
cin.ignore(80,'\n');

infile.open(inputfile, ios::in);     // Open file for input.
outfile.open(outputfile, ios::out), // Open file for output.
```

FIGURE 11-10
The filename can be provided by the user.

SECTION 11.3 QUESTIONS

1. What stream operation mode is used when adding data to the end of an existing file?
2. Write a code segment that prints the message END OF FILE REACHED if the file pointer named **infile** has reached the end of the file.
3. Give an example of a situation that may require that more than one file be opened at a time.
4. What statement prevents spaces in a file from being skipped?
5. What is the term used to describe a filename that is part of the program and cannot be changed by the user when the program runs?

PROBLEM 11.3.1

Modify the program written in Problem 11.2.1 so that it appends a name and address to the data file every time the program is run, rather than rewriting the output file. Run the program several times to append several names and addresses to the output file. Save the source code as *NAMEFILE.CPP*.

PROBLEM 11.3.2

Write a program that reads the data saved in Problem 11.3.1 and prints the data to the screen. Save the source code as *NAMEPRNT.CPP*.

PROBLEM 11.3.3

Modify the program you saved in Exercise 11-8 to convert uppercase letters to lowercase, rather than lowercase to uppercase. Save the modified source code as *UPPERLOW.CPP*.

KEY TERMS

appending
close
file pointer
hard coded
open
random-access file
reading
relational database systems
sequential-access file
stream operation modes
writing

SUMMARY

- Data files allow for the storage of data prior to a program's ending and the computer being turned off. Data files also allow for more data storage than can fit in RAM.
- A sequential-access file is like an audio cassette tape. Data must be written to and read from the file sequentially.
- A random-access file is like a compact disc. Any record can be accessed directly.
- The first step to using a file is declaring a file pointer. Some file pointers are for writing data and some are for reading data.
- After a file pointer has been declared, the next step is to open the file. Opening a file associates the file pointer with a physical data file.
- After data is written or read, the file must be closed.
- The extraction operator (<<) is used to write to a data file.

- When reading numeric data, use the insertion operator. When reading string data or when reading from a file with both string and numeric data, read the data as strings and convert the numbers to numeric variables.
- Adding data to the end of an existing file is called appending.
- The `eof` function detects the end of a file.
- You can use more than one file at a time by declaring multiple file pointers.
- You can prompt the user for input and output filenames.

PROJECTS

PROJECT 11-1 • FILE PROCESSING

Write a program that copies a text file. The program should remove any blank lines and spaces between words as it writes the new file.

PROJECT 11-2 • FILE PROCESSING

Write a program that prompts the user for the name of a text file and reports the number of characters in the text file, and the number of end-of-line characters in the text file.

PROJECT 11-3 • STRING CONVERSION

Write a program that prompts the user for his or her full name. The program should check the first character of each name to make sure it appears in uppercase, making conversion where necessary. For example, if the user enters *jessica hope baldwin*, the program should output *Jessica Hope Baldwin*.

PROJECT 11-4 • STRING MANIPULATION

Extend the program from Project 11-3 to output the name last name first in the format below.

Baldwin, Jessica Hope