6. **Phone Number Lookup**

   Recall that Programming Exercise 7 in Chapter 8 asked you to design a program with two parallel arrays: a `String` array named `people` and a `String` array named `phoneNumbers`. The program allows you to search for a person's name in the `people` array. If the name is found, it displays that person's phone number. Modify the program so it uses the binary search algorithm instead of the sequential search algorithm.

7. **Search Benchmarks**

   Design an application that has an array of at least 20 integers. It should call a module that uses the sequential search algorithm to locate one of the values. The module should keep a count of the number of comparisons it makes until it finds the value. Then the program should call another module that uses the binary search algorithm to locate the same value. It should also keep a count of the number of comparisons it makes. Display these values on the screen.

8. **Sorting Benchmarks**

   Modify the modules presented in this chapter that perform the bubble sort, selection sort, and insertion sort algorithms on an `Integer` array, such that each module keeps a count of the number of swaps it makes.

   Then, design an application that uses three identical arrays of at least 20 integers. It should call each module on a different array, and display the number of swaps made by each algorithm.

CHAPTER 10

Files

TOPICS

10.1 Introduction to File Input and Output

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

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

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

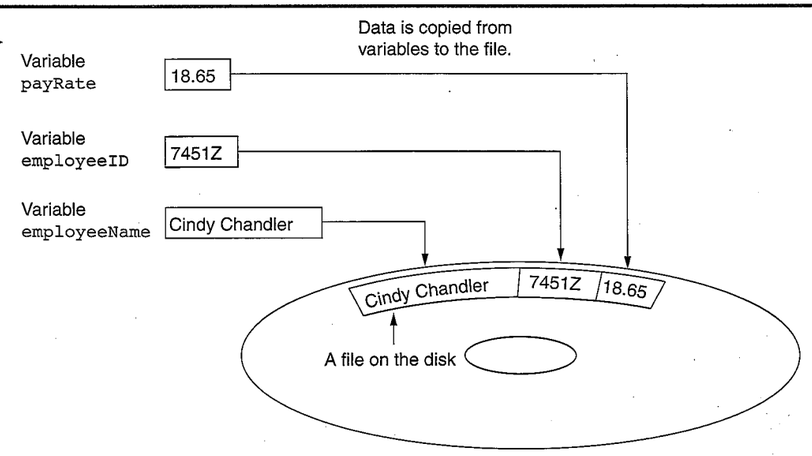
- **Word processors:** Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and printed.
- **Image editors:** Image editing programs are used to draw graphics and edit images such as the ones that you take with a digital camera. The images that you create or edit with an image editor are saved in files.
- **Spreadsheets:** Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved in a file for use later.

- **Games:** Many computer games keep data stored in files. For example, some games keep a list of player names with their scores stored in a file. These games typically display the players' names in order of their scores, from highest to lowest. Some games also allow you to save your current game status in a file so you can quit the game and then resume playing it later without having to start from the beginning.
- **Web browsers:** Sometimes when you visit a Web page, the browser stores a small file known as a *cookie* on your computer. Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

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

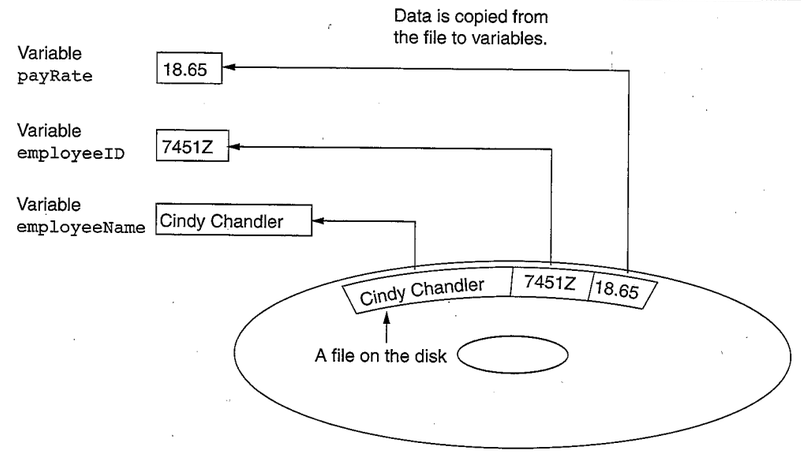
Programmers usually refer to the process of saving data in a file as "writing data to" the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 10-1. The term *output file* is used to describe a file that data is written to. It is called an output file because the program stores output in it.

Figure 10-1 Writing data to a file



The process of retrieving data from a file is known as "reading data from" the file. When a piece of data is read from a file, it is copied from the file into a variable in RAM. Figure 10-2 illustrates this. The term *input file* is used to describe a file that data is read from. It is called an input file because the program gets input from the file.

Figure 10-2 Reading data from a file



This chapter discusses how to design programs that write data to files and read data from files. There are always three steps that must be taken when a file is used by a program.

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

Types of Files

In general, there are two types of files: text and binary. A *text file* contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A *binary file* contains data that has not been converted to text. As a consequence, you cannot view the contents of a binary file with a text editor.

File Access Methods

Most programming languages provide two different ways to access data stored in a file: sequential access and direct access. When you work with a *sequential access file,*

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

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

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

Creating a File and Writing Data to It

Most computer users are accustomed to the fact that files are identified by a filename. For example, when you create a document with a word processor and then save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Figure 10-3 shows how three files named `cat.jpg`, `notes.txt`, and `resume.doc` might be represented in Windows Explorer.

Figure 10-3 Three files

Each operating system has its own rules for naming files. Many systems support the use of *filename extensions*, which are short sequences of characters that appear at the end of a filename preceded by a period (which is known as a "dot"). For example, the files depicted in Figure 10-3 have the extensions `.jpg`, `.txt`, and `.doc`. The extension usually indicates the type of data stored in the file. For example, the `.jpg` extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The `.txt` extension usually indicates that the file contains text. The `.doc` extension usually indicates that the file contains a Microsoft Word document. (In this book we will use the `.dat` extension with all of the files we create in our programs. The `.dat` extension simply stands for "data.")

When writing a program that performs an operation on a file, there are two names that you have to work with in the program's code. The first of these is the filename that identifies the file on the computer's disk. The second is an internal name that is similar to a variable name. In fact, you usually declare a file's internal name in a manner that is similar to declaring a variable. The following example shows how we declare a name for an output file in our pseudocode:

```
Declare OutputFile customerFile
```


This statement declares two things.

- The word `OutputFile` indicates the *mode* in which we will use the file. In our pseudocode, `OutputFile` indicates that we will be writing data to the file.
- The name `customerFile` is the internal name we will use to work with the output file in our code.

Although the syntax for making this declaration varies greatly among programming languages, you typically have to declare both the mode in which you will use a file and the file's internal name before you can work with the file.

The next step is to open the file. In our pseudocode we will use the `Open` statement. Here is an example:

```
Open customerFile "customers.dat"
```

The word `Open` is followed by an internal name that was previously declared, and then a string that contains a filename. After this statement executes, a file named `customers.dat` will be created on the disk, and we will be able to use the internal name `customerFile` to write data to the file.

WARNING! Remember, when you open an output file you are creating the file on the disk. In most languages, if a file with the specified external name already exists when the file is opened, the contents of the existing file will be erased.

Writing Data to a File

Once you have opened an output file you can write data to it. In our pseudocode we will use the `Write` statement to write data to a file. For example,

```
Write customerFile "Charles Pace"
```

writes the string `"Charles Pace"` to the file that is associated with `customerFile`. You can also write the contents of a variable to a file, as shown in the following pseudocode:

```
Declare String name = "Charles Pace"
Write customerFile name
```

The second statement in this pseudocode writes the contents of the `name` variable to the file associated with `customerFile`. (These examples show a string being written to a file, but you can also write numeric values.)

Closing an Output File

Once a program is finished working with a file, it should close the file. Closing a file disconnects the program from the file. In some systems, failure to close an output file can cause a loss of data. This happens because the data that is written to a file is first written to a *buffer*, which is a small "holding section" in memory. When the buffer is full, the computer's operating system writes the buffer's contents to the file. This technique increases the system's performance, because writing data to memory is faster than writing it to a disk. The process of closing an output file forces any unsaved data that remains in the buffer to be written to the file.

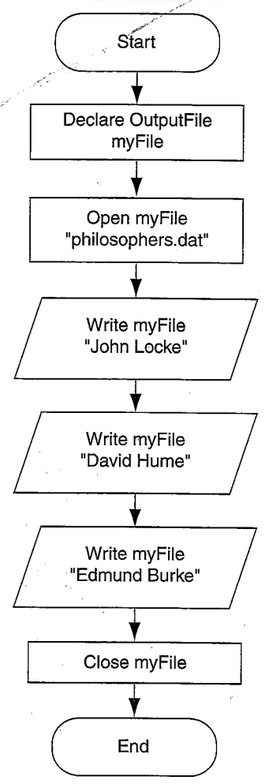
In our pseudocode, we will use the `Close` statement to close a file. For example,

```
Close customerFile
```

closes the file that is associated with the name `customerFile`.

Program 10-1 shows the pseudocode for a sample program that opens an output file, writes data to it, and then closes it. Figure 10-4 shows a flowchart for the program. Because the `Write` statements are output operations, they are shown in parallelograms.

Figure 10-4 Flowchart for Program 10-1



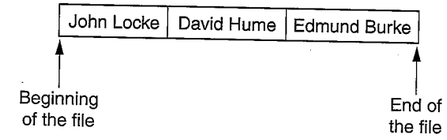
Program 10-1

```
1 // Declare an internal name for an output file.
2 Declare OutputFile myFile
3
4 // Open a file named philosophers.dat on
5 // the disk.
6 Open myFile "philosophers.dat"
7
8 // Write the names of three philosophers
9 // to the file.
10 Write myFile "John Locke"
11 Write myFile "David Hume"
12 Write myFile "Edmund Burke"
13
14 // Close the file.
15 Close myFile
```


The statement in line 2 declares the name `myFile` as the internal name for an output file. Line 6 opens the file `philosophers.dat` on the disk and creates an association between the file and the internal name `myFile`. This will allow us to use the name `myFile` to work with the file `philosophers.dat`.

The statements in lines 10 through 12 write three items to the file. Line 10 writes the string `"John Locke"`, line 11 writes the string `"David Hume"`, and line 12 writes the string `"Edmund Burke"`. Line 15 closes the file. If this were an actual program and were executed, the three items shown in Figure 10-5 would be written to the `philosophers.dat` file.

Figure 10-5 Contents of the file `philosophers.dat`



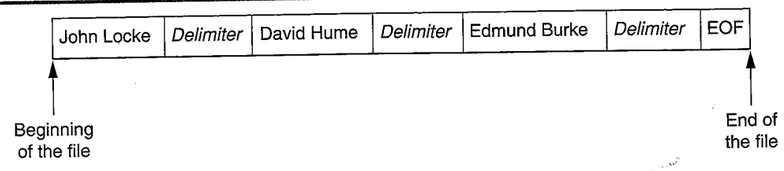
Notice that the items in the file appear in the same order that they were written by the program. "John Locke" is the first item, "David Hume" is the second item, and "Edmund Burke" is the third item. You will see the significance of this when we read data from the file momentarily.

Delimiters and the EOF Marker

Figure 10-5 shows that three items were written to the `philosphers.dat` file. In most programming languages, the actual contents of the file would be more complex than the figure shows. In many languages, a special character known as a delimiter is written to a file after each item. A *delimiter* is simply a predefined character or set of characters that marks the end of each piece of data. The delimiter's purpose is to separate the different items that are stored in a file. The exact character or set of characters that are used as delimiters varies from system to system.

In addition to delimiters, many systems write a special character or set of characters, known as an *end-of-file (EOF) marker,* at the end of a file. The purpose of the EOF marker is to indicate where the file's contents end. The character that is used as the EOF marker also varies among different systems. Figure 10-6 shows the layout of the `philosphers.dat` file, with delimiters and an EOF marker.

Figure 10-6 Contents of the file `philosophers.dat` with delimiters and the EOF marker



Reading Data from a File

To read data from an input file, you first declare an internal name that you will use to reference the file. In pseudocode we will use a `Declare` statement such as this:

```
Declare InputFile inventoryFile
```

This statement declares two things.

- The word `InputFile` indicates the mode in which we will use the file. In our pseudocode, `InputFile` indicates that we will be reading data from the file.
- The name `inventoryFile` is the internal name we will use to work with the output file in our code.

As previously mentioned, the actual syntax for declaring a file mode and internal name varies greatly among programming languages.

The next step is to open the file. In our pseudocode we will use the `Open` statement. For example, in

```
Open inventoryFile "inventory.dat"
```

the word `Open` is followed by an internal name that was previously declared, and then a string that contains a filename. After this statement executes, the file named `inventory.dat` will be opened, and we will be able to use the internal name `inventoryFile` to read data from the file.

Because we are opening the file for input, it makes sense that the file should already exist. In most systems, an error will occur if you try to open an input file but the file does not exist.

Reading Data

Once you have opened an input file you can read data from it. In our pseudocode we will use the `Read` statement to read a piece of data from a file. The following is an example (assume `itemName` is a variable that has already been declared).

```
Read inventoryFile itemName
```

This statement reads a piece of data from the file that is associated with `inventoryFile`. The piece of data that is read from the file will be stored in the `itemName` variable.

Closing an Input File

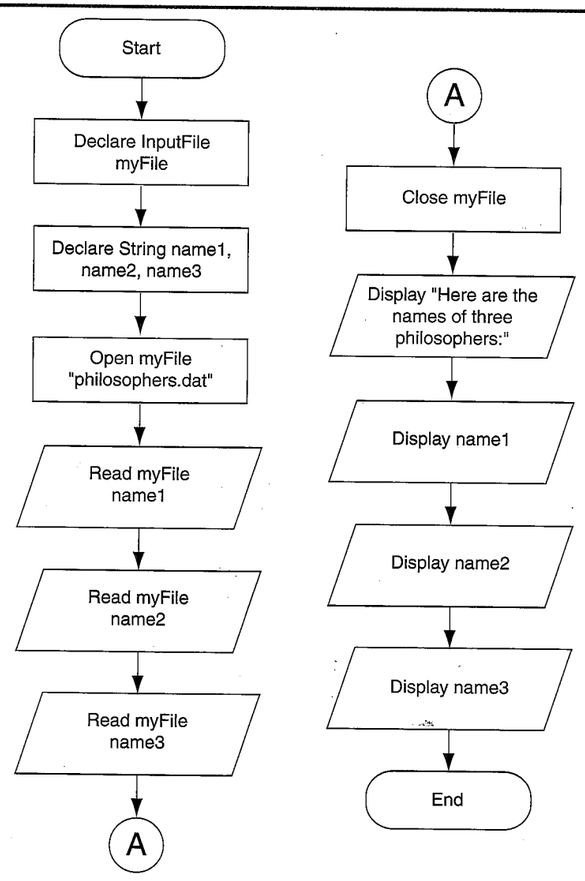
As previously mentioned, a program should close a file when it is finished working with it. In our pseudocode, we will use the `Close` statement to close input files, in the same way that we close output files. For example,

```
Close inventoryFile
```

closes the file that is associated with the name `inventoryFile`.

Program 10-2 shows the pseudocode for a program that opens the `philosophers.dat` file that would be created by Program 10-1, reads the three names from the file, closes the file, and then displays the names that were read. Figure 10-7 shows a flowchart for the program. Notice that the `Read` statements are shown in parallelograms.

Figure 10-7 Flowchart for Program 10-2



Program 10-2

```
1 // Declare an internal name for an input file.
2 Declare InputFile myFile
3
4 // Declare three variables to hold values
5 // that will be read from the file.
6 Declare String name1, name2, name3
7
8 // Open a file named philosophers.dat on
9 // the disk.
10 Open myFile "philosophers.dat"
11
12 // Read the names of three philosophers
13 // from the file into the variables.
14 Read myFile name1
15 Read myFile name2
16 Read myFile name3
17
18 // Close the file.
19 Close myFile
20
21 // Display the names that were read.
22 Display "Here are the names of three philosophers:"
23 Display name1
24 Display name2
25 Display name3
```

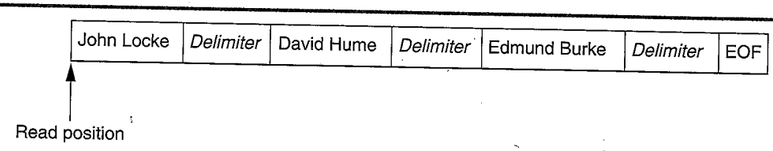
Program Output

```
Here are the names of three philosophers:
John Locke
David Hume
Edmund Burke
```

The statement in line 2 declares the name `myFile` as the internal name for an input file. Line 6 declares three `String` variables: `name1`, `name2`, and `name3`. We will use these variables to hold the values read from the file. Line 10 opens the file `philosophers.dat` on the disk and creates an association between the file and the internal name `myFile`. This will allow us to use the name `myFile` to work with the file `philosophers.dat`.

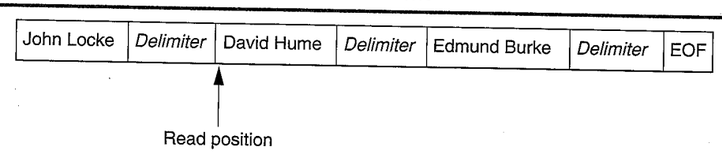
When a program works with an input file, a special value known as a *read position* is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. When an input file is opened, its read position is initially set to the first item in the file. After the statement in line 10 executes, the read position for the `philosophers.dat` file will be positioned as shown in Figure 10-8.

Figure 10-8 Initial read position



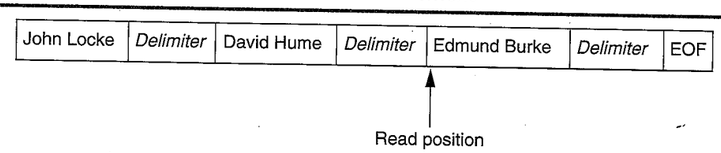
The `Read` statement in line 14 reads an item from the file's current read position and stores that item in the `name1` variable. Once this statement executes, the `name1` variable will contain the string `"John Locke"`. In addition, the file's read position will be advanced to the next item in the file, as shown in Figure 10-9.

Figure 10-9 Read position after the first `Read` statement



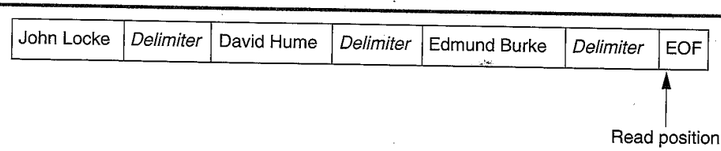
Another `Read` statement appears in line 15. This reads an item from the file's current read position and stores that value in the `name2` variable. Once this statement executes, the `name2` variable will contain the string `"David Hume"`. The file's read position will be advanced to the next item, as shown in Figure 10-10.

Figure 10-10 Read position after the second `Read` statement



Another `Read` statement appears in line 16. This reads the next item from the file's current read position and stores that value in the `name3` variable. Once this statement executes, the `name3` variable will contain the string `"Edmund Burke"`. The file's read position will be advanced to the EOF marker, as shown in Figure 10-11.

Figure 10-11 Read position after the third `Read` statement



The statement in line 19 closes the file. The `Display` statements in lines 23 through 25 display the contents of the `name1`, `name2`, and `name3` variables.

NOTE: Did you notice that Program 10-2 read the items in the `philosophers.dat` file in sequence, from the beginning of the file to the end of the file? Recall from our discussion at the beginning of the chapter that this is the nature of a sequential access file.

Appending Data to an Existing File

In most programming languages, when you open an output file and a file with the specified external name already exists on the disk, the existing file will be erased and a new empty file with the same name will be created. Sometimes you want to preserve an existing file and append new data to its current contents. Appending data to a file means writing new data to the end of the data that already exists in the file.

Most programming languages allow you to open an output file in *append mode,* which means the following:

- If the file already exists, it will not be erased. If the file does not exist, it will be created.
- When data is written to the file, it will be written at the end of the file's current contents.

The syntax for opening an output file in append mode varies greatly from one language to another. In pseudocode we will simply add the word `AppendMode` to the `Declare` statement, as shown here:

```
Declare OutputFile AppendMode myFile
```

This statement declares that we will use the internal name `myFile` to open an output file in append mode. For example, assume the file `friends.dat` exists and contains the following names:

```
Joe
Rose
Greg
Geri
Renee
```

The following pseudocode opens the file and appends additional data to its existing contents.

```
Declare OutputFile AppendMode myFile
Open myFile "friends.dat"
Write myFile "Matt"
Write myFile "Chris"
Write myFile "Suze"
Close myFile
```

After this program runs, the file `friends.dat` will contain the following data:

```
Joe
Rose
Greg
Geri
Renee
```



```
Matt
Chris
Suze
```

Checkpoint

10.1 Where are files normally stored?

10.2 What is an output file?

10.3 What is an input file?

10.4 What three steps must be taken by a program when it uses a file?

10.5 In general, what are the two types of files? What is the difference between these two types of files?

10.6 What are the two types of file access? What is the difference between these two?

10.7 When writing a program that performs an operation on a file, what two file-associated names do you have to work with in your code?

10.8 In most programming languages, if a file already exists what happens to it if you try to open it as an output file?

10.9 What is the purpose of opening a file?

10.10 What is the purpose of closing a file?

10.11 Generally speaking, what is a delimiter? How are delimiters typically used in files?

10.12 In many systems, what is written at the end of a file?

10.13 What is a file's read position? Initially, where is the read position when an input file is opened?

10.14 In what mode do you open a file if you want to write data to it, but you do not want to erase the file's existing contents? When you write data to such a file, to what part of the file is the data written?

10.2 Using Loops to Process Files

CONCEPT: Files usually hold large amounts of data, and programs typically use a loop to process the data in a file.

VideoNote
Using Loops to Process Files

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the pseudocode in Program 10-3. This program gets sales amounts for a series of days from the user and stores those amounts in a file named `sales.dat`. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days. Figure 10-12 shows the contents of