Chapter 7

File Handling and Applications

- Chapter Introduction
- 7-1 Understanding Computer Files
  - 7-1a Organizing Files
- 7-2 Understanding the Data Hierarchy
- 7-3 Performing File Operations
  - 7-3a Declaring a File
  - 7-3b Opening a File
  - 7-3c Reading Data from a File
  - 7-3d Writing Data to a File
  - 7-3e Closing a File
  - 7-3f A Program That Performs File Operations
- 7-4 Understanding Sequential Files and Control Break Logic
  - 7-4a Understanding Control Break Logic
- 7-5 Merging Sequential Files
- 7-6 Master and Transaction File Processing
- 7-7 Random Access Files
- 7-8 Chapter Review
  - 7-8a Chapter Summary
  - 7-8b Key Terms
  - 7-8c Review Questions
  - 7-8d Exercises

Chapter Introduction

In this chapter, you will learn about:

- Computer files
- The data hierarchy
- Performing file operations
- Sequential files and control break logic
- Merging files
- Master and transaction file processing
- Random access files

7-1 Understanding Computer Files

In Chapter 1, you learned that computer memory, or random access memory (RAM), is volatile, temporary storage. When you write a program that stores a value in a variable, you are using temporary storage; the value you store is lost when the program ends or the computer loses power.

Permanent, nonvolatile storage, on the other hand, is not lost when a computer loses power. When you write a program and save it to a disk, you are using permanent storage.

When discussing computer storage, *temporary* and *permanent* refer to volatility, not length of time. For example, a *temporary* variable might exist for several hours in a very large program or one that runs in an infinite loop, but a *permanent* piece of data might be saved and then deleted by a user within a few seconds. Because you can erase data from files, some programmers prefer the term *persistent storage* to permanent storage. In other words, you can remove data from a file stored on a device such as a disk drive, so it is not technically permanent. However, the data remains in the file even when the computer loses power, so, unlike in RAM, the data persists.

A **computer file** (A collection of data stored on a nonvolatile device in a computer system.) is a collection of data stored on a nonvolatile device in a computer system. Files exist on **permanent storage devices** (Hardware devices that hold nonvolatile data; examples include hard disks, DVDs, Zip disks, USB drives, and reels of magnetic tape.) , such as hard disks, DVDs, USB drives, and reels of magnetic tape. The two broad categories of files are:

- **Text files** (Files that contain data that can be read in a text editor.) , which contain data that can be read in a text editor because the data has been encoded using a scheme such as ASCII or Unicode. Text files might include facts and figures used by business programs, such as a payroll file that contains employee numbers, names, and salaries. The programs in this chapter will use text files.

- **Binary files** (Files that contain data that has not been encoded as text.) , which contain data that has not been encoded as text. Examples include images and music.

Although their contents vary, files have many common characteristics, as follows:

- Each file has a name. The name often includes a dot and a file extension that describes the type of the file. For example, the extension *.txt* indicates a plain text file, *.dat* is a common extension for a data file, and *.jpg* is used as an extension on image files in Joint Photographic Experts Group format.

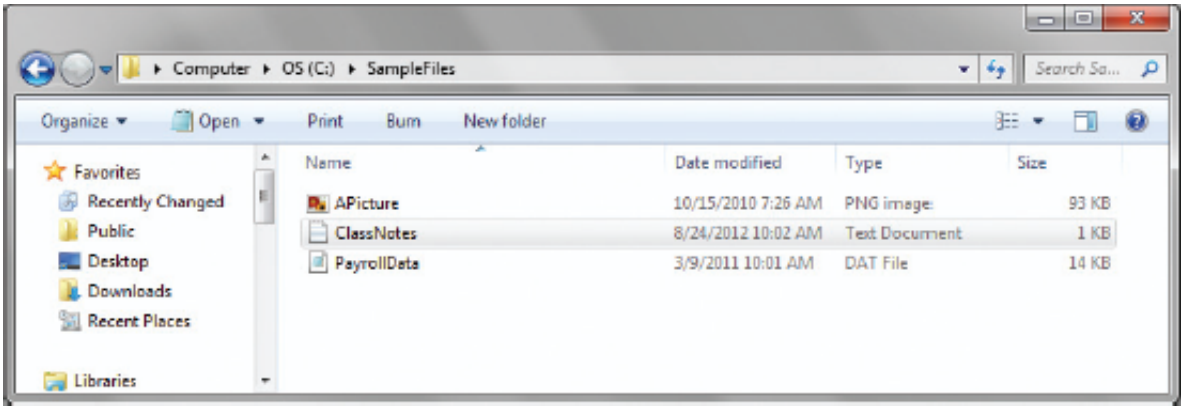
- Each file has specific times associated with it—for example, its creation time and the time it was last modified.

- Each file occupies space on a section of a storage device; that is, each file has a size. Sizes are measured in bytes. A **byte** (A unit of computer storage equal to eight bits.) is a small unit of storage; for example, in a simple text file, a byte holds only one character. Because a byte is so small, file sizes usually are expressed in **kilobytes** (Approximately 1000 bytes.) (thousands of bytes), **megabytes** (A million bytes.) (millions of bytes), or **gigabytes** (A billion bytes.) (billions of bytes). Appendix A contains more information on bytes and how file sizes are expressed.

Figure 7-1 shows how some files look when you view them in Microsoft Windows.

Figure 7-1

Three Stored Files and Their Attributes



Watch the video *Understanding Files*.

Chapter 7: File Handling and Applications: 7-1a Organizing Files
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

7-1a Organizing Files

Computer files on a storage device are the electronic equivalent of paper files stored in file cabinets. With a paper file, the easiest way to store a document is to toss it into a file cabinet drawer without a folder. However, for better organization, most office clerks place paper documents in folders—and most computer users organize their files into folders or directories. **Directories** (Organization units on storage devices; each can contain multiple files as well as additional directories. In a graphical interface system, directories are often called *folders*.) and **folders** (Organization units on storage devices; each can contain multiple files as well as additional folders. Folders are graphic directories.) are organization units on storage devices; each can contain multiple files as well as additional directories. The combination of the disk drive plus the complete hierarchy of directories in which a file resides is its **path** (The combination of a file's disk drive and the complete hierarchy of directories in which the file resides.) . For example, in the Windows operating system, the following line would be the complete path for a file named PayrollData.dat on the C drive in a folder named SampleFiles within a folder named Logic:

The terms *directory* and *folder* are used synonymously to mean an entity that organizes files. *Directory* is the more general term; the term *folder* came into use in graphical systems. For example, Microsoft began calling directories *folders* with the introduction of Windows 95.

Two Truths & A Lie

Understanding Computer Files

1. Temporary storage is volatile.

   T F

2. Computer files exist on permanent storage devices, such as RAM.

   T F

3. A file's path is the hierarchy of folders in which it is stored.

   T F

Chapter 7: File Handling and Applications: 7-2 Understanding the Data Hierarchy
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

7-2 Understanding the Data Hierarchy

When businesses store data items on computer systems, they are often stored in a framework called the **data hierarchy** (Represents the relationship of databases, files, records, fields, and characters.) that describes the relationships between data components. The data hierarchy consists of the following:

- **Characters** (A letter, number, or special symbol such as *A*, *7*, or *$*.) are letters, numbers, and special symbols, such as *A*, *7*, and *$*. Anything you can type from the keyboard in one keystroke is a character, including seemingly "empty" characters such as spaces and tabs. Computers also recognize characters you cannot enter from a standard keyboard, such as foreign-alphabet characters like ' or *Σ*. Characters are made up of smaller elements called bits, but just as most human beings can use a pencil without caring whether atoms are flying around inside it, most computer users store characters without thinking about these bits.

- **Fields** (A single data item, such as `lastName`, `streetAddress`, or `annualSalary`.) are data items that represent a single attribute of a record and are composed of one or more characters. Fields include items such as `lastName`, `middleInitial`, `streetAddress`, or `annualSalary`.

- **Records** (A group of fields stored together as a unit because they hold data about a single entity.) are groups of fields that go together for some logical reason. A random name, address, and salary aren't very useful, but if they're *your* name, *your* address, and *your* salary, then that's your record. An inventory record might contain fields for item number, color, size, and price; a student record might contain an ID number, grade point average, and major.

- **Files** (A group of records that go together for some logical reason.) are groups of related records. The individual records of each student in your class might go together in a file called Students.dat. Similarly, records of each person at your company might be in a file called Personnel.dat. Some files can have just a few records. For example, a student file for a college seminar might have only 10 records. Others, such as the file of credit card holders for a major department store chain or policy holders of a large insurance company, can contain thousands or even millions of records.

A **database** (A logical container that holds a group of files, often called tables, that together serve the information needs of an organization.) holds groups of files or **tables** (A database file that contains data in rows and columns; also, a term sometimes used by mathematicians to describe a two-dimensional array.) that together serve the information needs of an organization. Database software establishes and maintains relationships between fields in these tables, so that users can pull related data items together in a format that allows businesspeople to make managerial decisions efficiently. Chapter 14 of the comprehensive version of this text covers database creation.

Two Truths & A Lie

Understanding the Data Hierarchy

1. In the data hierarchy, a field is a single data item, such as `lastName`, `streetAddress`, or `annualSalary`.

   T F

2. In the data hierarchy, fields are grouped together to form a record; records are groups of fields that go together for some logical reason.

   T F

3. In the data hierarchy, related records are grouped together to form a field.

   T F

7-3 Performing File Operations

To use data files in your programs, you need to understand several file operations:

- Declaring a file
- Opening a file
- Reading from a file
- Writing to a file
- Closing a file

7-3a Declaring a File

Most languages support several types of files, but one way of categorizing files is by whether they can be used for input or for output. Just as variables and constants have data types such as `num` and `string`, each file has a data type that is defined in the language you are using. For example, a file's type might be `InputFile`. Just like variables and constants, files are declared by giving each a data type and an identifier. As examples, you might declare two files as follows:

```
InputFile employeeData
OutputFile updatedData
```

The `InputFile` and `OutputFile` types are capitalized in this book because their equivalents are capitalized in most programming languages. This approach helps to distinguish these complex types from simple types such as `num` and `string`. The identifiers given to files, such as `employeeData` and `updatedData`, are internal to the program, just as variable names are. To make a program read a file's data from a storage device, you also need to associate the

program's internal filename with the operating system's name for the file. Often, this association is accomplished when you open the file.

7-3b Opening a File

In most programming languages, before an application can use a data file, it must **open the file** (The process of locating a file on a storage device, physically preparing it for reading, and associating it with an identifier inside a program.) . Opening a file locates it on a storage device and associates a variable name within your program with the file. For example, if the identifier `employeeData` has been declared as type `InputFile`, then you might make a statement similar to the following:

```
open employeeData "EmployeeData.dat"
```

This statement associates the file named EmployeeData.dat on the storage device with the program's internal name `employeeData`. Usually, you also can specify a more complete path when the data file is not in the same directory as the program, as in the following:

```
open employeeData "C:\CompanyFiles\CurrentYear\EmployeeData.dat"
```


7-3c Reading Data from a File

Before you can use stored data within a program, you must load the data into computer memory. You never use the data values that are stored on a storage device directly. Instead, you use a copy that is transferred into memory. When you copy data from a file on a storage device into RAM, you **read from the file** (The act of copying data from a file on a storage device into RAM.) .

Especially when data items are stored on a hard disk, their location might not be clear to you—data just seems to be "in the computer." To a casual computer user, the lines between permanent storage and temporary memory are often blurred because many newer programs automatically save data for you periodically without asking your permission. However, at any moment in time, the version of a file in memory might differ from the version that was last saved to a storage device.

If data items have been stored in a file and a program needs them, you can write separate programming statements to input each field, as in the following example:

```
input name from employeeData
input address from employeeData
input payRate from employeeData
```

Most languages also allow you to write a single statement in the following format:

```
input name, address, payRate from employeeData
```

Most programming languages provide a way for you to use a group name for record data, as in the following statement:

```
input EmployeeRecord from employeeData
```

When this format is used, you need to define the separate fields that compose an `EmployeeRecord` when you declare the variables for the program.

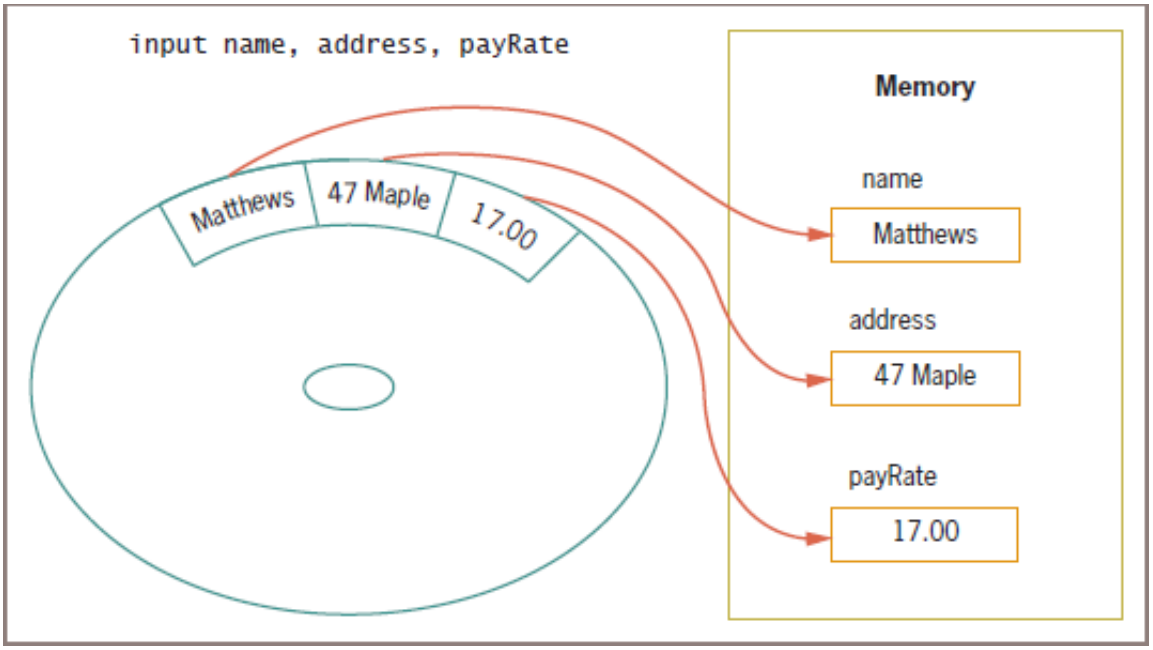
You usually do not want to input several items in a single statement when you read data from a keyboard, because you want to prompt the user for each item separately as you input it. However, when you retrieve data from a file, prompts are not needed. Instead, each item is retrieved in sequence and stored in memory at the appropriate named location.

The way a program knows how much data to input for each variable differs among programming languages. In many languages, a delimiter such as a comma is stored between data fields. In other languages, the amount of data retrieved depends on the data types of the variables in the input statement.

Figure 7-2 shows how an input statement works. When the input statement executes, each field is copied and placed in the appropriate variable in computer memory. Nothing on the disk indicates a field name associated with any of the data; the variable names exist within the program only. For example, another program could use the same file as input and call the fields `surname`, `street`, and `salary`.

Figure 7-2

Reading Three Data Items from a Storage Device into Memory



In some languages, you must specify the length of each field that is read from a data file. In other languages, the length of each field is determined by its data type.

When you read data from a file, you must read all the fields that are stored even though you might not want to use all of them. For example, suppose that you want to read an employee data file that contains names, addresses, and pay rates for each employee, and you want to output a list of names. Even though you are not concerned with the address or pay rate fields, you must read them into your program for each employee before you can get to the name for the next employee.

7-3d Writing Data to a File

When you store data in a computer file on a persistent storage device, you **write to the file** (The act of copying data from RAM to persistent storage.) . This means you copy data from RAM to the file. When you write data to a file, you write the contents of the fields using a statement such as the following:

```
output name, address, payRate to employeeData
```

When you write data to a file, you usually do not include explanations that make data easier for humans to interpret; you just write facts and figures. For example, you do not include column headings or write explanations such as *The pay rate is*, nor do you include

commas, dollar signs, or percent signs in numeric values. Those embellishments are appropriate for output on a monitor or on paper, but not for storage.

7-3e Closing a File

When you finish using a file, the program should **close the file** (An action that makes a file no longer available to an application.) —a closed file is no longer available to your application. Failing to close an input file (a file from which you are reading data) usually does not present serious consequences; the data still exists in the file. However, if you fail to close an output file (a file to which you are writing data), the data might not be saved correctly and might become inaccessible. You should always close every file you open, and you should close the file as soon as you no longer need it. When you leave a file open for no reason, you use computer resources, and your computer's performance suffers. Also, particularly within a network, another program might be waiting to use the file.

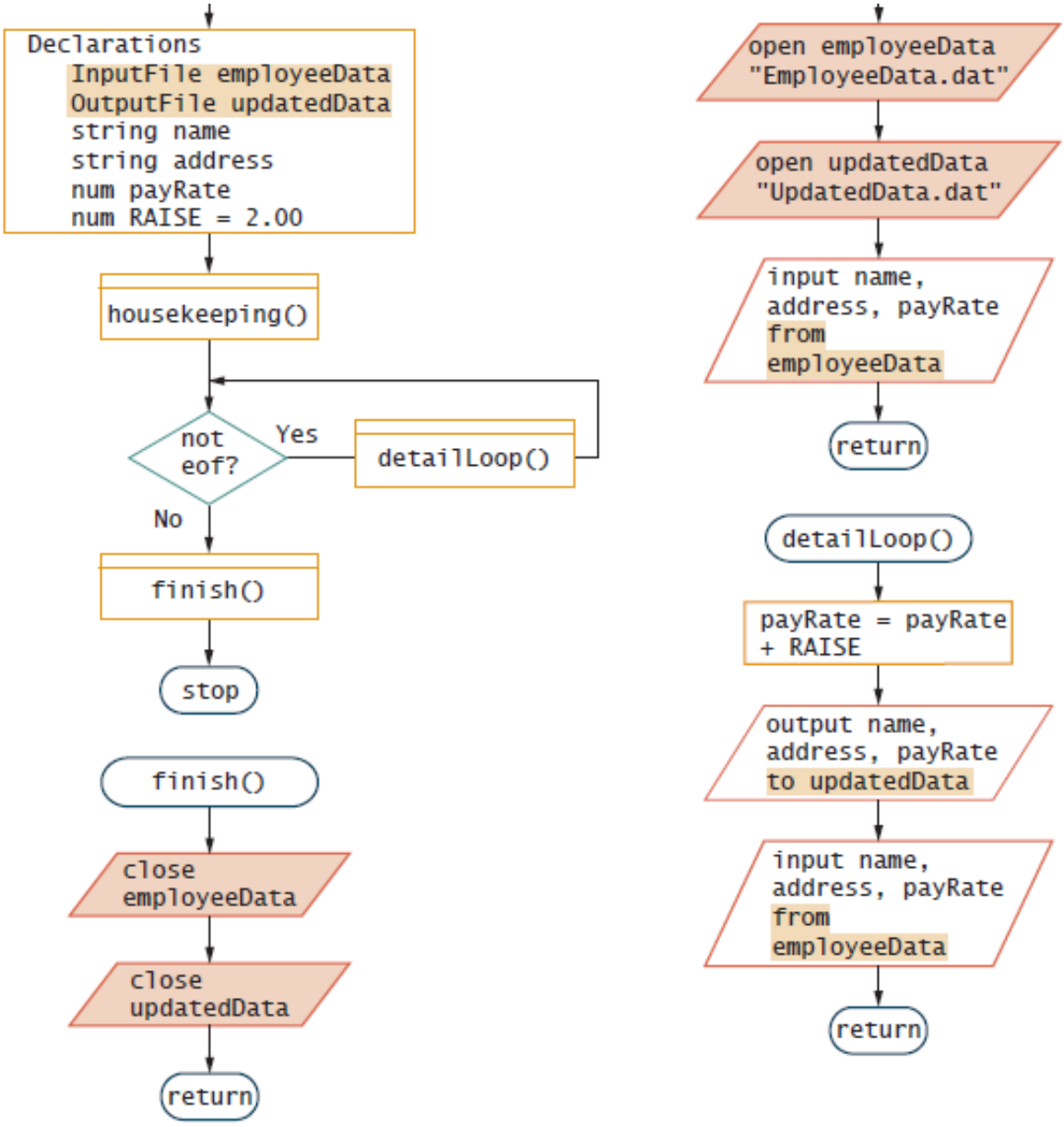
In most programming languages, if you read data from a keyboard or write it to the display monitor, you do not need to open or close the device. The keyboard and monitor are the **default input and output devices** (Hardware devices that do not require opening; usually they are the keyboard and monitor, respectively.) , respectively.

7-3f A Program That Performs File Operations

Figure 7-3 contains a program that opens two files, reads employee data from the input file, alters the employee's pay rate, writes the updated record to an output file, and closes the files. The statements that use the files are shaded. The convention in this book is to place file open and close statements in parallelograms in flowcharts, because they are operations closely related to input and output.

Figure 7-3

Flowchart and Pseudocode for Program That Uses Files




```
start
   Declarations
      InputFile employeeData
      OutputFile updatedData
      string name
      string address
      num payRate
      num RAISE = 2.00
   housekeeping()
   while not eof
      detailLoop()
   endwhile
   finish()
stop

housekeeping()
   open employeeData "EmployeeData.dat"
   open updatedData "UpdatedData.dat"
   input name, address, payRate from employeeData
return

detailLoop()
   payRate = payRate + RAISE
```

```
    output name, address, payRate to updatedData
    input name, address, payRate from employeeData
return

finish()
    close employeeData
    close updatedData
return
```

In the program in Figure 7-3, each employee's data is read into memory. Then the `payRate` variable in memory is increased by $2.00. The value of the pay rate on the input storage device is not altered. After the employee's pay rate is increased, the name, address, and newly altered pay rate values are stored in the output file. When processing is complete, the input file retains the original data and the output file contains the revised data. Many organizations would keep the original file as a backup file. A **backup file** (A copy that is kept in case values need to be restored to their original state.) is a copy that is kept in case values need to be restored to their original state. The backup copy is called a **parent file** (A copy of a file before revision.) and the newly revised copy is a **child file** (A copy of a file after revision.) .

Logically, the verbs *print*, *write*, and *display* mean the same thing—all produce output. However, in conversation, programmers usually reserve the word *print* for situations in which they mean *produce hard copy output*. Programmers are more likely to use *write* when talking about sending records to a data file and *display* when sending records to a monitor. In some programming languages, there is no difference in the verb used for output regardless of the hardware; you simply assign different output devices (such as printers, monitors, and disk drives) as needed to programmer-named objects that represent them.

Watch the video *File Operations*.

Throughout this book you have been encouraged to think of input as basically the same process, whether it comes from a user typing interactively at a keyboard or from a stored file on a disk or other media. The concept remains valid for this chapter, which discusses applications that commonly use stored file data. While data used by an application could be entered on a keyboard during program execution, this chapter assumes that data items have been entered, validated, and sorted earlier in another application, and then processed as input from files to achieve the results.

Sorting (The process of placing records in order by the value in a specific field or fields.) is the process of placing records in order by the value in a specific field or fields. Files can be

sorted manually before they are saved, or a program can sort them. You can learn about sorting techniques in Chapter 8 of the comprehensive version of this book; in this chapter, it is assumed that the record sorting process has already been accomplished.

Two Truths & A Lie

Performing File Operations

1. You give a file an internal name in a program and then associate it with the operating system's name for the file.

   T F

2. When you read from a file, you copy values from memory to a storage device.

   T F

3. If you fail to close an input file, usually no serious consequences will occur; the data still exists in the file.

   T F

7-4 Understanding Sequential Files and Control Break Logic

A **sequential file** (A file in which records are stored one after another in some order.) is a file in which records are stored one after another in some order. Frequently, records in a sequential file are organized based on the contents of one or more fields. Examples of sequential files include:

- A file of employees stored in order by ID number
- A file of parts for a manufacturing company stored in order by part number
- A file of customers for a business stored in alphabetical order by last name

7-4a Understanding Control Break Logic

A **control break** (A temporary detour in the logic of a program for special group processing.) is a temporary detour in the logic of a program. In particular, programmers use a **control break program** (A program in which a change in the value of a variable initiates special actions or processing.) when a change in a value initiates special actions or processing. You usually write control break programs to organize output for programs that handle data records organized logically in groups based on the value in a field or fields. As you read records, you examine the same field in each record, and when you encounter a record that contains a different value from the ones that preceded it, you perform a special action. For example, you might generate a report that lists all company clients in order by state of residence, with a count of clients after each state's client list. See Figure 7-4 for an example of a **control break report** (A report that lists items in groups. Frequently, each group is followed by a subtotal.) that breaks after each change in state.

Figure 7-4

A Control Break Report with Totals After Each State

Other examples of control break reports produced by control break programs could include:

- All employees listed in order by department number, with a new page started for each department
- All books for sale in a bookstore listed in order by category (such as reference or self-help), with a count following each category of book
- All items sold in order by date of sale, with a different ink color for each new month

Each of these reports shares two traits:

- The records used in each report are listed in order by a specific variable: state, department, category, or date.
- When that variable changes, the program takes special action: it starts a new page, prints a count or total, or switches ink color.

To generate a control break report, your input records must be organized in sequential order based on the field that will cause the breaks. In other words, to write a program that produces a report of customers by state, like the one in Figure 7-4, the records must be grouped by state before you begin processing. Frequently, this grouping will mean placing the records in alphabetical order by state, although they could just as easily be ordered by population, governor's name, or any other factor, as long as all of one state's records are together.

With some newer languages, such as SQL, the details of control breaks are handled automatically. Still, understanding how control break programs work improves your competence as a programmer.

Suppose that you have an input file that contains client names, cities, and states, and you want to produce a report like the one in Figure 7-4. The basic logic of the program works like this:

- Each time you read a client's record from the input file, you determine whether the client resides in the same state as the previous client.
- If so, you simply output the client's data, add 1 to a counter, and read another record, without any special processing. If there are 20 clients in a state, these steps are repeated 20 times in a row—read a client's data, count it, and output it.
- Eventually you will read a record for a client who is not in the same state. At that point, before you output the data for the first client in the new state, you must output the count for the previous state. You also must reset the counter to 0 so it is ready to start counting customers in the next state. Then you can proceed to handle client records for the new state, and you continue to do so until the next time you encounter a client from a different state.

This type of program contains a **single-level control break** (A break in the logic of a program based on the value of a single variable.) , a break in the logic of the program (in this case, pausing or detouring to output a count) that is based on the value of a single variable (in this case, the state). The technique you must use to "remember" the old state so

you can compare it with each new client's state is to create a special variable, called a **control break field** (A variable that holds the value that signals a special processing break in a program.) , to hold the previous state. As you read each new record, comparing the new and old state values determines when it is time to output the count for the previous state.

Figure 7-5 shows the mainline logic and `getReady()` module for a program that produces the report in Figure 7-4. In the mainline logic, the control break variable `oldState` is declared in the shaded statement. In the `getReady()` module, the report headings are output, the file is opened, and the first record is read into memory. Then, the state value in the first record is copied to the `oldState` variable. (See shading.) Note that it would be incorrect to initialize `oldState` when it is declared. When you declare the variables at the beginning of the main program, you have not yet read the first record; therefore, you don't know what the value of the first state will be. You might assume that it is *Alabama* because that is the first state alphabetically, and you might be right, but perhaps in this data set the first state is *Alaska* or even *Wyoming*. You are assured of storing the correct first state value if you copy it from the first input record.

Figure 7-5

Mainline Logic and `getready()` Module for the Program That Produces Clients by State Report

Within the `produceReport()` module in Figure 7-6, the first task is to check whether `state` holds the same value as `oldState`. For the first record, on the first pass through this method, the values are equal (because you set them to be equal right after getting the first input record in the `getReady()` module). Therefore, you proceed by outputting the first client's data, adding 1 to `count`, and inputting the next record.

Figure 7-6

The `producereport()` and `controlbreak()` Modules for the Program That

As long as each new record holds the same `state` value, you continue outputting, counting, and inputting, never pausing to output the count. Eventually, you will read in a client whose state is different from the previous one. That's when the control break occurs. Whenever a new state differs from the old one, three tasks must be performed:

- The count for the previous state must be output.
- The count must be reset to 0 so it can start counting records for the new state.
- The control break field must be updated.

When the `produceReport()` module receives a client record for which `state` is not the same as `oldState`, you cause a break in the normal flow of the program. The new client record must "wait" while the count for the just-finished state is output and `count` and the control break field `oldState` acquire new values.

The `produceReport()` module continues to output client names, cities, and states until the end of file is reached; then the `finishUp()` module executes. As shown in Figure 7-7, the

module that executes after processing the last record in a control break program must complete any required processing for the last group that was handled. In this case, the `finishUp()` module must display the count for the last state that was processed. After the input file is closed, the logic can return to the main program, where the program ends.

Figure 7-7

The `finishup()` Module for the Program That Produces Clients by State Report


Watch the video *Control Break Logic*.

Two Truths & A Lie

Understanding Sequential Files and Control Break Logic

1. In a control break program, a change in the value of a variable initiates special actions or processing.

   T F

2. When a control break variable changes, the program takes special action.

   T F

3. To generate a control break report, your input records must be organized in sequential order based on the first field in the record.

   T F

7-5 Merging Sequential Files

Businesses often need to merge two or more sequential files. **Merging files** (The act of combining two or more files while maintaining the sequential order.) involves combining two or more files while maintaining the sequential order. For example:

- Suppose that you have a file of current employees in ID number order and a file of newly hired employees, also in ID number order. You need to merge these two files into one combined file before running this week's payroll program.
- Suppose that you have a file of partsmanufactured in the Northside factory in part-number order and a file of partsmanufactured in the Southside factory, also in part-number order. You need to merge these two files into one combined file, creating a master list of available parts.
- Suppose that you have a file that lists last year's customers in alphabetical order and another file that lists this year's customers in alphabetical order. You want to create a mailing list of all customers in order by last name.

Before you can easily merge files, two conditions must be met:

- Each file must contain the same record layout.
- Each file used in the merge must be sorted in the same order based on the same field. **Ascending order** (Describes the arrangement of data items from lowest to highest.) describes records in order from lowest to highest values; **descending order** (Describes the arrangement of data items from highest to lowest.) describes records in order from highest to lowest values.

For example, suppose that your business has two locations, one on the East Coast and one on the West Coast, and each location maintains a customer file in alphabetical order by customer name. Each file contains fields for name and customer balance. You can call the fields in the East Coast file `eastName` and `eastBalance`, and the fields in the West Coast file `westName` and `westBalance`. You want to merge the two files, creating one combined file containing records for all customers. Figure 7-8 shows some sample data for the files; you want to create a merged file like the one shown in Figure 7-9.

Figure 7-8

Sample Data Contained in Two Customer Files

Figure 7-9

Merged Customer File

The mainline logic for a program that merges two files is similar to the main logic you've used before in other programs: It contains preliminary, housekeeping tasks; a detail module that repeats until the end of the program; and some clean-up, end-of-job tasks. However, most programs you have studied processed records until an `eof` condition was met, either because an input data file reached its end or because a user entered a sentinel value in an interactive program. In a program that merges two input files, checking for `eof` in only one of them is insufficient. Instead, the program can check a flag variable with a name such as `areBothAtEnd`. For example, you might initialize a string variable `areBothAtEnd` to `"N"`, but change its value to `"Y"` after you have encountered `eof` in both input files. (If the language you use supports a Boolean data type, you can use the values `true` and `false` instead of strings.)

Figure 7-10 shows the mainline logic for a program that merges the files shown in Figure 7-8. After the `getReady()` module executes, the shaded question that sends the logic to the `finishUp()` module tests the `areBothAtEnd` variable. When it holds `"Y"`, the program ends.

Figure 7-10

Mainline Logic of a Program That Merges Files

The `getReady()` module is shown in Figure 7-11. It opens three files—the input files for the east and west customers, and an output file in which to place the merged records. The program then reads one record from each input file. If either file has reached its end, the `END_NAME` constant is assigned to the variable that holds the file's customer name. The `getReady()` module then checks to see whether both files are finished (admittedly, a rare occurrence in the `getReady()` portion of the program's execution) and sets the `areBothAtEnd` flag variable to `"Y"` if they are. Assuming that at least one record is available, the logic then enters the `mergeRecords()` module.

Figure 7-11

The `getready()` Module for a Program That Merges Files, and the Methods It Calls

When you begin the `mergeRecords()` module in the program using the files shown in Figure 7-8, two records—one from `eastFile` and one from `westFile`—are sitting in the memory of the computer. One of these records needs to be written to the new output file first. Which one? Because the two input files contain records stored in alphabetical order, and you want the new file to store records in alphabetical order, you first output the input record that has the lower alphabetical value in the name field. Therefore, the process begins as shown in Figure 7-12.

Figure 7-12

Start of Merging Process

Using the sample data from Figure 7-8, you can see that the *Able* record from the East Coast file should be written to the output file, while Chen's record from the West Coast file waits in memory. The `eastName` value *Able* is alphabetically lower than the `westName` value *Chen*.

After you write Able's record, should Chen's record be written to the output file next? Not necessarily. It depends on the next `eastName` following Able's record in `eastFile`. When data records are read into memory from a file, a program typically does not "look ahead" to determine the values stored in the next record. Instead, a program usually reads the record into memory before making decisions about its contents. In this program, you need to read the next `eastFile` record into memory and compare it to *Chen*. Because in this case the next record in `eastFile` contains the name *Brown*, another `eastFile` record is written; no `westFile` records are written yet.

After the first two `eastFile` records, is it Chen's turn to be written now? You really don't know until you read another record from `eastFile` and compare its name value to *Chen*. Because this record contains the name *Dougherty*, it is indeed time to write Chen's record. After Chen's record is written, should you now write Dougherty's record? Until you read the next record from `westFile`, you don't know whether that record should be placed before or after Dougherty's record.

Therefore, the merging method proceeds like this: compare two records, write the record with the lower alphabetical name, and read another record from the *same* input file. See Figure 7-13.

Figure 7-13

Continuation of Merging Process

Recall the names from the two original files in Figure 7-8, and walk through the processing steps.

1

Compare *Able* and *Chen*. Write Able's record. Read Brown's record from `eastFile`.

2

Compare *Brown* and *Chen*. Write Brown's record. Read Dougherty's record from `eastFile`.

3

Compare *Dougherty* and *Chen*. WriteChen's record. Read Edgar's record from `westFile`.

4

Compare *Dougherty* and *Edgar*. Write Dougherty's record. Read Hanson's record from `eastFile`.

5

Compare *Hanson* and *Edgar*. Write Edgar's record. Read Fell's record from `westFile`.

6

Compare *Hanson* and *Fell*. Write Fell's record. Read Grand's record from `westFile`.

7

Compare *Hanson* and *Grand*. Write Grand's record. Read from `westFile`, encountering `eof`. This causes `westName` to be set to `END_NAME`.

What happens when you reach the end of the West Coast file? Is the program over? It shouldn't be because records for Hanson, Ingram, and Johnson all need to be included in the new output file, and none of them are written yet. Because the `westName` field is set to `END_NAME`, and `END_NAME` has a very high alphabetic value (*ZZZZZ*), each subsequent `eastName` will be lower than the value of `westName`, and the rest of the `eastName` file will be processed. With a different set of data, the `eastFile` might have ended first. In that case, `eastName` would be set to `END_NAME`, and each subsequent `westFile` record would be

processed.

Figure 7-14 shows the complete `mergeRecords()` module and the `finishUp()` module.

Figure 7-14

The `mergerecords()` and `finishup()` Modules for the File-merging Program

As the value for `END_NAME`, you might choose to use 10 or 20 Zs instead of only five. Although it is unlikely that a person will have the last name ZZZZZ, you should make sure that the high value you choose is actually higher than any legitimate value.

After Grand's record is processed, `westFile` is read and `eof` is encountered, so `westName` gets set to `END_NAME`. Now, when you enter the loop again, `eastName` and `westName` are compared, and `eastName` is still *Hanson*. The `eastName` value (*Hanson*) is lower than the `westName` value (*ZZZZZ*), so the data for `eastName`'s record is written to the output file, and another `eastFile` record (*Ingram*) is read.

The complete run of the file-merging program now executes the first six of the seven steps listed previously, and then proceeds as shown in Figure 7-14 and as follows, starting with a modified Step 7:

7

Compare *Hanson* and *Grand*. Write Grand's record. Read from `westFile`, encountering `eof` and setting `westName` to *ZZZZZ*.

8

Compare *Hanson* and *ZZZZZ*. Write Hanson's record. Read Ingram's record.

9

Compare *Ingram* and *ZZZZZ*. Write Ingram's record. Read Johnson's record.

10

Compare *Johnson* and *ZZZZZ*. Write Johnson's record. Read from `eastFile`, encountering `eof` and setting `eastName` to *ZZZZZ*.

11

Now that both names are *ZZZZZ*, set the flag `areBothAtEnd` equal to `"Y"`.

When the `areBothAtEnd` flag variable equals `"Y"`, the loop is finished, the files are closed, and the program ends.

If two names are equal during the merge process—for example, when there is a Hanson record in each file—then both Hansons will be included in the final file. When `eastName` and `westName` match, `eastName` is not lower than `westName`, so you write the `westFile` Hanson record. After you read the next `westFile` record, `eastName` will be lower than the next `westName`, and the `eastFile` Hanson record will be output. A more complicated merge program could check another field, such as first name, when last-name values match.

You can merge any number of files. To merge more than two files, the logic is only slightly more complicated; you must compare the key fields from all the files before deciding which file is the next candidate for output.

Watch the video *Merging Files*.

Two Truths & A Lie

Merging Sequential Files

1. A sequential file is a file in which records are stored one after another in some order. Most frequently, the records are stored based on the contents of one or more fields within each record.

   T F

2. Merging files involves combining two or more files while maintaining the sequential order.

   T F

3. Before you can easily merge files, each file must contain the same number of records.

   T F

7-6 Master and Transaction File Processing

In the last section, you learned how to merge related sequential files in which each record in each file contained the same fields. Some related sequential files, however, do not contain the same fields. Instead, some related files have a master-transaction relationship. A **master file** (A file that holds complete and relatively permanent data.) holds complete and relatively permanent data; a **transaction file** (A file that holds temporary data used to update a master file.) holds more temporary data. For example, a master customer file might hold customers' names, addresses, and phone numbers, and a customer transaction file might contain data that describes each customer's most recent purchase.

Commonly, you gather transactions for a period of time, store them in a file, and then use them one by one to update matching records in a master file. You **update the master file** (To modify the values in a master file based on transaction records.) by making appropriate changes to the values in its fields based on the recent transactions. For example, a file containing transaction purchase data for a customer might be used to update each balance due field in a customer record master file.

Here are a few other examples of files that have a master-transaction relationship:

- A library maintains a master file of all patrons and a transaction file with information about each book or other items checked out.
- A college maintains a master file of all students and a transaction file for each course registration.
- A telephone company maintains a master file for every telephone line (number) and a transaction file with information about every call.

When you update a master file, you can take two approaches:

- You can actually change the information in the master file. When you use this approach, the information that existed in the master file prior to the transaction processing is lost.
- You can create a copy of the master file, making the changes in the new version. Then, you can store the previous, parent version of the master file for a period of time, in case there are questions or discrepancies regarding the update process. The updated, child version of the file becomes the new master file used in subsequent processing. This approach is used in a program later in this chapter.

When a child file is updated, it becomes a parent, and its parent becomes a grandparent. Individual organizations create policies concerning the number of generations of backup files they will save before discarding them. The terms *parent* and *child* refer to file backup generations, but they are used for a different purpose in object-oriented programming. When you base a class on another using inheritance, the original class is the parent and the derived class is the child. You can learn about these concepts in Chapters 10 and Chapter 11 of the comprehensive version of this book.

The logic you use to perform a match between master and transaction file records is similar to the logic you use to perform a merge. As with a merge, you must begin with both files sorted in the same order on the same field. Figure 7-15 shows the mainline logic for a program that matches files. The master file contains a customer number, name, and a field that holds the total dollar amount of all purchases the customer has made previously. The transaction file holds data for sales, including a transaction number, the number of the customer who made the transaction, and the amount of the transaction.

Figure 7-15

The Mainline Logic for the Master-Transaction Program

Figure 7-16 contains the `housekeeping()` module for the program, and the modules it calls. These modules are very similar to their counterparts in the file-merging program earlier in the chapter. When the program begins, one record is read from each file. When any file ends, the field used for matching is set to a high value, 9999, and when both files end, a flag variable is set so the mainline logic can test for the end of processing. In the file-merging program presented earlier in this chapter, you placed the string `"ZZZZZ"` in the customer name field at the end of the file because string fields were being compared. In this example, because you are using numeric fields (customer numbers), you can store 9999 in them at the end of the file. The assumption is that 9999 is higher than any valid customer number.

Figure 7-16

The `housekeeping()` Module for the Master-Transaction Program, and the Modules It Calls

Imagine that you will update master file records by hand instead of using a computer program, and imagine that each master and transaction record is stored on a separate piece of paper. The easiest way to accomplish the update is to sort all the master records by customer number and place them in a stack, and then sort all the transactions by customer number (not transaction number) and place them in another stack. You then would examine the first transaction and look through the master records until you found a match. Any master records without transactions would be placed in a "completed" stack without changes. When a transaction matched a master record, you would correct the master record using the new transaction amount, and then go on to the next transaction. Of course, if there is no matching master record for a transaction, then you would realize an error had occurred, and you would probably set the transaction aside before continuing. The `updateRecords()` module works exactly the same way.

In the file-merging program presented earlier in this chapter, your first action in the program's detail loop was to determine which file held the record with the lower value; then, you wrote that record. In a matching program, you are trying to determine not only whether one file's comparison field is larger than another's; it's also important to know if they are *equal*. In this example, you want to update the master file record's `masterTotal` field only if the transaction record `transCustNum` field contains an exact match for the customer number in the master file record. Therefore, you compare `masterCustNum` from the master file and `transCustNum` from the transaction file. Three possibilities exist:

- The `transCustNum` value equals `masterCustNum`. In this case, you add `transAmount` to `masterTotal` and then write the updated master record to the output file. Then, you read in both a new master record and a new transaction record.

- The `transCustNum` value is higher than `masterCustNum`. This means a sale was not recorded for that customer. That's all right; not every customer makes a transaction every period, so you simply write the original customer record with exactly the same information it contained when input. Then, you get the next customer record to see if this customer made the transaction currently under examination.

- The `transCustNum` value is lower than `masterCustNum`. This means you are trying to apply a transaction for which no master record exists, so there must be an error, because a transaction should always have a master record. You can handle this error in a variety of ways; here, you will write an error message to an output device before reading the next transaction record. A human operator can then read the message

and take appropriate action.

The logic used here assumes that there can be only one transaction per customer. In the exercises at the end of this chapter, you will develop the logic for a program in which the customer can have multiple transactions.

Whether `transCustNum` was higher than, lower than, or equal to `masterCustNum`, after reading the next transaction or master record (or both), you check whether both `masterCustNum` and `transCustNum` have been set to 9999. When both are 9999, you set the `areBothAtEnd` flag to `"Y"`.

Figure 7-17 shows the `updateRecords()` module that carries out the logic of the file-matching process. Figure 7-18 shows some sample data you can use to walk through the logic for this program.

Figure 7-17

The `updaterecords()` Module for the Master-Transaction Program

Figure 7-18

Sample Data for the File-matching Program

The program proceeds as follows:

1. Read customer 100 from the master file and customer 100 from the transaction file. Customer numbers are equal, so 400.00 from the transaction file is added to 1000.00 in the master file, and a new master file record is written with a 1400.00 total sales figure. Then, read a new record from each input file.

2. The customer number in the master file is 102 and the customer number in the transaction file is 105, so there are no transactions today for customer 102. Write the master record exactly the way it came in, and read a new master record.

3. Now, the master customer number is 103 and the transaction customer number is still 105. This means customer 103 has no transactions, so you write the master record as is and read a new one.

4. Now, the master customer number is 105 and the transaction number is 105. Because customer 105 had a 75.00 balance and now has a 700.00 transaction, the new total sales figure for the master file is 775.00, and a new master record is written. Read one record from each file.

5. Now, the master number is 106 and the transaction number is 108. Write customer record 106 as is, and read another master.

6. Now, the master number is 109 and the transaction number is 108. An error has occurred. The transaction record indicates that you made a sale to customer 108, but there is no master record for customer number 108. Either the transaction is incorrect (there is an error in the transaction's customer number) or the transaction is correct but you have failed to create a master record. Either way, write an error message so that a clerk is notified and can handle the problem. Then, get a new transaction record.

7. Now, the master number is 109 and the transaction number is 110. Write master record 109 with no changes and read a new one.

8. Now, the master number is 110 and the transaction number is 110. Add the 400.00

transaction to the previous 500.00 balance in the master file, and write a new master record with 900.00 in the `masterTotal` field. Read one record from each file.

9. Because both files are finished, end the job. The result is a new master file in which some records contain exactly the same data they contained going in, but others (for which a transaction has occurred) have been updated with a new total sales figure. The original master and transaction files that were used as input can be saved for a period of time as backups.

Figure 7-19 shows the `finishUp()` module for the program. After all the files are closed, the updated master customer file contains all the customer records it originally contained, and each holds a current total based on the recent group of transactions.

Figure 7-19

The `finishup()` Module for the Master-Transaction Program

Two Truths & A Lie

Master and Transaction File Processing

1. You use a master file to hold temporary data related to transaction file records.

   T F

2. You use a transaction file to hold data that is used to update a master file.

   T F

3. The saved version of a master file is the parent file; the updated version is the child file.

T F

7-7 Random Access Files

The examples of files that have been written to and read from in this chapter are sequential access files, which means that you access the records in sequential order from beginning to end. For example, if you wrote an employee record with an ID number 234, and then you created a second record with an ID number 326, you would see when you retrieved the records that they remain in the original data-entry order. Businesses store data in sequential order when they use the records for **batch processing** (Processing that performs the same tasks with many records in sequence.) , or processing that involves performing the same tasks with many records, one after the other. For example, when a company produces paychecks, the records for the pay period are gathered in a batch and the checks are calculated and printed in sequence. It really doesn't matter whose check is produced first because none are distributed to employees until all have been printed. Batch processing usually implies some delay in processing. That is, records are gathered over a period of time and then processed together later. For example, when a company produces paychecks, the records might be gathered every day for two weeks before processing occurs.

Besides indicating a system that works with many records, the term *batch processing* can also mean a system in which you issue many operating system commands as a group.

For many applications, sequential access is inefficient. These applications, known as **real-time** (Describes applications that require a record to be accessed immediately while a client is waiting.) applications, require that a record be accessed immediately while a client is waiting. A program in which the user makes direct requests is an **interactive program** (A program in which a user makes direct requests, as opposed to one in which input comes from a file.) . For example, if a customer telephones a department store with a question about a monthly bill, the customer service representative does not need or want to access every customer account in sequence. With tens of thousands of account records to read, it would take too long to access the customer's record. Instead, customer service representatives require **random access files** (Files that contain records that can be located

in any order.) , files in which records can be located in any order. Files in which records must be accessed immediately are also called **instant access files** (Random access files in which records must be accessed immediately.) . Because they enable you to locate a particular record directly (without reading all of the preceding records), random access files are also called **direct access files** (Random access files.) . You can declare a random access file with a statement similar to the following:

You associate this name with a stored file in the same manner that you associate an identifier with sequential input and output files. You also can use read, write, and close operations with a random access file. However, with random access files you have the additional capability to find a record directly. For example, you might be able to use a statement similar to the following to find customer number 712 on a random access file:

This feature is particularly useful in random access processing. Consider a business with 20,000 customer accounts. When the customer who has the 14,607th record in the file acquires a new telephone number, it is convenient to access the 14,607th record directly, writing the new telephone number to the file in the location where the old number was previously stored.

Two Truths & A Lie

Random Access Files

1. A batch program usually uses instant access files.

   T F

2. In a real-time application, a record is accessed immediately while a client is waiting.

   T F

3. An interactive program usually uses random access files.

   T F

7-8 Chapter Review

7-8a Chapter Summary

- A computer file is a collection of data stored on a nonvolatile device in a computer system. Although the contents of files differ, each file occupies space on a section of a storage device, and each has a name and specific times associated with it. Computer files are organized in directories or folders. A file's complete list of directories is its path.

- Data items in a file usually are stored in a hierarchy. Characters are letters, numbers, and special symbols, such as *A*, *7*, and *$*. Fields are data items that represent a single attribute of a record and are composed of one or more characters. Records are groups of fields that go together for some logical reason. Files are groups of related records.

- When you use a data file in a program, you must declare the file and open it; opening a file associates an internal program identifier with the name of a physical file on a storage device. When you read from a file, the data is copied into memory. When you write to a file, the data is copied from memory to a storage device. When you are done with a file, you close it.

- A sequential file is a file in which records are stored one after another in some order. A control break program is one that reads a sequential file and performs special processing based on a change in one or more fields in each record in the file.

- Merging files involves combining two or more files while maintaining the sequential order.

- Some related sequential files are master files that hold relatively permanent data, and transaction files that hold more temporary data. Commonly, you gather transactions for a period of time, store them in a file, and then use them one by one to update matching records in a master file.

- Real-time, interactive applications require random access files in which records can be located in any order. Files in which records must be accessed immediately are also called instant access files and direct access files.

Chapter Review

7-8b Key Terms

computer file (A collection of data stored on a nonvolatile device in a computer system.)

permanent storage devices (Hardware devices that hold nonvolatile data; examples include hard disks, DVDs, Zip disks, USB drives, and reels of magnetic tape.)

Text files (Files that contain data that can be read in a text editor.)

Binary files (Files that contain data that has not been encoded as text.)

byte (A unit of computer storage equal to eight bits.)

kilobytes (Approximately 1000 bytes.)

megabytes (A million bytes.)

gigabytes (A billion bytes.)

Directories (Organization units on storage devices; each can contain multiple files as well as additional directories. In a graphical interface system, directories are often called *folders*.)

folders (Organization units on storage devices; each can contain multiple files as well as additional folders. Folders are graphic directories.)

path (The combination of a file's disk drive and the complete hierarchy of directories in which the file resides.)

data hierarchy (Represents the relationship of databases, files, records, fields, and characters.)

Characters (A letter, number, or special symbol such as *A*, *7*, or *$*.)

Fields (A single data item, such as `lastName`, `streetAddress`, or `annualSalary`.)

Records (A group of fields stored together as a unit because they hold data about a single entity.)

Files (A group of records that go together for some logical reason.)

database (A logical container that holds a group of files, often called tables, that together serve the information needs of an organization.)

tables (A database file that contains data in rows and columns; also, a term sometimes used by mathematicians to describe a two-dimensional array.)

open the file (The process of locating a file on a storage device, physically preparing it for reading, and associating it with an identifier inside a program.)

read from the file (The act of copying data from a file on a storage device into RAM.)

write to the file (The act of copying data from RAM to persistent storage.)

close the file (An action that makes a file no longer available to an application.)

default input and output devices (Hardware devices that do not require opening; usually they are the keyboard and monitor, respectively.)

backup file (A copy that is kept in case values need to be restored to their original state.)

parent file (A copy of a file before revision.)

child file (A copy of a file after revision.)

Sorting (The process of placing records in order by the value in a specific field or fields.)

sequential file (A file in which records are stored one after another in some order.)

control break (A temporary detour in the logic of a program for special group processing.)

control break program (A program in which a change in the value of a variable initiates special actions or processing.)

control break report (A report that lists items in groups. Frequently, each group is followed by a subtotal.)

single-level control break (A break in the logic of a program based on the value of a single variable.)

control break field (A variable that holds the value that signals a special processing break in a program.)

Merging files (The act of combining two or more files while maintaining the sequential order.)

Ascending order (Describes the arrangement of data items from lowest to highest.)

descending order (Describes the arrangement of data items from highest to lowest.)

master file (A file that holds complete and relatively permanent data.)

transaction file (A file that holds temporary data used to update a master file.)

update the master file (To modify the values in a master file based on transaction records.)

batch processing (Processing that performs the same tasks with many records in sequence.)

real-time (Describes applications that require a record to be accessed immediately while a client is waiting.)

interactive program (A program in which a user makes direct requests, as opposed to

one in which input comes from a file.)

random access files (Files that contain records that can be located in any order.)

instant access files (Random access files in which records must be accessed immediately.)

direct access files (Random access files.)

Chapter Review

7-8c Review Questions

1. Random access memory is ____ .
    a. permanent
    b. volatile
    c. persistent
    d. continual
2. Which is true of text files?
    a. Text files contain data that can be read in a text editor.
    b. Text files commonly contain images and music.
    c. both of the above
    d. none of the above
3. Every file on a storage device has a ____ .
    a. name
    b. size
    c. both of the above
    d. none of the above

4. Which of the following is true regarding the data hierarchy?
    a. Files contain records.
    b. Characters contain fields.
    c. Fields contain files.
    d. Fields contain records.
5. The process of ____ a file locates it on a storage device and associates a variable name within your program with the file.
    a. opening
    b. closing
    c. declaring
    d. defining
6. When you write to a file, you ____ .
    a. move data from a storage device to memory
    b. copy data from a storage device to memory
    c. move data from memory to a storage device
    d. copy data from memory to a storage device
7. Unlike when you print a report, when a program's output is a data file, you do not ____ .
    a. include headings or other formatting
    b. open the files
    c. include all the fields represented as input
    d. all of the above
8. When you close a file, it ____ .
    a. is no longer available to the program
    b. cannot be reopened
    c. becomes associated with an internal identifier

d. ceases to exist

9. A file in which records are stored one after another in some order is a(n) ____ file.

   a. temporal

   b. sequential

   c. random

   d. alphabetical

10. When you combine two or more sorted files while maintaining their sequential order based on a field, you are ____ the files.

    a. tracking

    b. collating

    c. merging

    d. absorbing

11. A control break occurs when a program ____ .

    a. takes one of two alternate courses of action for every record

    b. ends prematurely, before all records have been processed

    c. pauses to perform special processing based on the value of a field

    d. passes logical control to a module contained within another program

12. Which of the following is an example of a control break report?

    a. a list of all customers of a business in zip code order, with a count of the number of customers who reside in each zip code

    b. a list of all students in a school, arranged in alphabetical order, with a total count at the end of the report

    c. a list of all employees in a company, with a “Retain” or “Dismiss” message following each employee record

    d. a list of medical clinic patients who have not seen a doctor for at least two years

13. A control break field ____ .

    a. always is output prior to any group of records on a control break report

    b. always is output after any group of records on a control break report

    c. never is output on a report

    d. causes special processing to occur

14. Whenever a control break occurs during record processing in any control break program, you must ____ .

    a. declare a control break field

    b. set the control break field to 0

    c. update the value in the control break field

    d. output the control break field

15. Assume that you are writing a program to merge two files named FallStudents and SpringStudents. Each file contains a list of students enrolled in a programming logic course during the semester indicated, and each file is sorted in student ID number order. After the program compares two records and subsequently writes a Fall student to output, the next step is to ____ .

    a. read a SpringStudents record

    b. read a FallStudents record

    c. write a SpringStudents record

    d. write another FallStudents record

16. When you merge records from two or more sequential files, the usual case is that the records in the files ____ .

    a. contain the same data

    b. have the same format

    c. are identical in number

    d. are sorted on different fields

17. A file that holds more permanent data than a transaction file is a ____ file.

a. master

b. primary

c. key

d. mega-

18. A transaction file is often used to ____ another file.

a. augment

b. remove

c. verify

d. update

19. The saved version of a file that does not contain the most recently applied transactions is known as a ____ file.

a. master

b. child

c. parent

d. relative

20. Random access files are used most frequently in all of the following except ____ .

a. interactive programs

b. batch processing

c. real-time applications

d. programs requiring direct access

Chapter Review

7-8d Exercises

Your downloadable files for Chapter 7 include one or more comma-delimited sample data files for each exercise in this section and the Game Zone section. You might want to use these files in any of several ways:

- You can look at the file contents to better understand the types of data each program uses.
- You can use the file contents as sample data when you desk-check the logic of your flowcharts or pseudocode.
- You can use the files as input files if you implement the solutions in a programming language and write programs that accept file input.
- You can use the data as guides for entering appropriate values if you implement the solutions in a programming language and write interactive programs.

1. The Vernon Hills Mail Order Company often sends multiple packages per order. For each customer order, output a mailing label for each box to be mailed. The mailing labels contain the customer's complete name and address, along with a box number in the form *Box 9 of 9*. For example, an order that requires three boxes produces three labels: *Box 1 of 3*, *Box 2 of 3*, and *Box 3 of 3*. Design an application that reads records that contain a customer's title (for example, *Mrs.*), first name, last name, street address, city, state, zip code, and number of boxes. The application must read the records until `eof` is encountered and produce enough mailing labels for each order.
2. The Cupid Matchmaking Service maintains two files—one for male clients and another for female clients. Each file contains a client ID, last name, first name, and address. Each file is in client ID number order. Design the logic for a program that merges the two files into one file containing a list of all clients, maintaining ID number order.
3. Laramie Park District has files of participants in its summer and winter programs this year. Each file is in participant ID number order and contains additional fields for first name, last name, age, and class taken (for example, *Beginning Swimming*).
    a. Design the logic for a program that merges the files for summer and winter programs to create a list of the first and last names of all participants for the year.

b. Modify the program so that if a participant has more than one record, you output the participant's name only once.

c. Modify the program so that if a participant has more than one record, you output the name only once, but you also output a count of the total number of classes the participant has taken.

4. The Apgar Medical group keeps a patient file for each doctor in the office. Each record contains the patient's first and last name, home address, and birth year. The records are sorted in ascending birth year order. Two doctors, Dr. Best and Dr. Cushing, have formed a partnership. Design the logic that produces a merged list of their patients in ascending order by birth year.

5. The Martin Weight Loss Clinic maintains two patient files—one for male clients and one for female clients. Each record contains the name of a patient and current total weight loss in pounds. Each file is in descending weight loss order. Design the logic that merges the two files to produce one combined file in order by weight loss.

6. a. The Curl Up and Dye Beauty Salon maintains a master file that contains a record for each of its clients. Fields in the master file include the client's ID number, first name, last name, and total amount spent this year. Every week, a transaction file is produced. It contains a customer's ID number, the service received (for example, *Manicure*), and the price paid. Each file is sorted in ID number order. Design the logic for a program that matches the master and transaction file records and updates the total paid for each client by adding the current week's price paid to the cumulative total. Not all clients purchase services each week. The output is the updated master file and an error report that lists any transaction records for which no master record exists.

b. Modify the program to output a coupon for a free haircut each time a client exceeds $750 in services. The coupon, which contains the client's name and an appropriate congratulatory message, is output during the execution of the update program when a client total surpasses $750.

7. a. The Timely Talent Temporary Help Agency maintains an employee master file that contains an employee ID number, last name, first name, address, and hourly rate for each temporary worker. The file has been sorted in employee ID number order. Each week, a transaction file is created with a job number, address, customer name, employee ID, and hours worked for every job filled by Timely Talent workers. The

transaction file is also sorted in employee ID order. Design the logic for a program that matches the master and transaction file records, and output one line for each transaction, indicating job number, employee ID number, hours worked, hourly rate, and gross pay. Assume that each temporary worker works at most one job per week; output one line for each worker who has worked that week.

b. Modify the help agency program so that any temporary worker can work any number of separate jobs in a week. Print one line for each job that week.

c. Modify the help agency program so that it accumulates the worker's total pay for all jobs in a week and outputs one line per worker.

Find the Bugs

8. Your downloadable files for Chapter 7 include DEBUG07-01.txt, DEBUG07-02.txt, and DEBUG07-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

9. The International Rock Paper Scissors Society holds regional and national championships. Each region holds a semifinal competition in which contestants play 500 games of Rock Paper Scissors. The top 20 competitors in each region are invited to the national finals. Assume that you are provided with files for the East, Midwest, and Western regions. Each file contains the following fields for the top 20 competitors: last name, first name, and number of games won. The records in each file are sorted in alphabetical order. Merge the three files to create a file of the top 60 competitors who will compete in the national championship.

10. In the Game Zone section of Chapter 5, you designed a guessing game in which the application generates a random number and the player tries to guess it. After each guess, you displayed a message indicating whether the player's guess was correct, too high, or too low. When the player eventually

guessed the correct number, you displayed a score that represented a count of the number of required guesses. Modify the game so that when it starts, the player enters his or her name. After a player plays the game exactly five times, save the best (lowest) score from the five games to a file. If the player's name already exists in the file, update the record with the new lowest score. If the player's name does not already exist in the file, create a new record for the player. After the file is updated, display all the best scores stored in the file.

Up for Discussion

11. Suppose that you are hired by a police department to write a program that matches arrest records with court records detailing the ultimate outcome or verdict for each case. You have been given access to current files so that you can test the program. Your friend works in the personnel department of a large company and must perform background checks on potential employees. (The job applicants sign a form authorizing the check.) Police records are open to the public and your friend could look up police records at the courthouse, but it would take many hours per week. As a convenience, should you provide your friend with outcomes of any arrest records of job applicants?

12. Suppose that you are hired by a clinic to match a file of patient office visits with patient master records to print various reports. While working with the confidential data, you notice the name of a friend's fiancé. Should you tell your friend that the fiancé is seeking medical treatment? Does the type of treatment affect your answer?