CHAPTER 7

# Introduction to Sequential Files

An important characteristic of a programming language designed for enterprise or business computing is that it should have an external, rather than an internal focus. It should concentrate on processing data held externally in files and databases rather than on manipulating data in memory through linked lists, trees, stacks, and other sophisticated data structures. Whereas in most programming languages the focus is internal, in COBOL it is external. A glance at the table of contents of any programming book on Java, C, Pascal, or Ruby emphasizes the point. In most cases, only one chapter, if that, is devoted to files. In this book, over a quarter of the book deals with files: it covers such topics as sequential files, relative files, indexed files, the `SORT`, the `MERGE`, the Report Writer, control breaks, and the file-update problem.

COBOL supports three file organizations: sequential files, relative files, and indexed files. Relative and indexed are direct-access file organizations that are discussed later in the book. They may be compared to a music CD on which you select the track you desire. Sequential files are like a music cassette: to listen to a particular song, you must go through all the preceding songs.

This chapter provides a gentle introduction to sequential files. I introduce some of the terminology used when referring to files and explain how sequential files are organized and processed. Every COBOL file organization requires entries in the `INPUT-OUTPUT SECTION` of the `ENVIRONMENT DIVISION` and the `FILE SECTION` of the `DATA DIVISION`, and these declarations are specified and explained. Because files require more sophisticated data definition than the elementary data items introduced in Chapter 3, this chapter also introduces hierarchically structured data definitions.

## What Is a File?

A *file* is a repository for data that resides on backing storage (hard disk, magnetic tape, or CD-ROM). Nowadays, files are used to store a variety of different types of information such as programs, documents, spreadsheets, videos, sounds, pictures, and record-based data. In a record-based file, the data is organized into discrete packages of information. For instance, a customer record holds information about a customer such as their identifying number, name, address, date of birth, and gender. A customer file may contain thousands or even millions of instances of the customer record. In a picture file or music file, by way of contrast, the information is essentially an undifferentiated stream of bytes.

COBOL is often used in systems where the volume of data to be processed is large—not because the data is inherently voluminous, as it is in video or sound files, but because the same items of information have been recorded about a great many instances of the same object. Although COBOL can be used to process other kinds of data files, it is generally used only to process record-based files.

There are essentially two types of record-based file organization—serial files (COBOL calls these sequential files) and direct-access files:

- In a serial file, the records are organized and accessed serially (one after another).
- In a direct-access file, the records are organized in a manner that allows direct access to a particular record based on a key value. Unlike serial files, a record in a direct-access file can be accessed without having to read any of the preceding records.

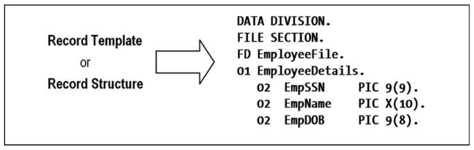
## Terminology

Before I discuss sequential files, I need to introduce some terminology:

- *Field*: An item of information that you are recording about an object (`StockNumber`, `SupplierCode`, `DateOfBirth`, `ValueOfSale`)
- *Record*: The collection of fields that record information about an object (for example, a `CustomerRecord` is a collection of fields recording information about a customer)
- *File*: A collection of one or more occurrences (instances) of a record template (structure)

## Files, Records, and Fields

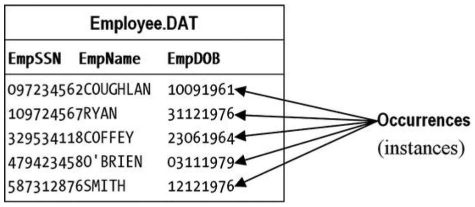
It is important to distinguish between the record *occurrence* (the instance or values of a record) and the record *template* (the structure of the record). Every record in a file has a different value but the same structure. For instance, the record template illustrated in Figure 7-1 describes the structure of each record occurrence (instance).



```
DATA DIVISION.
FILE SECTION.
FD EmployeeFile.
01 EmployeeDetails.
   02  EmpSSN     PIC 9(9).
   02  EmpName    PIC X(10).
   02  EmpDOB     PIC 9(8).
```

***Figure 7-1.*** *Record template/structure*

The occurrences of the employee records (Figure 7-2) are the actual values in the file. There is only one record template, but there are many record instances.


Employee.DAT
Occurrences
(instances)

```
EmpSSN   EmpName    EmpDOB
097234562COUGHLAN   10091961
109724567RYAN       31121976
329534118COFFEY     23061964
479423458O'BRIEN    03111979
587312876SMITH      12121976
```

***Figure 7-2.*** *Record occurrences/instances*

## How Files Are Processed

Before a computer can process a piece of data, the data must be loaded into the computer's main memory (RAM). For instance, if you want to manipulate a picture in Photoshop or edit a file in Word, you have to load the data file into main memory (RAM), make the changes you want, and then save the file on to backing storage (disk).

Programmers in other languages, who may not be used to processing record-based data, often seek to load the entire file into memory as if it were an undifferentiated stream of bytes. For record-based data, this is inefficient and consumes unnecessary computing resources.

A record-based file may consist of millions, tens of millions, or even hundreds of millions of records and may require gigabytes of storage. For instance, suppose you want to keep some basic census information about all the people in the United States. Suppose that each record is about 1,000 characters/bytes (1KB) in size. If you estimate the population of the United States at 314 million, this gives you a size for the file of 1,000 × 314,000,000 = 314,000,000,000 bytes = 314GB. Most computers do not have 314GB of RAM available, and those that do are unlikely to be standalone machines running only your program. The likelihood is that your program is only one of many running on the machine at the same time. If your program is found to be using a substantial proportion of the available RAM, your manager is going to be less than gruntled.

**Note** I once asked an M.Sc. student who was a proficient C++ programmer to write the C++ equivalent of a COBOL file processing program I had written. His first action was to load the entire file into memory. Doing this used an inordinate amount of memory and offered no benefit. He still had to read the file from disk, and the file size so overwhelmed the available RAM that the virtual memory manager had to keep paging to disk.

The data in a record-based file consists of discrete packages of information (records). The correct way to process such a file is to load a record into RAM, process it, and then load the next record. To store the record in memory and allow access to its individual fields, you must declare the record structure (Figure 7-1) in your program. The computer uses your description of the record (the record template) to set aside sufficient memory to store one instance of the record.

The memory allocated for storing a record is usually called a *record buffer*. To process a file, a program reads the records, one at a time, into the record buffer, as shown in Figure 7-3. The record buffer is the only connection between the program and the records in the file.

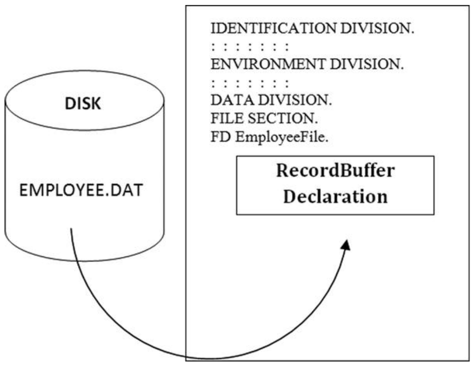


*Figure 7-3. Reading records into the record buffer*

## Implications of Buffers

If your program processes more than one file, you have to describe a record buffer for each file. To process all the records in an *input* file, each record instance must be copied (read) from the file into the record buffer when required. To create an *output* file, each record must be placed in the record buffer and then transferred (written) to the file. To transfer a record from an input file to an output file, your program will have to do the following:

- Read the record into the input record buffer.
- Transfer it to the output record buffer.
- Write the data to the output file from the output record buffer.

This type of data transfer between buffers is common in COBOL programs.

## File and Record Declarations

Suppose you want to create a file to hold information about your employees. What kind of information do you need to store about each employee?

One thing you need to store is the employee's Name. Each employee is also assigned a unique Social Security Number (SSN), so you need to store that as well. You also need to store the employee's date of birth and gender.

These fields are summarized here:

- Employee SSN
- Employee Name
- Employee DOB
- Employee Gender

---

**Note** This is for demonstration only. In reality, you would need to include far more items than these.

---

Creating a Record

To create a record buffer large enough to store one instance of the employee record you must decide on the type and size of each of the fields:

- Employee SSN is nine digits in size, so the data item to hold it is declared as `PIC 9(9)`.
- To store Employee Name, you can assume that you require only 25 characters. So the data item can be declared as `PIC X(25)`.
- Employee Date of Birth requires eight digits, so you can declare it as `PIC 9(8)`.
- Employee Gender is represented by a one-letter character, where *m* is male and *f* is female, so it can be declared as `PIC X`.

These fields are individual data items, but they are collected together into a record structure as shown in Example 7-1.

***Example 7-1***. The `EmployeeDetails` Record Description/Template

```
01 EmployeeDetails.
   02  EmpSSN          PIC 9(9).
   02  EmpName         PIC X(25).
   02  EmpDateOfBirth  PIC 9(8).
   02  EmpGender       PIC X.
```

This record description reserves the correct amount of storage for the record buffer, but it does not allow access to all the individual parts of the record that might be of interest.

For instance, the name is actually made up of the employee's surname and forename. And the date consists of four digits for the year, two digits for the month, and two digits for the day. To be able to access these fields individually, you need to declare the record as shown in Example 7-2.

***Example 7-2***. A More Granular Version of the `EmployeeDetails` Record

```
01 EmployeeDetails.
   02  EmpSSN          PIC 9(9).
   02  EmpName.
       03 EmpSurname   PIC X(15).
       03 EmpForename  PIC X(10).
   02  EmpDateOfBirth.
       03 EmpYOB       PIC 9(4).
       03 EmpMOB       PIC 99.
       03 EmpDOB       PIC 99.
   02  EmpGender       PIC X.
```

Declaring the Record Buffer in Your Program

The record description in Example 7-2 sets aside sufficient storage to store one instance of the employee record. This area of storage is the record buffer; it's the only connection between the program and the records in the file. To process the file, you must read the records from the file, one at a time, into the record buffer. The record buffer is connected to the file that resides on backing storage by declarations made in the `FILE SECTION` of the `DATA DIVISION` and the `SELECT` and `ASSIGN` clause of the `ENVIRONMENT DIVISION`.

A record template (description/buffer) for every file used in a program must be described in the `FILE SECTION` by means of an `FD` (file description) entry. The `FD` entry consists of the letters `FD` and an internal name that you assign to the file. The full file description for the employee file might be as shown in Example 7-3.

***Example 7-3***. The `DATA DIVISION` Declarations for the Employee File.

```
DATA DIVISION.
FILE SECTION.
FD EmployeeFile.
01 EmployeeDetails.
   02  EmpSSN          PIC 9(9).
   02  EmpName.
       03 EmpSurname   PIC X(15).
       03 EmpForename  PIC X(10).
   02  EmpDateOfBirth.
       03 EmpYOB       PIC 9(4).
       03 EmpMOB       PIC 99.
       03 EmpDOB       PIC 99.
   02  EmpGender       PIC X.
```

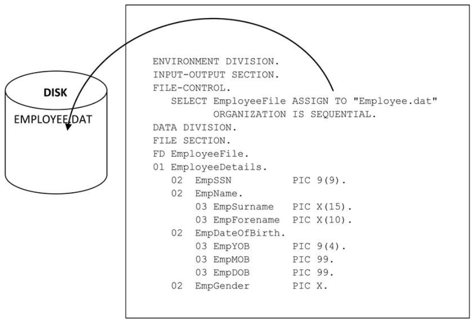
In this example, the name `EmployeeFile` has been assigned as the internal name for the file. This name is then used in the program for file operations such as these:

```
OPEN INPUT EmployeeFile
READ EmployeeFile
CLOSE EmployeeFile
```

The SELECT and ASSIGN Clause

Although you are going to refer to the employee file as `EmployeeFile` in the program, the actual name of the file on disk is `Employee.dat`. To connect the name used in the program to the file's actual name on backing storage, you require entries in the `SELECT` and `ASSIGN` clause of the `FILE-CONTROL` paragraph, in the `INPUT-OUTPUT SECTION` of the `ENVIRONMENT DIVISION`. As shown in Example 7-4, the `SELECT` and `ASSIGN` clause allows you to specify that an internal file name is to be connected to an external data resource. It also lets you specify how the file is organized. In the case of a sequential file, you specify that the file organization is sequential. Sequential files are ordinary text files such as you might create with a text editor.

***Example 7-4***. Using `SELECT` and `ASSIGN`



```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT EmployeeFile ASSIGN TO "Employee.dat"
          ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD EmployeeFile.
01 EmployeeDetails.
   02  EmpSSN          PIC 9(9).
   02  EmpName.
       03 EmpSurname   PIC X(15).
       03 EmpForename  PIC X(10).
   02  EmpDateOfBirth.
       03 EmpYOB       PIC 9(4).
       03 EmpMOB       PIC 99.
       03 EmpDOB       PIC 99.
   02  EmpGender       PIC X.
```

SELECT and ASSIGN Syntax

Here is the `SELECT` and `ASSIGN` syntax:

```
SELECT InternalFileName
    ASSIGN TO ExternalFileSpecification
    [[ORGANIZATION IS ] SEQUENTIAL]
```

**Note** The `SELECT` and `ASSIGN` clause has far more entries (even for sequential files) than those shown here. I deal with these entries in this book as you require them.

As illustrated by the examples in Example 7-5, `ExternalFileSpecification` can be either an identifier or a literal. The identifier or literal can consist of a simple file name or a full or partial file specification. If you use a simple file name, the drive and directory where the program is running are assumed.

When you use a literal, the file specification is hard-coded into the program; but if you want to specify the name of a file when you run the program, you can use an identifier. If an identifier is used, you must move the actual file specification into the identifier before the file is opened.

***Example 7-5***. Some Example `SELECT` and `ASSIGN` Declarations

```
SELECT EmployeeFile
    ASSIGN TO "D:\Cobol\ExampleProgs\Employee.Dat"
    ORGANIZATION IS SEQUENTIAL.

SELECT EmployeeFile
    ASSIGN TO "Employee.Dat"
    ORGANIZATION IS SEQUENTIAL.

SELECT EmployeeFile
    ASSIGN TO EmployeeFileName
    ORGANIZATION IS SEQUENTIAL.
:  :  :  :  :  :  :  :  :  :  :  :
MOVE "C:\datafiles\Employee.dat" TO EmployeeFileName
OPEN INPUT EmployeeFile
```

**EXTENDED SELECT AND ASSIGN**

I mentioned that sequential files are ordinary text files such as might be created with a text editor. This is not entirely true. A text editor appends the Carriage Return (CR) and Line Feed (LF) characters to each line of text. If you specify `ORGANIZATION IS SEQUENTIAL` and create your test data as lines of text in an ordinary text editor, these extra characters will be counted, and this will throw your records off by two characters each time you read a new record. For this reason, some vendors have extended `SELECT` and `ASSIGN` to allow these line-terminating characters to be either ignored or included. For instance, in Micro Focus COBOL, the metalanguage for the `SELECT` and `ASSIGN` is

$$\begin{array}{l}\underline{\text{SELECT}}\ \text{InternalFileName}\\ \quad \underline{\text{ASSIGN}}\ \text{TO ExternalFileSpecification}\\ \quad \left[\left[\underline{\text{ORGANIZATION}}\ \text{IS}\right]\left[\begin{array}{l}\underline{\text{LINE SEQUENTIAL}}\\ \underline{\text{RECORD SEQUENTIAL}}\end{array}\right]\right]\end{array}$$

Here `LINE SEQUENTIAL` means the CR and LF characters are not considered part of the record, and `RECORD SEQUENTIAL` means they are (same as the standard `SEQUENTIAL`).

Because it is very convenient to be able to use an ordinary text editor to create test data files, I use the Micro Focus `LINE SEQUENTIAL` extension in the example programs.

**Processing Sequential Files**

Unlike direct-access files, sequential files are uncomplicated both in organization and in processing. To write programs that process sequential files, you only need to know four new verbs: `OPEN`, `CLOSE`, `READ`, and `WRITE`.

The OPEN Statement

Before your program can access the data in an input file or place data in an output file, you must make the file available to the program by `OPEN`ing it. When you open a file, you have to indicate how you intend to use it (`INPUT`, `OUTPUT`, `EXTEND`) so the system can manage the file correctly:

$$\underline{\text{OPEN}}\ \left\{\begin{array}{l}\underline{\text{INPUT}}\\ \underline{\text{OUTPUT}}\\ \underline{\text{EXTEND}}\end{array}\right\}\ \text{InternalFileName} \ldots$$

Opening a file *does not* transfer any data to the record buffer; it simply provides access.

**Notes on the OPEN Statement**

When a file is opened for `INPUT` or `EXTEND`, the file must exist or the `OPEN` will fail.

When a file is opened for `INPUT`, the Next Record Pointer is positioned at the beginning of the file. The Next Record Pointer is conceptual; it points to the position in the file where the file system will get or put the next record.

When the file is opened for `EXTEND`, the Next Record Pointer is positioned after the last record in the file. This allows records to be appended to the file.

When a file is opened for `OUTPUT`, it is created if it does not exist, and it is overwritten if it already exists.

---

**Bug Alert** Although the ellipses after `InternalFileName` in the metalanguage indicate that it is possible to open a number of files with one `OPEN` statement, it is not advisable to do so. If an error is detected on opening a file and only one `OPEN` statement has been used to open all the files, the system will not be able to indicate which particular file is causing the problem. If all the files are opened separately, it will.

---

The CLOSE Statement

The metalanguage for the `CLOSE` statement is fairly simple:

```
CLOSE InternalFilename ...
```

**Notes**

Before the program terminates, you must make sure the program closes all the open files. Failure to do so may result in some data not being written to the file or users being prevented from accessing the file.

Hard disk access is about a million times slower than RAM access (hard disk access times are measured in milliseconds, whereas RAM access is measured in nanoseconds: 1 millisecond = 1,000 microseconds = 1,000,000 nanoseconds), so data is often cached in memory until a sufficient quantity of records have been accumulated to make the write to disk worthwhile. If the file is not closed, it is possible that these cached records will never be sent to the file.

---

**Bug Alert** The ellipses in the `CLOSE` metalanguage indicate that you may specify more than one file name. I advised against this for the `OPEN` statement; but because very few errors affect the `CLOSE` statement, the same advice does not hold. For convenience, you can often choose to close multiple files in one `CLOSE` statement.

---

The READ Statement

Once the system has opened a file and made it available to the program, it is your responsibility to process it correctly. To process all the records in the file, the program has to transfer them, one record at a time, from the file to the file's record buffer. The `READ` is provided for this purpose:

```
READ InternalFilename [NEXT] RECORD
        [INTO Identifier]
        [AT END StatementBlock1]
        [NOT AT END StatementBlock2]
[END-READ]
```

The `READ` statement copies a record occurrence (instance) from the file on backing storage and places it in the record buffer defined for it.

**Notes**

When the `READ` attempts to read a record from the file and encounters the end of file marker, the `AT END` is triggered and `StatementBlock1` is executed. If the `NOT AT END` clause is specified then `StatementBlock2` is executed.

When the `INTO` clause is used, the data is read into the record buffer and then copied from there, to the `Identifier`, in one operation. This option creates two copies of the data: one in the record buffer and one in the `Identifier`. Using the `INTO` clause is equivalent to reading a record and then moving the contents of the record buffer to the `Identifier`.

---

**COBOL Detail** Because `AT END` is an optional element, you might have wondered how the end-of-file condition can be detected in its absence. COBOL has a special kind of exception handler for files called *declaratives*. When declaratives are specified for a file, any file error—including the end-of-file condition, causes the code in the declaratives to execute. Declaratives are an advanced topic that I address later in the book.

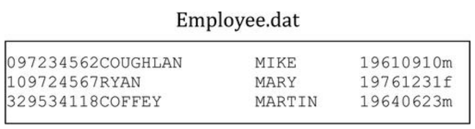
---

How READ Works

Listing 7-1 is a small program that simply reads the records in the employee file: `employee.dat`. The test data for the program is given in Figure 7-4. The effect on the data storage of running the program with this data is shown in Figure 7-5.

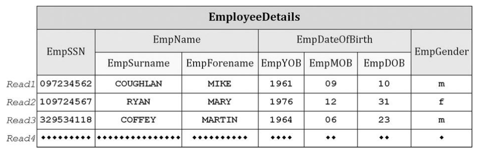
***Listing 7-1***. Reading the Employee File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing7-1.
AUTHOR. Michael Coughlan.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT EmployeeFile ASSIGN TO "Employee.dat"
          ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD EmployeeFile.
01 EmployeeDetails.
   88  EndOfEmployeeFile   VALUE HIGH-VALUES.
   02  EmpSSN              PIC 9(9).
   02  EmpName.
       03 EmpSurname       PIC X(15).
       03 EmpForename      PIC X(10).
   02  EmpDateOfBirth.
       03 EmpYOB           PIC 9(4).
       03 EmpMOB           PIC 99.
       03 EmpDOB           PIC 99.
   02  EmpGender           PIC X.

PROCEDURE DIVISION.
Begin.
   OPEN INPUT EmployeeFile
   READ EmployeeFile
     AT END SET EndOfEmployeeFile TO TRUE
   END-READ
   PERFORM UNTIL EndOfEmployeeFile
      READ EmployeeFile
        AT END SET EndOfEmployeeFile TO TRUE
      END-READ
   END-PERFORM
   CLOSE EmployeeFile
   STOP RUN.
```



```
Employee.dat

097234562COUGHLAN       MIKE      19610910m
109724567RYAN           MARY      19761231f
329534118COFFEY         MARTIN    19640623m
```

***Figure 7-4***. *Employee.dat test data file*



| | EmployeeDetails | | | | | | |
|---|---|---|---|---|---|---|---|
| | EmpSSN | EmpName | | EmpDateOfBirth | | | EmpGender |
| | | EmpSurname | EmpForename | EmpYOB | EmpMOB | EmpDOB | |
| Read1 | 097234562 | COUGHLAN | MIKE | 1961 | 09 | 10 | m |
| Read2 | 109724567 | RYAN | MARY | 1976 | 12 | 31 | f |
| Read3 | 329534118 | COFFEY | MARTIN | 1964 | 06 | 23 | m |
| Read4 | ◆◆◆◆◆◆◆◆◆ | ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆ | ◆◆◆◆◆◆◆◆◆◆ | ◆◆◆◆ | ◆◆ | ◆◆ | ◆ |

***Figure 7-5***. *Effect on data storage of reading each record in the file*

The effect on the data storage each time the `READ` is executed is shown in Figure 7-5:

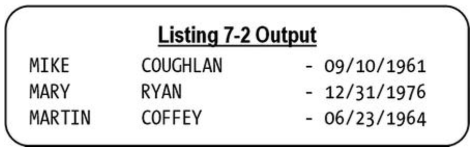
- `Read1` shows the effect of reading the first record. When the record is read from `Employee.dat`, it is copied into the `EmployeeDetails` area of storage as shown.
- `Read2` and `Read3` show the results of reading the second and third records.
- `Read4` shows what happens when an attempt to read a fourth record is made. Because there is no fourth record, the `AT END` activates and the condition name `EndOfEmployeeFile` is set to `TRUE`. This condition name is defined on the whole record, and as a result the whole record is filled with `HIGH-VALUES` (represented here as the ◆ symbol).

**Note** Because of space constraints in this book, the various pieces of test data given obviously are not comprehensive enough to test any of the programs adequately. The test data is provided for the purposes of illustration only. You should create your own, more comprehensive test data if you want to test the programs.

Of course, this program does not do anything practical. It reads the file but doesn't do anything with the records it reads. Listing 7-2 tweaks the program a little so that it displays the name and date of birth of each employee in the file. Notice that I have chosen not to display the data items in the same order they are in in the record. The employee name is displayed in forename-surname order, and the date of birth is displayed in the standard U.S. order (month, day, year).

***Listing 7-2***. Reading the Employee File and Displaying the Records

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing7-2.
AUTHOR. Michael Coughlan.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT EmployeeFile ASSIGN TO "Employee.dat"
          ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD EmployeeFile.
01 EmployeeDetails.
   88  EndOfEmployeeFile   VALUE HIGH-VALUES.
   02  EmpSSN              PIC 9(9).
   02  EmpName.
       03 EmpSurname       PIC X(15).
       03 EmpForename      PIC X(10).
   02  EmpDateOfBirth.
       03 EmpYOB           PIC 9(4).
       03 EmpMOB           PIC 99.
       03 EmpDOB           PIC 99.
   02  EmpGender           PIC X.

PROCEDURE DIVISION.
Begin.
   OPEN INPUT EmployeeFile
   READ EmployeeFile
     AT END SET EndOfEmployeeFile TO TRUE
   END-READ
   PERFORM UNTIL EndOfEmployeeFile
      DISPLAY EmpForename SPACE EmpSurname " - "
              EmpMOB "/" EmpDOB "/" EmpYOB
      READ EmployeeFile
        AT END SET EndOfEmployeeFile TO TRUE
      END-READ
   END-PERFORM
   CLOSE EmployeeFile
   STOP RUN.
```


**Listing 7-2 Output**

```
MIKE      COUGHLAN       - 09/10/1961
MARY      RYAN           - 12/31/1976
MARTIN    COFFEY         - 06/23/1964
```

The WRITE Statement

The `WRITE` statement is used to copy data from the record buffer (RAM) to the file on backing storage (tape, disk, or CD-ROM). To write data to a file, the data must be moved to the record buffer (declared in the file's FD entry), and then the `WRITE` statement is used to send the contents of the record buffer to the file:

$$\underline{\text{WRITE}}\ \text{RecordName}\ \left[\underline{\text{FROM}}\ \text{Identifier}\right]$$
$$\left[\left\{\begin{array}{l}\underline{\text{BEFORE}}\\ \underline{\text{AFTER}}\end{array}\right\}\text{ADVANCING}\left\{\begin{array}{l}\text{AdvanceNum}\left[\begin{array}{l}\underline{\text{LINE}}\\ \underline{\text{LINES}}\end{array}\right]\\ \text{MnemonicName}\\ \underline{\text{PAGE}}\end{array}\right\}\right]$$

When `WRITE..FROM` is used, the data contained in the `Identifier` is copied into the record buffer and is then written to the file. `WRITE..FROM` is the equivalent of a `MOVE Identifier TO RecordName` statement followed by a `WRITE RecordName` statement.

---

**Note** The full metalanguage for the sequential-file version of `WRITE` statement is given here, but I postpone discussion of the `ADVANCING` clause until later in the book. This clause is used when writing print files and is a bit more complicated than it appears on the surface. It is best discussed when considering files with multiple record types.

---

**Write a Record, Read a File**

You probably noticed that the metalanguage for the `READ` and `WRITE` statements indicates that while you read a file, you write a record. You may have wondered why there is this difference.

So far, you have only seen files that contain one type of record. In the employee file, for example, there is only one type of employee record. But a file may contain a number of different types of record. For instance, if you wanted to update the employee file, you might have a file of transaction records containing both Employee Insertion records and Employee Deletion records. Although an `Insertion` record would have to contain all the fields in the employee record, a `Deletion` record would only need the Employee SSN.

The reason you *read a file*, not a record, is that until the record is in the buffer you cannot tell what type of record it is. You have to read the file and then look at the data in the buffer to see what type of record has been supplied. It is your responsibility to discover what type of record has been read into the buffer and then to take the appropriate actions.

The reason you *write a record* instead of a file is that when the output file will contain multiple types of record, you have to specify which record type you want to write to the file.

**How WRITE works**

Suppose you want to add some records to the end of the employee file. To do so, you use this statement:

```
OPEN EXTEND EmployeeFile
```

This tells the system that you are going to add records to the end of the file. If you opened the file for `OUTPUT`, then `Employee.dat` would be replaced (overwritten) with a new version of the employee file.

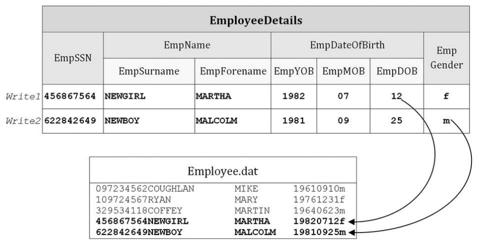
To write a record to the file, you place the data in the `EmployeeDetails` record buffer and then use the following statement:

```
WRITE EmployeeDetails
```

The example program fragment in Example 7-6 writes two records to the end of the employee file. Figure 7-6 shows the interaction between the data in memory and the file on backing storage. The first `MOVE` statement places the record data in the record buffer, and the `WRITE` statement (`Write1`) copies it to the file. The second `MOVE` places the second record in the record buffer, and the `WRITE` (`Write2`) copies it to the file.

***Example 7-6***. Writing Records to the End of a Sequential File

```
PROCEDURE DIVISION.
Begin.
  OPEN EXTEND EmployeeFile

  MOVE "456867564NEWGIRL        MARTHA    19820712f"
        TO EmployeeDetails
  WRITE EmployeeDetails

  MOVE "622842649NEWBOY         MALCOLM   19810925m"
        TO EmployeeDetails
  WRITE EmployeeDetails

  CLOSE EmployeeFile
  STOP RUN.
```



| | EmployeeDetails | | | | | | |
|---|---|---|---|---|---|---|---|
| | EmpSSN | EmpName | | EmpDateOfBirth | | | Emp Gender |
| | | EmpSurname | EmpForename | EmpYOB | EmpMOB | EmpDOB | |
| Write1 | 456867564 | NEWGIRL | MARTHA | 1982 | 07 | 12 | f |
| Write2 | 622842649 | NEWBOY | MALCOLM | 1981 | 09 | 25 | m |

Employee.dat

```
097234562COUGHLAN       MIKE      19610910m
109724567RYAN           MARY      19761231f
329534118COFFEY         MARTIN    19640623m
456867564NEWGIRL        MARTHA    19820712f
622842649NEWBOY         MALCOLM   19810925m
```

***Figure 7-6.*** *Writing two records to the employee file (see Example 7-6)*

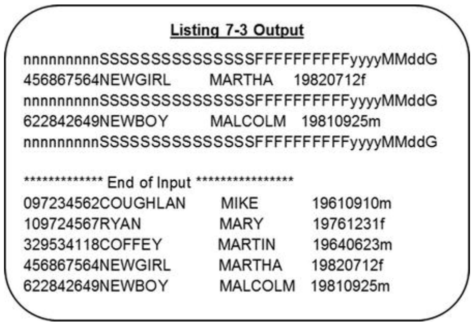
## Reading and Writing to the Employee File

The program in Listing 7-3 extends the fragment in Example 7-3 into a full-blown program. However, whereas the data sent to the employee file in Example 7-3 was hard-coded in the form of literal values, in Listing 7-3 the records to be added to the file are obtained from the user. A very simple interface is used to get the records. A template for the record is displayed, and the user then enters the data based on the template. A screen capture shows the data requested from the user and then output when the file is read.

***Listing 7-3***. Writing and Reading the Employee File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing7-3.
AUTHOR. Michael Coughlan.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT EmployeeFile ASSIGN TO "Employee.dat"
          ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD EmployeeFile.
01 EmployeeDetails.
   88  EndOfEmployeeFile   VALUE HIGH-VALUES.
   02  EmpSSN              PIC 9(9).
   02  EmpName.
       03 EmpSurname       PIC X(15).
       03 EmpForename      PIC X(10).
   02  EmpDateOfBirth.
       03 EmpYOB           PIC 9(4).
       03 EmpMOB           PIC 99.
       03 EmpDOB           PIC 99.
   02  EmpGender           PIC X.

PROCEDURE DIVISION.
Begin.
   OPEN EXTEND EmployeeFile
   PERFORM GetEmployeeData
   PERFORM UNTIL EmployeeDetails = SPACES
      WRITE EmployeeDetails
      PERFORM GetEmployeeData
   END-PERFORM
   CLOSE EmployeeFile
   DISPLAY "************* End of Input ****************"

   OPEN INPUT EmployeeFile
   READ EmployeeFile
     AT END SET EndOfEmployeeFile TO TRUE
   END-READ
   PERFORM UNTIL EndOfEmployeeFile
      DISPLAY EmployeeDetails
```

```
         READ EmployeeFile
            AT END SET EndOfEmployeeFile TO TRUE
         END-READ
      END-PERFORM
      CLOSE EmployeeFile
      STOP RUN.

   GetEmployeeData.
      DISPLAY "nnnnnnnnnSSSSSSSSSSSSSSSFFFFFFFFFFyyyyMMddG"
      ACCEPT EmployeeDetails.
```



**Listing 7-3 Output**

```
nnnnnnnnnSSSSSSSSSSSSSSSFFFFFFFFFFyyyyMMddG
456867564NEWGIRL       MARTHA    19820712f
nnnnnnnnnSSSSSSSSSSSSSSSFFFFFFFFFFyyyyMMddG
622842649NEWBOY        MALCOLM   19810925m
nnnnnnnnnSSSSSSSSSSSSSSSFFFFFFFFFFyyyyMMddG

************ End of Input ***************
097234562COUGHLAN      MIKE      19610910m
109724567RYAN          MARY      19761231f
329534118COFFEY        MARTIN    19640623m
456867564NEWGIRL       MARTHA    19820712f
622842649NEWBOY        MALCOLM   19810925m
```

**Summary**

This chapter provided a gentle introduction to sequential files. You learned how to declare the record buffer for a file. You learned how to connect the file's internal file name with its name and location on the backing storage device. You saw how to `READ` records from a file and how to `WRITE` them to a file.

Although this chapter is a good start, there is still much more to discover about sequential files. Although I touched on the idea of files that contain multiple record types, I did not explore the full ramifications of this concept; nor did I discuss the true magic of the `FILE SECTION`. I mentioned print files, but I did not explore the relevant options in the metalanguage; nor did I discuss how to create a print file. I also have not mentioned or discussed the idea of variable-length records. I discussed some of the mechanics of using sequential files in this chapter, but I did not discuss sequential-file processing issues. The chapters that follow explore some of those issues by examining the control-break and file-update problems.

**LANGUAGE KNOWLEDGE EXERCISE**

Unsheathe your 2B pencil. It is exercise time again.

1. Locate errors in these `FILE SECTION` entries.

(a)

```
FD SalesFile.
01 SalesRecord                ~~PIC X(13).~~ group item
   02 SalesmanNumber          PIC 9(7).
   02 SaleValue               PIC 9(5)V99.
```

(b)

```
FD TemperatureFile.
01 DayRecord.
   05 MonthNumber             PIC 99  .
   05 MaxTemp                 PIC 999 .
   05 MinTemp                 PIC 999 .
05 ~~06~~ AverageTemp         PIC 999 .
```

(c)

```
FD StudentFile
01 StudentRecord.
02 StudentName.               PIC X(20).
05 StudentInitials            PIC XX.
05 StudnetSurname             PIC X(18).
02 StudentAddress             PIC X(65).
03 AddressLine1.              PIC X(10).
03 AddressLine2               PIC X(10)
03 AddressLine3               PIC X(10).
02 StudentGPA.
```

```
05 Year1GPA
10 Sem1GPA                PIC 9V99.
10 Sem2GPA                PIC 9V99.
05 Year2GPA.
10 Sem1GPA                PIC 9V99.
10 Sem2GPA                PIC 9V99.
05 Year3GPA
10 Sem1GPA                PIC 9V99.
10 Sem2GPA                PIC 9V99.
```

2. Complete the `SELECT` and `ASSIGN` clause for a sequential file called `Stock.dat` in the directory `C:\COBOL-Data\`. The record buffer for the file has this description:

```
FD StockFile.
01 StockRec.
   02 StockNumber      PIC 9(5)
   02 ManfNumber       PIC 9(4)
   02 QtyInStock       PIC 9(6)
   02 ReorderLevel     PIC 9(6)
   02 ReorderQty       PIC 9(6).
```

Write your answer here:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Exercise7-2.
ENVIRONMENT DIVISION.
```

```
Input-output Section.
File-Control.
    select StockFile Assign to "                    "
        organization is sequential
```

**PROGRAMMING EXERCISE 1**

A `StockFile` holds details of gadgets sold by the Gadget Shop (`Gadget-Shop.Com` (http://GadgetShop.Com)). The `StockFile` is a sequential file sorted in ascending `GadgetId` order. It is named `GadgetStock.dat`. Each record has the following description:

| Field | Type | Length | Value |
|---|---|---|---|
| GadgetID | N | 6 | 000001-999999 |
| GadgetName | X | 30 | - |
| QtyInStock | N | 4 | 0000-9999 |
| Price | N | 6 | 0000.00-9999.99 |

Write a program to process the data in the `StockFile` and, for each record, display the item's `GadgetName` and the total value of the quantity in stock (`QtyInStock * Price`). When the `StockFile` has ended, display the total value of all the stock.

display "GadgetName:"
GadgetName
space
total

**Example Test Data**

```
ID              Name                stock  price
123456SoundDisk MP3 Player 4GB      0650003095
234567BioLite Camp Stove            0057029550
345678Collapsible Kettle - Green    0155002590
456789Digital Measuring Jug         0325000895
```

```
567890MicroLite LED Torch           0512000745
678901Pocket Sized Fishing Rod      0055001799
```

Note: Place the test data in the data file as one long string.

**Example Run**

```
SoundDisk MP3 Player 4GB        $20,117.50
BioLite Camp Stove              $16,843.50
Collapsible Kettle - Green       $4,014.50
Digital Measuring Jug            $2,908.75
MicroLite LED Torch              $3,814.40
Pocket Sized Fishing Rod           $989.45
                Stock Total:    $48,688.10
```

**PROGRAMMING EXERCISE 2**

Amend the program you wrote for exercise 1 so that it adds the following two records to the end of the file. Then display the stock report as before:

```
313245Spy Pen - HD Video Camera     0125003099
593486Scout Cash Capsule - Red      1234000745
```

The records in the `StockFile` are held in ascending `GadgetID` order. When you add these two records to the file, the records will be out of order. Without sorting the `StockFile` after the update, how could you update the file so that the ordering of the records was maintained?

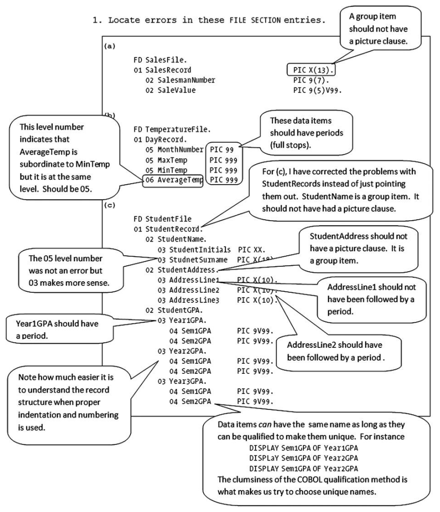
**Example Run**

```
SoundDisk MP3 Player 4GB        $20,117.50
BioLite Camp Stove              $16,843.50
Collapsible Kettle - Green       $4,014.50
Digital Measuring Jug            $2,908.75
MicroLite LED Torch              $3,814.40
Pocket Sized Fishing Rod           $989.45
Spy Pen - HD Video Camera        $3,873.75
Scout Cash Capsule - Red         $9,193.30
                Stock Total:    $61,755.15
```

**LANGUAGE KNOWLEDGE EXERCISES: ANSWERS**

Unsheath your 2B pencil. It is exercise time again.

1. Locate errors in these `FILE SECTION` entries.



1. Locate errors in these FILE SECTION entries.

```
(a)

      FD SalesFile.
      01 SalesRecord                          PIC X(13).
         02 SalesmanNumber                    PIC 9(7).
         02 SaleValue                         PIC 9(5)V99.

(b)

      FD TemperatureFile.
      01 DayRecord.
         05 MonthNumber  PIC 99
         05 MaxTemp      PIC 999
         05 MinTemp      PIC 999
         06 AverageTemp  PIC 999
(c)

      FD StudentFile
      01 StudentRecord.
         02 StudentName.
            03 StudentInitials  PIC XX.
            03 StudnetSurname   PIC X(10).
         02 StudentAddress.
            03 AddressLine1     PIC X(10).
            03 AddressLine2     PIC X(10).
            03 AddressLine3     PIC X(10).
         02 StudentGPA.
            03 Year1GPA.
               04 Sem1GPA       PIC 9V99.
               04 Sem2GPA       PIC 9V99.
            03 Year2GPA.
               04 Sem1GPA       PIC 9V99.
               04 Sem2GPA       PIC 9V99.
            03 Year3GPA.
               04 Sem1GPA       PIC 9V99.
               04 Sem2GPA       PIC 9V99.
```

A group item should not have a picture clause.

This level number indicates that AverageTemp is subordinate to MinTemp but it is at the same level. Should be 05.

These data items should have periods (full stops).

For (c), I have corrected the problems with StudentRecords instead of just pointing them out. StudentName is a group item. It should not have had a picture clause.

StudentAddress should not have a picture clause. It is a group item.

The 05 level number was not an error but 03 makes more sense.

AddressLine1 should not have been followed by a period.

Year1GPA should have a period.

AddressLine2 should have been followed by a period .

Note how much easier it is to understand the record structure when proper indentation and numbering is used.

Data items *can* have the same name as long as they can be qualified to make them unique. For instance

```
DISPLAY Sem1GPA OF Year1GPA
DISPLaY Sem1GPA OF Year2GPA
DISPLAY Sem1GPA OF Year2GPA
```

The clumsiness of the COBOL qualification method is what makes us try to choose unique names.

2. Complete the SELECT and ASSIGN clause for a sequential file called Stock.dat in the directory C:\COBOL-Data\. The record buffer for the file has this description:

```
FD StockFile.
01 StockRec.
   02 StockNumber      PIC 9(5)
   02 ManfNumber       PIC 9(4)
   02 QtyInStock       PIC 9(6)
   02 ReorderLevel     PIC 9(6)
   02 ReorderQty       PIC 9(6).
```

Write your answer here:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Exercise7-2.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT StockFile ASSIGN TO "C:\COBOL-Data\St
          ORGANIZATION IS SEQUENTIAL.
```

**PROGRAMMING EXERCISE 1: ANSWER**

A StockFile holds details of gadgets sold by the Gadget Shop (Gadget-Shop.Com (http://GadgetShop.Com)). The StockFile is a sequential file sorted in ascending GadgetId order. It is named GadgetStock.dat. Each record has the following description.

| Field | Type | Length | Value |
|---|---|---|---|
| GadgetID | N | 6 | 000001–999999 |
| GadgetName | X | 30 | – |
| QtyInStock | N | 4 | 0000–9999 |
| Price | N | 6 | 0000.00–9999.99 |

Write a program to process the data in the StockFile and, for each record, display the item's GadgetName and the total value of the quantity in stock (QtyInStock * Price). When the StockFile has ended, display the total value of all the stock.

**Example Test Data**

```
123456SoundDisk MP3 Player 4GB      0650003095
234567BioLite Camp Stove            0057029550
345678Collapsible Kettle - Green    0155002590
456789Digital Measuring Jug         0325000895
567890MicroLite LED Torch           0512000745
678901Pocket Sized Fishing Rod      0055001799
```

Note: Place the test data in the data file as one long string.

**Example Run**

```
SoundDisk MP3 Player 4GB        $20,117.50
BioLite Camp Stove              $16,843.50
Collapsible Kettle - Green       $4,014.50
Digital Measuring Jug            $2,908.75
MicroLite LED Torch              $3,814.40
Pocket Sized Fishing Rod           $989.45
                Stock Total:    $48,688.10
```

***Listing 7-4***. Displays the Value of the Gadgets in Stock

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Listing7-4.
AUTHOR. Michael Coughlan

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT GadgetStockFile ASSIGN TO "input.txt"
          ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD GadgetStockFile.
01 StockRec.
   88 EndOfStockFile      VALUE HIGH-VALUES.
   02 GadgetID            PIC 9(6).
   02 GadgetName          PIC X(30).
   02 QtyInStock          PIC 9(4).
   02 Price               PIC 9(4)V99.

WORKING-STORAGE SECTION.
01 PrnStockValue.
   02 PrnGadgetName       PIC X(30).
   02 FILLER              PIC XX VALUE SPACES.
   02 PrnValue            PIC $$$,$$9.99.

01 PrnFinalStockTotal.
   02 FILLER              PIC X(16) VALUE SPACES.
   02 FILLER              PIC X(16) VALUE "Stock Total:".
   02 PrnFinalTotal       PIC $$$,$$9.99.

01 FinalStockTotal        PIC 9(6)V99.
01 StockValue             PIC 9(6)V99.

PROCEDURE DIVISION.
Begin.
    OPEN INPUT GadgetStockFile
    READ GadgetStockFile
       AT END SET EndOfStockFile TO TRUE
    END-READ
    PERFORM DisplayGadgetValues UNTIL EndOfStockFile
    MOVE FinalStockTotal TO PrnFinalTotal
    DISPLAY PrnFinalStockTotal
    CLOSE GadgetStockFile
    STOP RUN.

DisplayGadgetValues.
    COMPUTE StockValue = Price * QtyInStock
    ADD StockValue  TO FinalStockTotal
    MOVE GadgetName TO PrnGadgetName
    MOVE StockValue TO PrnValue
    DISPLAY PrnStockValue
    READ GadgetStockFile
         AT END SET EndOfStockFile TO TRUE
    END-READ.
```

**PROGRAMMING EXERCISE 2: ANSWER**

Amend the program you wrote for exercise 1 so that it adds the following two records to the end of the file. Then display the stock report as before:

```
313245Spy Pen - HD Video Camera     0125003099
593486Scout Cash Capsule - Red      1234000745
```

The records in the `StockFile` are held in ascending `GadgetID` order. When you add these two records to the file, the records will be out of order. Without sorting the `StockFile` after the update, how could you update the file so that the ordering of the records was maintained?

**Example Run**

```
SoundDisk MP3 Player 4GB        $20,117.50
BioLite Camp Stove              $16,843.50
Collapsible Kettle - Green       $4,014.50
Digital Measuring Jug            $2,908.75
MicroLite LED Torch              $3,814.40
Pocket Sized Fishing Rod           $989.45
Spy Pen - HD Video Camera        $3,873.75
Scout Cash Capsule - Red         $9,193.30
                Stock Total:    $61,755.15
```

***Listing 7-5***. Adds Two Records and Then Displays Stock Values Again

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Listing7-5.
AUTHOR. Michael Coughlan

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT GadgetStockFile ASSIGN TO "input.txt"
          ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD GadgetStockFile.
01 StockRec.
   88 EndOfStockFile      VALUE HIGH-VALUES.
   02 GadgetID            PIC 9(6).
   02 GadgetName          PIC X(30).
   02 QtyInStock          PIC 9(4).
   02 Price               PIC 9(4)V99.

WORKING-STORAGE SECTION.
01 PrnStockValue.
   02 PrnGadgetName       PIC X(30).
   02 FILLER              PIC XX VALUE SPACES.
   02 PrnValue            PIC $$$,$$9.99.

01 PrnFinalStockTotal.
   02 FILLER              PIC X(16) VALUE SPACES.
   02 FILLER              PIC X(16) VALUE "Stock Total:".
   02 PrnFinalTotal       PIC $$$,$$9.99.

01 FinalStockTotal        PIC 9(6)V99.
01 StockValue             PIC 9(6)V99.

PROCEDURE DIVISION.
Begin.
    OPEN EXTEND GadgetStockFile
    MOVE "313245Spy Pen - HD Video Camera     0125003099"
         TO StockRec
    WRITE StockRec
    MOVE "593486Scout Cash Capsule - Red      1234000745"
         TO StockRec
    WRITE StockRec
    CLOSE GadgetStockFile

    OPEN INPUT  GadgetStockFile
    READ GadgetStockFile
       AT END SET EndOfStockFile TO TRUE
    END-READ
    PERFORM DisplayGadgetValues UNTIL EndOfStockFile
    MOVE FinalStockTotal TO PrnFinalTotal
    DISPLAY PrnFinalStockTotal
    CLOSE GadgetStockFile
    STOP RUN.

DisplayGadgetValues.
    COMPUTE StockValue = Price * QtyInStock
    ADD StockValue  TO FinalStockTotal
    MOVE GadgetName TO PrnGadgetName
    MOVE StockValue TO PrnValue
    DISPLAY PrnStockValue
```

```
READ GadgetStockFile
     AT END SET EndOfStockFile TO TRUE
END-READ.
```

Support / Sign Out

PREV
CHAPTER 6: Control Structures: Iteration

NEXT
CHAPTER 8: Advanced Sequential Files