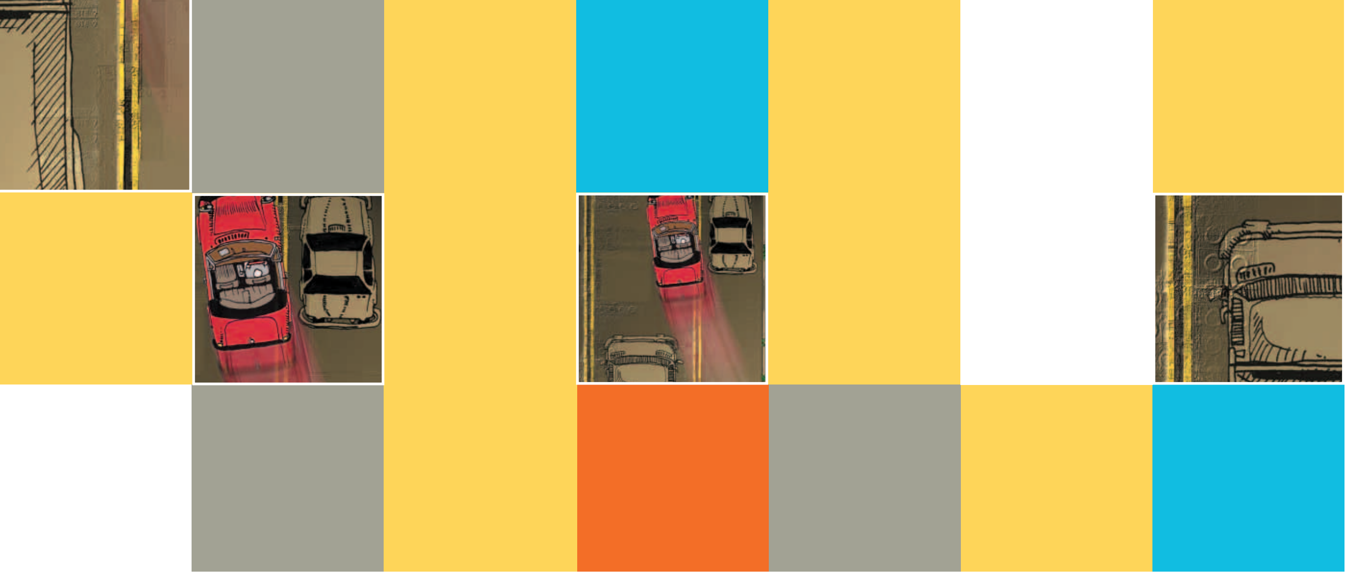


11 SEQUENTIAL FILE MERGING, MATCHING, AND UPDATING

After studying Chapter 11, you should be able to:

- ☐ Understand sequential files and the need for merging them
- ☐ Create the mainline and `housekeeping()` logic for a merge program
- ☐ Create the `mergeFiles()` and `finishUp()` modules for a merge program
- ☐ Modify the `housekeeping()` module to check for `eof`
- ☐ Understand master and transaction file processing
- ☐ Match files to update master file fields
- ☐ Allow multiple transactions for a single master file record
- ☐ Update records in sequential files

UNDERSTANDING SEQUENTIAL DATA FILES AND THE NEED FOR MERGING FILES

A **sequential file** is a file in which records are stored one after another in some order. One option is to store records in a sequential file in the order in which the records are created. For example, if you maintain records of your friends, you might store the records as you make the friends; you could say the records are stored in **temporal order**—that is, in order based on time. At any point in time, the records of your friends will be stored in sequential order based on how long you have known them—the data stored about your best friend from kindergarten is record 1, and the data about the friend you just made last week could be record 30.

Instead of temporal order, records in a sequential file are more frequently stored based on the contents of one or more fields within each record. Perhaps it is most useful for you to store your friends' records sequentially in alphabetical order by last name, or maybe in order by birthday.

Other examples of sequential files include:

- A file of employees stored in order by Social Security 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

TIP Recall from Chapter 9 that the field that makes a record unique from all records in a file is the key field. Frequently, though not always, records are most conveniently stored in order by their key fields.

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

- Suppose you have a file of current employees in Social Security number order and a file of newly hired employees, also in Social Security number order. You need to merge these two files into one combined file before running this week's payroll program.
- Suppose you have a file of parts manufactured in the Northside factory in part-number order and a file of parts manufactured 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 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 (ascending or descending) based on the same field.

For example, suppose 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 master file containing records for all customers. Figure 11-1 shows some sample data for the files; you want to create a merged file like the one shown in Figure 11-2.

FIGURE 11-1: SAMPLE DATA CONTAINED IN TWO CUSTOMER FILES

```
East Coast File                West Coast File
eastName    eastBalance        westName       westBalance
Able        100.00             Chen           200.00
Brown        50.00             Edgar          125.00
Dougherty    25.00             Fell            75.00
Hanson      300.00             Grand          100.00
Ingram      400.00
Johnson      30.00
```

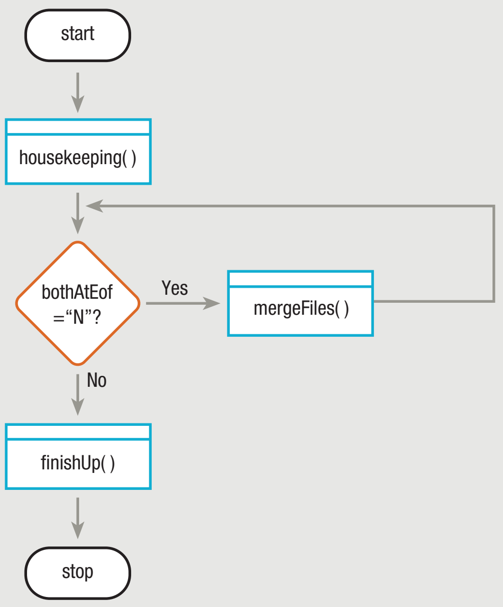
FIGURE 11-2: MERGED CUSTOMER FILE

```
mergedName                    mergedBalance
Able                          100.00
Brown                          50.00
Chen                          200.00
Dougherty                      25.00
Edgar                         125.00
Fell                           75.00
Grand                         100.00
Hanson                        300.00
Ingram                        400.00
Johnson                        30.00
```

CREATING THE MAINLINE AND `housekeeping()` LOGIC FOR A MERGE PROGRAM

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 a `housekeeping()` module, a `mergeFiles()` module that repeats until the end of the program, and a `finishUp()` module. Most programs you have written would repeat the main, central module (in this program, the `mergeFiles()` module) until the `eof` condition occurs. In a program that merges files, there are two input files, so checking for `eof` on one of them is insufficient. Instead, the mainline logic will check a flag variable that you create with a name such as `bothAtEof`. You will set the `bothAtEof` flag to "Y" after you have encountered `eof` in both input files. Figure 11-3 shows the mainline logic.

FIGURE 11-3: FLOWCHART AND PSEUDOCODE FOR MAINLINE LOGIC OF THE MERGE PROGRAM



```
start
  perform housekeeping()
  while bothAtEof = "N"
        perform mergeFiles()
  endwhile
  perform finishUp()
stop
```

TIP You first used flag variables in Chapter 8. A flag is a variable that keeps track of whether an event has occurred.

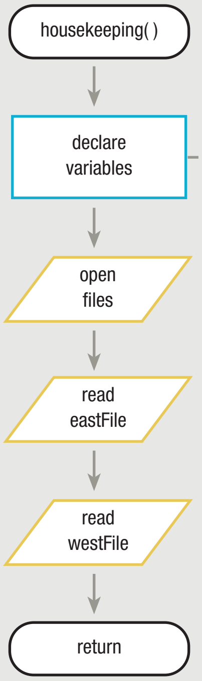
When you declare variables within the `housekeeping()` module, you must declare the `bothAtEof` flag and initialize it to "N" to indicate that the input files have not yet reached the end-of-file condition. In addition, you need to define two input files, one for the file from the East Coast office and one for the file from the West Coast office. Figure 11-4 shows that the files are called `eastFile` and `westFile`. Their variable fields are `eastName`, `eastBalance`, `westName`, and `westBalance`, respectively.

TIP You will modify the `housekeeping()` module in Figure 11-4 later in this chapter, after you learn about the special techniques needed to handle the `eof` conditions in this program.

At the end of a `housekeeping()` module, typically you read the first file input record into memory. In this file-merging program with two input files, you will read one record from *each* input file into memory at the end of the `housekeeping()` module.

The output from the merge program is a new, merged file containing all records from the two original input files. Logically, writing to a file and writing a printed report are very similar—each involves sending data to an output device. The major difference is that when you write a data file, typically you do not include headings or other formatting for people to read, as you do when creating a printed report. A **data file** contains only data for another computer program to read.

FIGURE 11-4: FLOWCHART AND PSEUDOCODE FOR THE `housekeeping()` MODULE IN THE MERGE PROGRAM, VERSION 1



```
eastFile
   char eastName
   num eastBalance
westFile
   char westName
   num westBalance
char bothAtEof = "N"
```

```
housekeeping()
     declare variables
     open files
     read eastFile
     read westFile
return
```

TIP Logically, the verbs "print," "write," and "display" mean the same thing—all produce output. However, in conversations, programmers usually reserve the word "print" for situations in which they mean "produce hard copy output," and 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 you use for output, no matter what type of hardware you use; you simply assign different output devices (such as printers, monitors, and disk drives) as needed to programmer-named objects that represent them.

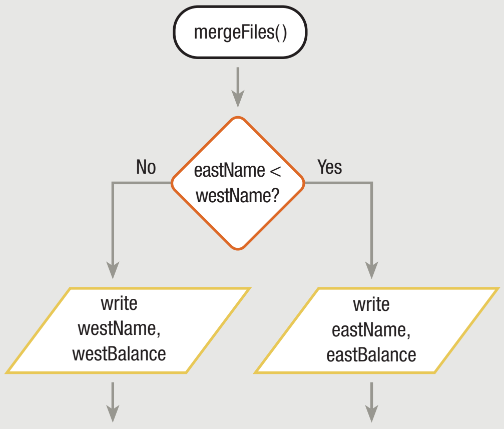
TIP In some programming languages, you might assign each input field to a named variable designed specifically for output before writing. In many languages, such as Java, C++, and C#, you can use the same variable as an input field and an output field—a convention that is followed in this book.

TIP In many organizations, both data files and printed report files are sent to disk storage devices when they are created. Later, as time becomes available on the organization's busy printers (often after business hours), the report disk files are copied to paper.

CREATING THE `mergeFiles()` AND `finishUp()` MODULES FOR A MERGE PROGRAM

When you begin the `mergeFiles()` module, 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 `mergeFiles()` module begins as shown in Figure 11-5.

FIGURE 11-5: BEGINNING OF THE `mergeFiles()` MODULE OF THE MERGE PROGRAM



```
mergeFiles()
     if eastName < westName then
          write eastName, eastBalance
          …
     else
          write westName, westBalance
          …
```

TIP Don't be confused by a statement such as `write  eastName, eastBalance`. Even though `eastName` and `eastBalance` are input fields, *writing* them sends their contents to a new output file, just as printing them sends them to a piece of paper. In some older programming languages, you had to move input fields such as `eastName` and `eastBalance` to an output area before you could write them to a file.

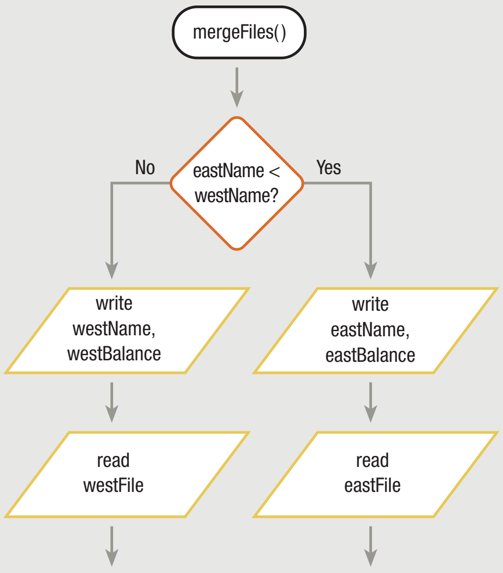
Using the sample data from Figure 11-1, you can see that the record from the East Coast file containing "Able" should be written to the output file, while Chen's record from the West Coast file waits in memory because the `eastName` value "Able" is alphabetically lower than the `westName` value "Chen".

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 to output, 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 `mergeFiles()` module 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 11-6.

FIGURE 11-6: CONTINUATION OF THE `mergeFiles()` MODULE FOR THE MERGE PROGRAM



```
mergeFiles()
     if eastName < westName then
          write eastName, eastBalance
          read eastFile
          …
     else
          write westName, westBalance
          read westFile
          …
```

Recall the names from the two original files (see Figure 11-7) and walk through the processing steps.

FIGURE 11-7: NAMES FROM TWO FILES TO MERGE

eastName	westName
Able	Chen
Brown	Edgar
Dougherty	Fell
Hanson	Grand
Ingram	
Johnson	

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". Write Chen'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`.

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 is written yet. You need to find a way to write the Hanson record as well as read and write all the remaining `eastFile` records. And you can't just write statements to read and write from `eastFile`; sometimes, when you run this program, records in `eastFile` will finish first alphabetically, and in that case you need to continue reading from `westFile`.

An elegant solution to this problem involves setting the field on which the merge is based to a "high" value when the end of the file is encountered. A **high value** is one that is greater than any possible value in a field. Programmers often use all 9s in a numeric field and all Zs in a character field to indicate a high value. Every time you read from `westFile` you can check for `eof`, and when it occurs, set `westName` to "ZZZZZ". Similarly, when reading `eastFile`, set `eastName` to "ZZZZZ" when `eof` occurs. When both `eastName` and `westName` are "ZZZZZ", then you set the `bothAtEof` variable to "Y". Figure 11-8 shows the complete `mergeFiles()` logic.

TIP

At the end of the file, you might choose to use 10 or 20 Zs in the `eastName` and `westName` fields instead of using only five. Although it is unlikely that a person will have the last name ZZZZZ, you should make sure that the value you choose for a high value is actually a higher value than any legitimate value.

TIP

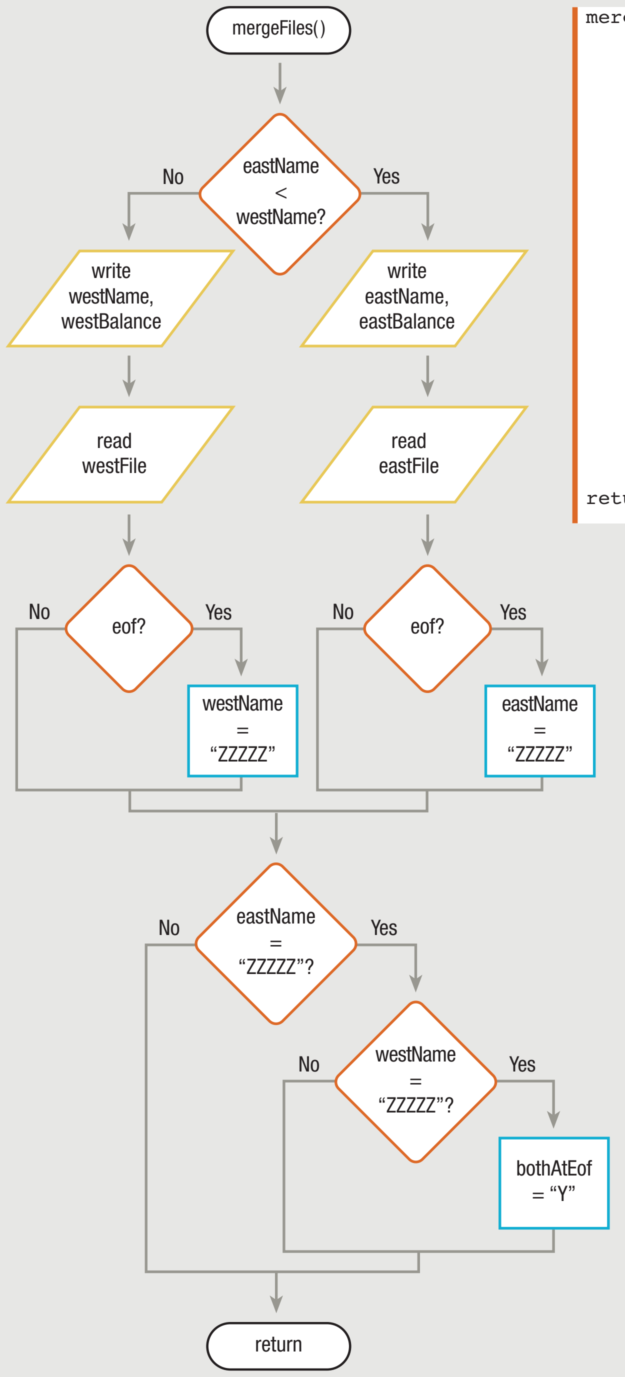
Several programming languages contain a name you can use for a value that occurs when every bit in a byte is an "on" bit, creating a value that is even higher than all Zs or all 9s. For example, in COBOL this value is called HIGH-VALUES, and in RPG it is called HIVAL.

Using the sample data in Figure 11-7, after Grand's record is processed, `westFile` is read and `eof` is encountered, so `westName` gets set to "ZZZZZ". Now, when you enter the `mergeFiles()` module 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 writes 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 as listed previously, and then proceeds as shown in Figure 11-8 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 the `eastFile`, encountering `eof` and setting `eastName` to "ZZZZZ".
11. Now that both names are "ZZZZZ", set the flag `bothAtEof` equal to "Y".

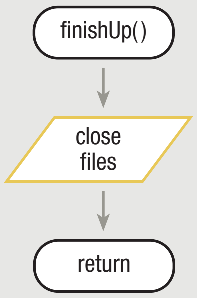
FIGURE 11-8: THE mergeFiles() MODULE FOR THE MERGE PROGRAM, COMPLETED



```
mergeFiles()
     if eastName < westName then
          write eastName, eastBalance
          read eastFile
          if eof then
               eastName = "ZZZZZ"
          endif
     else
          write westName, westBalance
          read westFile
          if eof then
               westName = "ZZZZZ"
          endif
     endif
     if eastName = "ZZZZZ" then
          if westName = "ZZZZZ" then
               bothAtEof = "Y"
          endif
     endif
return
```

When the `bothAtEof` flag variable equals "Y" at the end of the `mergeFiles()` module, the mainline logic then proceeds to the `finishUp()` module. See Figure 11-9.

FIGURE 11-9: THE `finishUp()` MODULE FOR THE MERGE PROGRAM



```
finishUp()
     close files
return
```

TIP

Notice that 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.

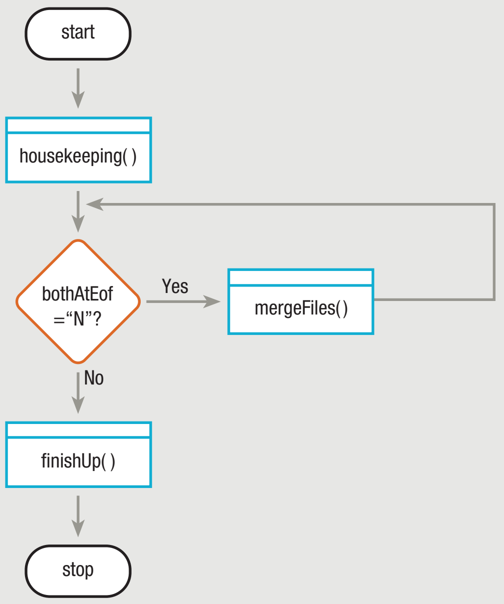
TIP

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 three files before deciding which file is the next candidate for output.

MODIFYING THE housekeeping() MODULE IN THE MERGE PROGRAM TO CHECK FOR eof

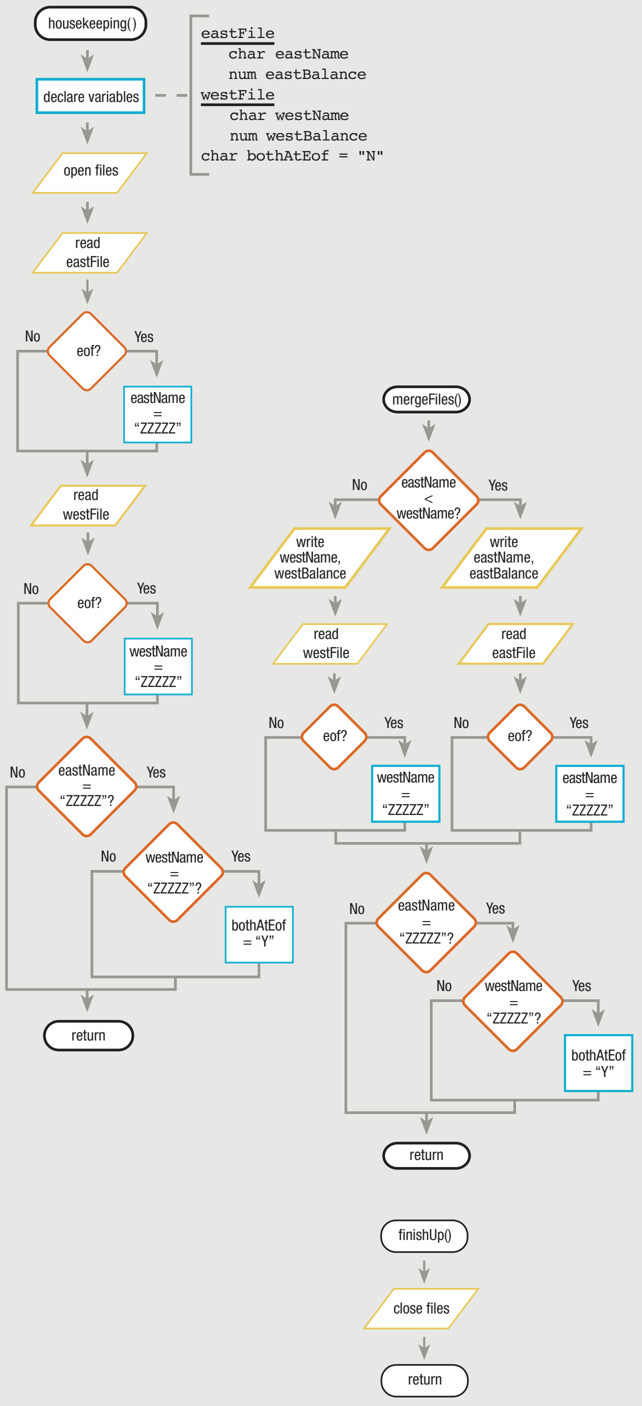
Recall that in the `housekeeping()` module for the merge program that combines East Coast and West Coast customer files, you read one record from each of the two input files. Although it is unlikely that you will reach the end of the file after attempting to read the first record in a file, it is good practice to check for `eof` every time you read. In the `housekeeping()` module, you first read from one of the input files. Whether you encounter `eof` or not, you then read from the second input file. If both files are at `eof`, then both name fields are set to “ZZZZZ”, and you can set the `bothAtEof` flag to “Y”. Then, when the `housekeeping()` module ends, if the value of `bothAtEof` is “Y”, it means that there are no records to merge, and the mainline logic will immediately send the program to the `finishUp()` module. Figure 11-10 shows the complete merge program, including the newly modified `housekeeping()` module that checks for the end of each input file.

FIGURE 11-10: THE COMPLETE FILE MERGE PROGRAM



```
start
     perform housekeeping()
     while bothAtEof = "N"
          perform mergeFiles()
     endwhile
     perform finishUp()
stop
```

FIGURE 11-10: THE COMPLETE FILE MERGE PROGRAM (CONTINUED)



```
eastFile
    char eastName
    num eastBalance
westFile
    char westName
    num westBalance
char bothAtEof = "N"
```

```
housekeeping()
     declare variables
     open files
     read eastFile
     if eof then
          eastName = "ZZZZZ"
     endif
     read westFile
     if eof then
          westName = "ZZZZZ"
     endif
     if eastName = "ZZZZZ" then
          if westName = "ZZZZZ" then
               bothAtEof = "Y"
          endif
     endif
return
```

```
mergeFiles()
     if eastName < westName then
          write eastName, eastBalance
          read eastFile
          if eof then
               eastName = "ZZZZZ"
          endif
     else
          write westName, westBalance
          read westFile
          if eof then
               westName = "ZZZZZ"
          endif
     endif
     if eastName = "ZZZZZ" then
          if westName = "ZZZZZ" then
               bothAtEof = "Y"
          endif
     endif
return
```

```
finishUp()
     close files
return
```

MASTER AND TRANSACTION FILE PROCESSING

When two related sequential files seem "equal," in that they hold the same *type* of information—for example, when one holds customers from the East Coast and one holds customers from the West Coast—you often need to merge the files to use them as a single unit. When you merge records from two or more files, the records (almost) always contain the same fields in the same order; in other words, every record in the merged file has the same format.

Some related sequential files, however, are unequal and you do not want to merge them. For example, you might have a file containing records for all your customers, in which each record holds a customer ID number, name, address, and balance due. You might have another file that contains data for every purchase made, containing the customer ID number and other purchase information such as a dollar amount. Although both files contain a customer ID number, the file with the customer names and addresses is an example of a master file. You use a **master file** to hold relatively permanent data, such as customers' names. The file containing customer purchases is a **transaction file**, a file that holds more temporary data generated by the actions of the customers. You may maintain certain customers' names and addresses for years, but the transaction file will contain new data daily, weekly, or monthly, depending on your organization's billing cycle. Commonly, you periodically use a transaction file to find a **matching record** in a master file—one that contains data about the same customer. Sometimes, you match records so you can **update the master file** by making changes to the values in its fields. For example, the file containing transaction purchase data might be used to update each master file record's balance due field. At other times, you might match a transaction file's records to its master file counterpart, creating an entity that draws information from both files—an invoice, for example. This type of program requires matching, but no updating. Whether a program simply matches records in master and transaction files, or updates the master file, depending on the application, there might be none, one, or many transaction records corresponding to each master file record.

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 version of the master file for a period of time, in case there are questions or discrepancies regarding the update process. The saved version of a master file is the **parent file**; the updated version is the **child file**. This approach is used later in this chapter.

TIP

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.

TIP

The terms "parent" and "child" refer to file backup generations, but they also are used 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 will learn about these concepts in Chapter 13.

MATCHING FILES TO UPDATE FIELDS IN MASTER FILE RECORDS

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.

Assume you have a master file with the fields shown in Figure 11-11.

FIGURE 11-11: MASTER CUSTOMER FILE DESCRIPTION

```
MASTER CUSTOMER FILE DESCRIPTION
File name: CUSTOMERS
FIELD DESCRIPTION        DATA TYPE      COMMENTS
Customer number          Numeric        3 digits
Name                     Character
Address                  Character
Phone number             Character
Total sales              Numeric        2 decimal places
```

The `custTotalSales` field holds the total dollar amount of all purchases the customer has made previously; in other words, it holds the total amount the customer has spent prior to the current week. At the end of each week, you want to update this field with any new sales transaction that occurred during the week. Assume a transaction file contains one record for every transaction that has occurred and that each record holds a transaction number, the number of the customer who made the transaction, the transaction date, and the amount of the transaction. The fields in the transaction file are shown in Figure 11-12.

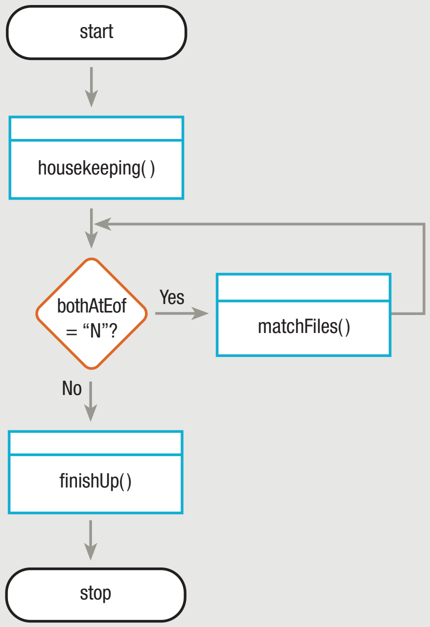
FIGURE 11-12: TRANSACTION FILE DESCRIPTION

```
TRANSACTION FILE DESCRIPTION
File name: TRANSACTIONS
FIELD DESCRIPTION        DATA TYPE      COMMENTS
Transaction number       Numeric        7 digits
Customer number          Numeric        3 digits
Transaction date         Numeric        8 digits YYYYMMDD
Transaction amount       Numeric        2 decimal places
```

You want to create a new master file in which almost all information is the same as in the original file, but the `custTotalSales` field increases to reflect the most recent transaction. The process involves going through the old master file, one record at a time, and determining whether there is a new transaction for that customer. If there is no transaction for a customer, the new customer record will contain exactly the same information as the old customer record. However, if there is a transaction for a customer, the `transAmount` value adds to the `custTotalSales` field before you write the updated master file record to output. Imagine you were going to update master file records by hand instead of using a computer program, and imagine each master and transaction record was stored on a separate piece of paper. The easiest way to accomplish the update would be 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. You would then correct the matching master record and examine the next transaction. The computer program you write to perform the update works exactly the same way.

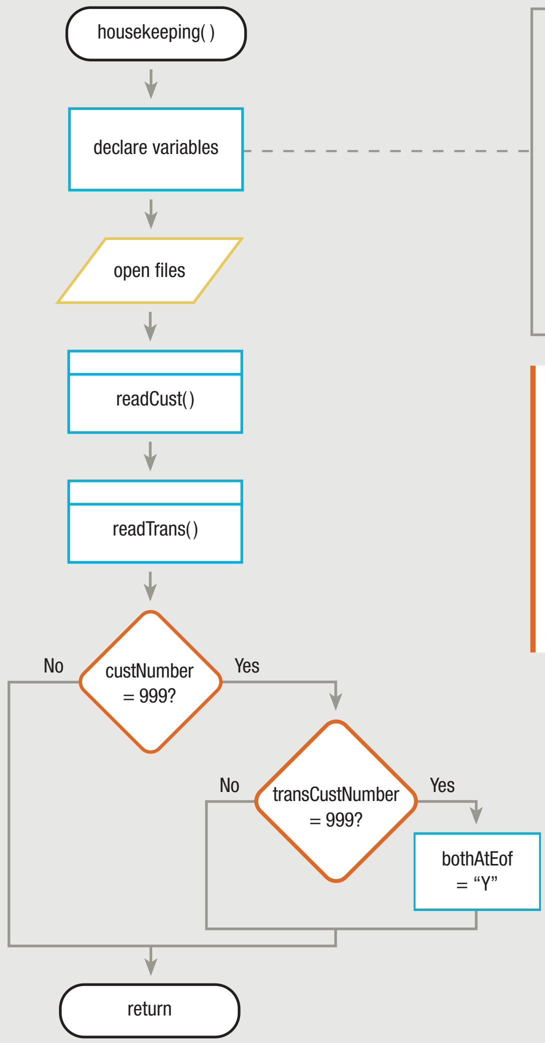
The mainline logic (see Figure 11-13) and `housekeeping()` module (see Figure 11-14) for this matching program look similar to their counterparts in a file-merging program. Two records are read, one from the master file and one from the transaction file. When you encounter `eof` for either file, store a high value (999) in the customer number field. Using the `readCust()` and `readTrans()` modules moves the reading of files and checking for `eof` off into their individual modules, as shown in Figure 11-15.

FIGURE 11-13: MAINLINE LOGIC FOR THE FILE-MATCHING PROGRAM



```
start
     perform housekeeping()
     while bothAtEof = "N"
            perform matchFiles()
     endwhile
     perform finishUp()
stop
```

FIGURE 11-14: THE `housekeeping()` MODULE FOR THE FILE-MATCHING PROGRAM



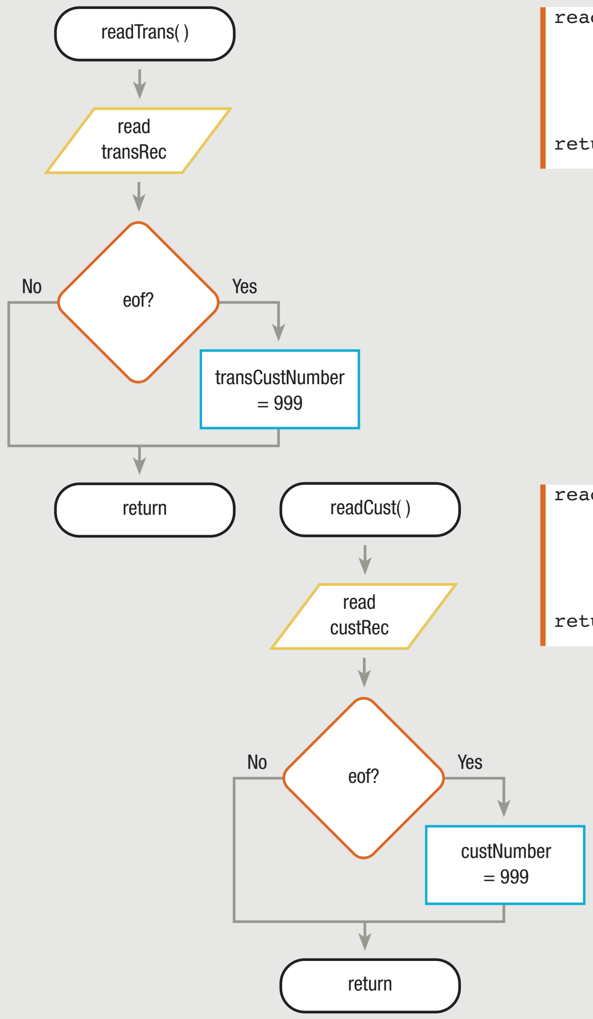
```
custRec
   num custNumber
   char custName
   char custAddress
   char custPhone
   num custTotalSales
transRec
   num transNumber
   num transCustNumber
   num transDate
   num transAmount
char bothAtEof = "N"
```

```
housekeeping()
     declare variables
     open files
     perform readCust()
     perform readTrans()
     if custNumber = 999 then
           if transCustNumber = 999 then
                 bothAtEof = "Y"
           endif
     endif
return
```

TIP

In the file-merging program earlier in this chapter, you placed "ZZZZZ" in the customer name field at the end of the file because character fields were being compared. In this example, because you are using numeric fields (customer numbers), you can store 999 in them at the end of the file. The value 999 is the highest possible numeric value for a three-digit number in the customer number field.

FIGURE 11-15: THE `readTrans()` AND `readCust()` MODULES FOR THE FILE-MATCHING PROGRAM



```
readTrans()
    read transRec
    if eof then
       transCustNumber = 999
    endif
return
```

```
readCust()
    read custRec
    if eof then
       custNumber = 999
    endif
return
```

In the file-merging program, your first action in the mainline `mergeFiles()` module was to determine which file held the record with the lower value; then, you wrote that file to output. In a main module within a matching program, you need to determine more than 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 `custTotalSales` field only if the transaction record `transCustNumber` field contains an exact match for the customer number in the master file record. Therefore, in the file-matching module (called `matchFiles()` in this example), you compare `custNumber` from `custRec` and `transCustNumber` from `transRec`. Three possibilities exist:

- The `transCustNumber` value equals `custNumber`.
- The `transCustNumber` value is higher than `custNumber`.
- The `transCustNumber` value is lower than `custNumber`.

When you compare records from the two input files, if `custNumber` and `transCustNumber` are equal, you add `transAmount` to `custTotalSales`, 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.

TIP The logic used here assumes there can be only one transaction per customer. Later in this chapter, you will develop the logic for a program in which the customer can have multiple transactions.

If `transCustNumber` is higher than `custNumber`, there wasn't a sale for that customer. That's all right; not every customer makes a transaction every period. If `transCustNumber` is higher than `custNumber` when you compare records, you simply write the original customer record to output 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.

Finally, when you compare records from the master and transaction files, if `transCustNumber` is lower than `custNumber` in the master file, you are trying to record a transaction for which no master record exists. That means 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.

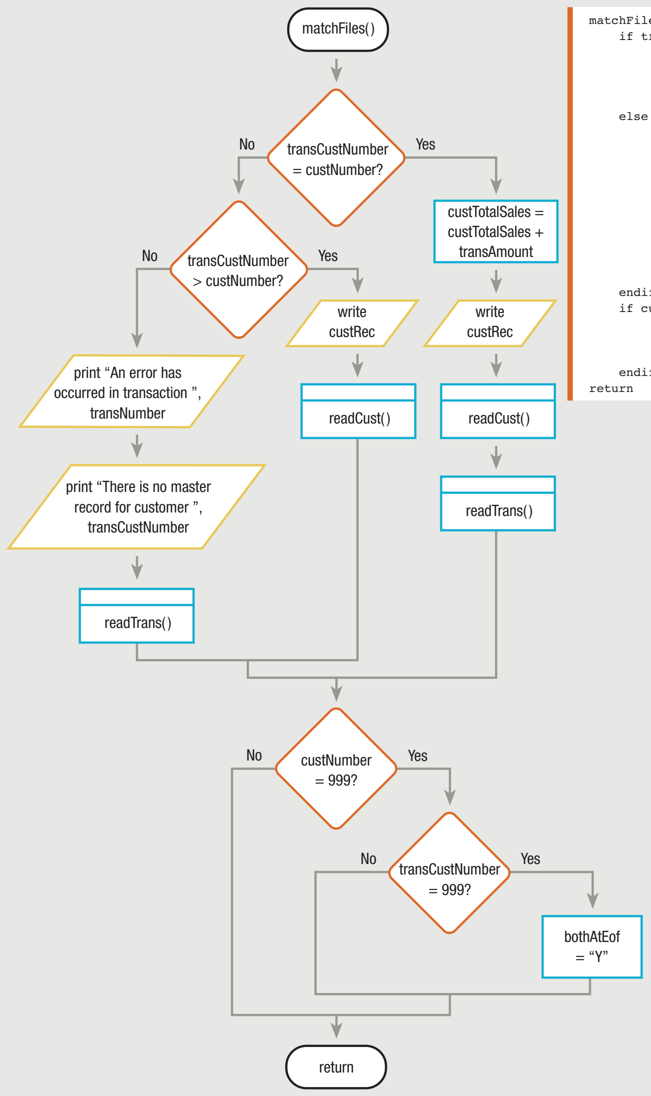
Whether `transCustNumber` was higher than, lower than, or equal to `custNumber`, at the bottom of the `matchFiles()` module you check whether both `custNumber` and `transCustNumber` are 999; when they are, you set the `bothAtEof` flag to "Y".

Figure 11-16 shows some sample data you can use to walk through the logic for this program, and Figure 11-17 shows the pseudocode and flowchart.

FIGURE 11-16: SAMPLE DATA FOR THE FILE-MATCHING PROGRAM

Master File		**Transaction File**	
custNumber	custTotalSales	transCustNumber	transAmount
100	1000.00	100	400.00
102	50.00	105	700.00
103	500.00	108	100.00
105	75.00	110	400.00
106	5000.00		
109	4000.00		
110	500.00		

FIGURE 11-17: THE `matchFiles()` MODULE LOGIC FOR THE FILE-MATCHING PROGRAM


matchFiles()
transCustNumber = custNumber?
No
Yes
custTotalSales = custTotalSales + transAmount
transCustNumber > custNumber?
No
Yes
write custRec
write custRec
print "An error has occurred in transaction ", transNumber
readCust()
readCust()
print "There is no master record for customer ", transCustNumber
readTrans()
readTrans()
custNumber = 999?
No
Yes
transCustNumber = 999?
No
Yes
bothAtEof = "Y"
return

```
matchFiles()
    if transCustNumber = custNumber then
          custTotalSales = custTotalSales + transAmount
          write custRec
          perform readCust()
          perform readTrans()
    else
          if transCustNumber > custNumber then
                write custRec
                perform readCust()
          else
                print "An error has occurred in transaction ",
                   transNumber
                print "There is no master record for customer ",
                   transCustNumber
                perform readTrans()
          endif
    endif
    if custNumber = 999 then
          if transCustNumber = 999 then
                bothAtEof = "Y"
          endif
    endif
return
```

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.

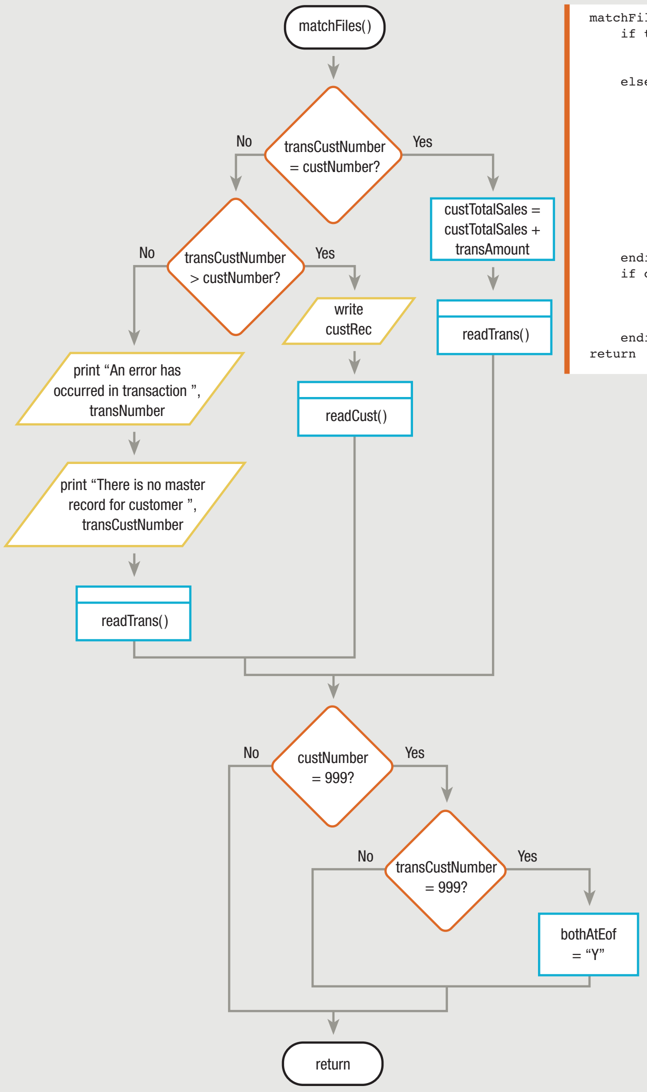
2. The customer number in the master file is 102 and the customer number in the transaction file is 105, so there are no transactions today for customer 102. Write the master record exactly the way it came in, and read a new master record.
3. Now, the master customer number is 103 and the transaction 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 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 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 figure, and write a new record with a 900.00 value in the `custTotalSales` 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.

ALLOWING MULTIPLE TRANSACTIONS FOR A SINGLE MASTER FILE RECORD

In the last example, the logic provided for, at most, one transaction record per master customer record. You would use very similar logic if you wanted to allow multiple transactions for a single customer. Figure 11-18 shows the new logic. A small but important difference exists between logic that allows multiple transactions and logic that allows only a single transaction per master file record. If a customer can have multiple transactions, whenever a transaction matches a customer, you add the transaction amount to the master total sales field. Then, you read *only* from the transaction file. After you exit `mainLoop()` and reenter it, the next transaction might also pertain to the same master customer. (Compare the first "Yes" branch in Figure 11-18 with the one in Figure 11-17; the `readCust()`

module is removed in Figure 11-18.) Only when a transaction number is greater than a master file customer number do you write the customer master record.

FIGURE 11-18: THE `matchFiles()` LOGIC ALLOWING MULTIPLE TRANSACTIONS FOR EACH MASTER FILE RECORD


matchFiles()
transCustNumber = custNumber?
No
Yes
custTotalSales = custTotalSales + transAmount
readTrans()
transCustNumber > custNumber?
No
Yes
write custRec
readCust()
print "An error has occurred in transaction ", transNumber
print "There is no master record for customer ", transCustNumber
readTrans()
custNumber = 999?
No
Yes
transCustNumber = 999?
No
Yes
bothAtEof = "Y"
return

```
matchFiles()
    if transCustNumber = custNumber then
          custTotalSales = custTotalSales + transAmount
          perform readTrans()
    else
          if transCustNumber > custNumber then
                write custRec
                perform readCust()
          else
                print "An error has occurred in transaction ",
                      transNumber
                print "There is no master record for customer ",
                      transCustNumber
                perform readTrans()
          endif
    endif
    if custNumber = 999 then
          if transCustNumber = 999 then
                bothAtEof = "Y"
          endif
    endif
return
```

UPDATING RECORDS IN SEQUENTIAL FILES

In the example in the preceding section, you needed to update a field in some of the records in a master file with new data. A more sophisticated update program allows you not only to make changes to data in a master file record, but also to update a master file either by adding new records or by eliminating the ones you no longer want.

Assume you have a master employee file, as shown on the left side of Figure 11-19. Sometimes, a new employee is hired and a record must be added to this file, or an employee quits and the employee's record must be removed from the file. Sometimes, you need to change an employee record by recording a raise in salary, for example, or a change of department.

For this kind of update program, it's common to have a transaction file in which each record contains all the same fields as the master file records do, with one exception. The transaction file has one extra field to indicate whether this transaction is meant to be an addition, a deletion, or a change—for example, a one-letter code of "A", "D", or "C". Figure 11-19 shows the master and transaction file layouts.

FIGURE 11-19: MASTER AND TRANSACTION FILES FOR THE UPDATE PROGRAM

```
MASTER AND TRANSACTION FILES FOR THE UPDATE PROGRAM
File name: EMPREC                  File name: TRANSREC
FIELD DESCRIPTION   DATA TYPE      FIELD DESCRIPTION   DATA TYPE
Employee number     Numeric        Employee number     Numeric
Name                Character      Name                Character
Salary              Numeric        Salary              Numeric
Department          Numeric        Department          Numeric
                                   Transaction code    Character
```

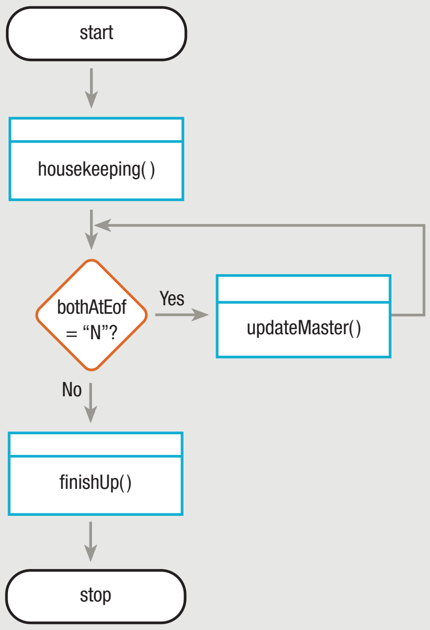
The master file records contain data in each of the fields shown in Figure 11-19—an employee number, name, salary, and department number. The three types of transaction records stored in the transaction file would differ as follows:

- An **addition record** in a transaction file actually represents a new master file record. An addition record would contain data in each of the fields—the employee number, name, salary, and department; because an addition record represents a new employee, data for all the fields must be captured for the first time. Also, in this example, such a record contains an "A" for "Addition" in the transaction code field.
- A **deletion record** in a transaction file flags a master file record that should be removed from the file. In this example, a deletion record really needs data in only two fields—a "D" for "Deletion" in the transaction code field and a number in the employee number field. If a "D" transaction record's employee number matches an employee number on a master record, then you have identified a record you want to delete. You do not need data indicating the salary, department, or anything else for a record you are deleting.

- A **change record** indicates an alteration that should be made to a master file record. In this case, it contains a "C" code for "Change" and needs data only in the employee number field and any fields that are going to be changed. In other words, if an employee's salary is not changing, the salary field in the transaction record will be blank; but if the employee is transferring to Department 28, then the department field of the transaction record will hold a 28.

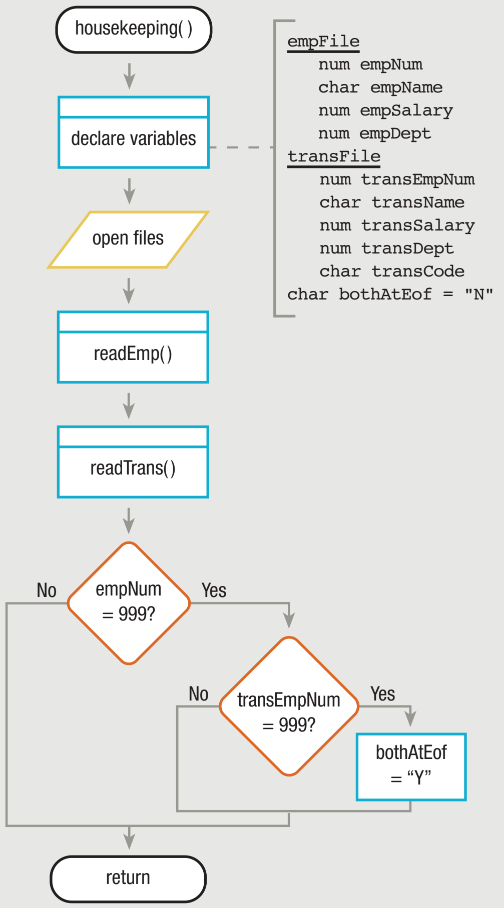
The mainline logic for an update program is very similar to the merging and matching programs shown in Figures 11-3 and 11-13, respectively. After `housekeeping()` and before `finishUp()`, the module that does the real work of the program executes repeatedly. Within the `housekeeping()` module, you declare the variables, open the files, and read the first record from each file. You can use the `readEmp()` and `readTrans()` modules to set the key fields `empNum` and `transEmpNum` to high values at `eof`. See Figures 11-20, 11-21, and 11-22.

FIGURE 11-20: THE MAINLINE LOGIC FOR THE UPDATE PROGRAM



```
start
     perform housekeeping()
     while bothAtEof = "N"
            perform updateMaster()
     endwhile
     perform finishUp()
stop
```

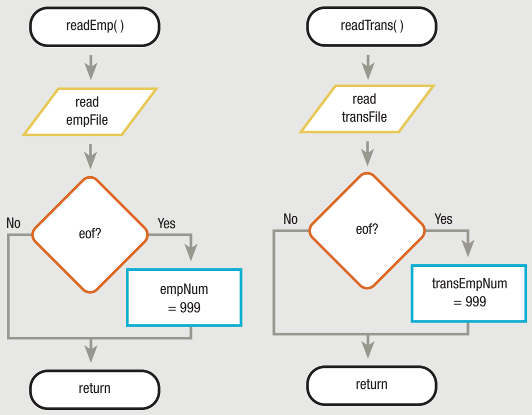
FIGURE 11-21: THE `housekeeping()` MODULE FOR THE UPDATE PROGRAM



```
empFile
    num empNum
    char empName
    num empSalary
    num empDept
transFile
    num transEmpNum
    char transName
    num transSalary
    num transDept
    char transCode
char bothAtEof = "N"
```

```
housekeeping()
     declare variables
     open files
     perform readEmp()
     perform readTrans()
     if empNum = 999 then
          if transEmpNum = 999 then
               bothAtEof = "Y"
          endif
     endif
return
```

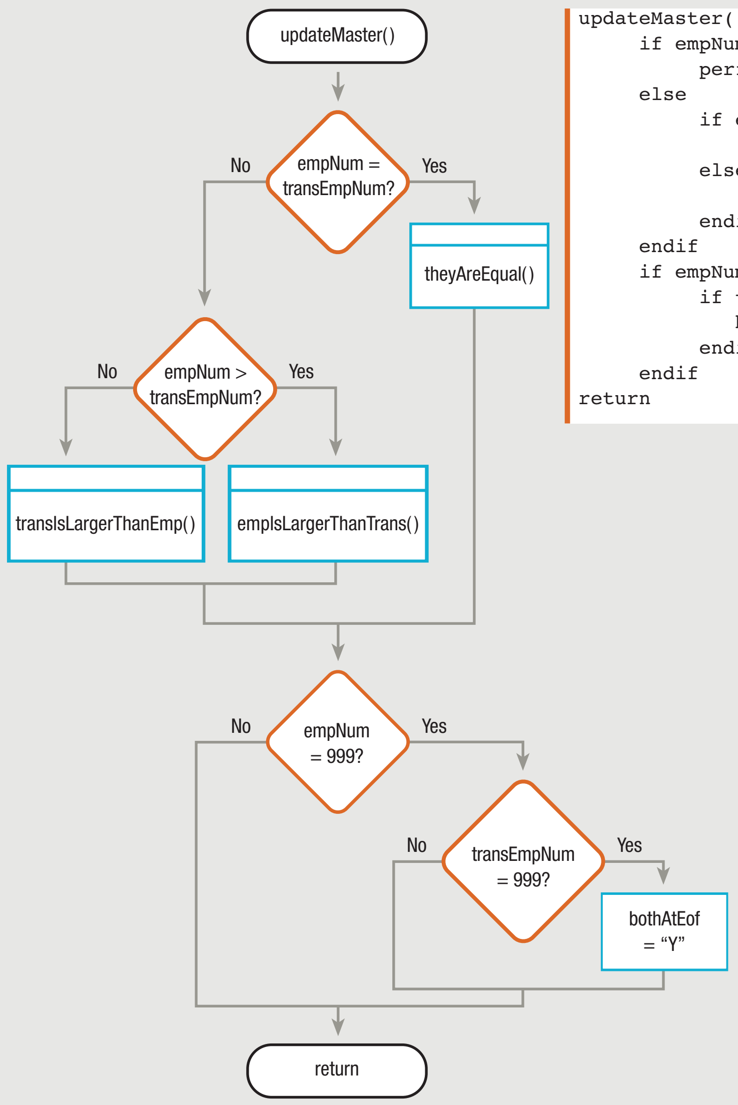
FIGURE 11-22: THE `readEmp()` AND `readTrans()` MODULES FOR THE UPDATE PROGRAM



```
readEmp()
     read empFile
     if eof then
          empNum = 999
     endif
return
readTrans()
     read transFile
     if eof then
          transEmpNum = 999
     endif
return
```

The `updateMaster()` module of the update program begins like the `matchFiles()` module in the matching program. You need to know whether `empNum` in the master file and `transEmpNum` in the transaction file are equal, or, if not, then you need to know which value is higher. To keep the `updateMaster()` module simple, you can create modules for each of the three possible scenarios: `theyAreEqual()`, `empIsLargerThanTrans()`, and `transIsLargerThanEmp()`. (Of course, you might choose shorter module names; the long names used here are intended to help you remember what condition preceded the execution of each module.) At the end of the `updateMaster()` module, after one of the three submodules has finished, you can set the `bothAtEof` flag variable to "Y" if both files have completed. Figure 11-23 shows the `updateMaster()` module.

FIGURE 11-23: THE `updateMaster()` MODULE OF THE UPDATE PROGRAM



```
updateMaster()
     if empNum = transEmpNum then
          perform theyAreEqual()
     else
          if empNum > transEmpNum then
             perform empIsLargerThanTrans()
          else
             perform transIsLargerThanEmp()
          endif
     endif
     if empNum = 999 then
          if transEmpNum = 999 then
             bothAtEof = "Y"
          endif
     endif
return
```

You perform the `theyAreEqual()` module only if a record in the master file and a record in the transaction file contain the same employee number. This should be the situation when a change is made to a record (for example, a change in salary) or when a record is to be deleted. If the master file and the transaction file records are equal, but the `transCode` value in the transaction record is an “A”, then an error has occurred. You should not attempt to add a full employee record when the employee already exists in the master file.

As shown in Figure 11-24, within the `theyAreEqual()` module, you check `transCode` and perform one of three actions:

- If the code is an “A”, print an error message. But what is the error? (Is the code wrong? Was this meant to be a change or a deletion of an existing employee? Is the employee number wrong—was this meant to be the addition of some new employee?) Because you’re not completely sure, you can only print an error message to let an employee know that an error has occurred; then, the employee can handle the error. You should also write the existing master record to output exactly the same way it came in, without making any changes.
- If the code is a “C”, you need to make changes. You must check each field in the transaction record. If any field is blank, the data in the new master record should come from the old master record. If, however, a field in the transaction record contains data, this data is intended to constitute a change, and the corresponding field in the new master record should be created from the transaction record. Then, for each changed field, you replace the contents of the old field in the master file with the new value in the corresponding field in the transaction file, and then write the master file record.
- If the code is not an “A” or a “C”, it must be a “D” and the record should be deleted. How do you delete a record from a new master file? Just don’t write it out to the new master file! In other words, as Figure 11-24 shows, no action is necessary when a record is deleted.

TIP

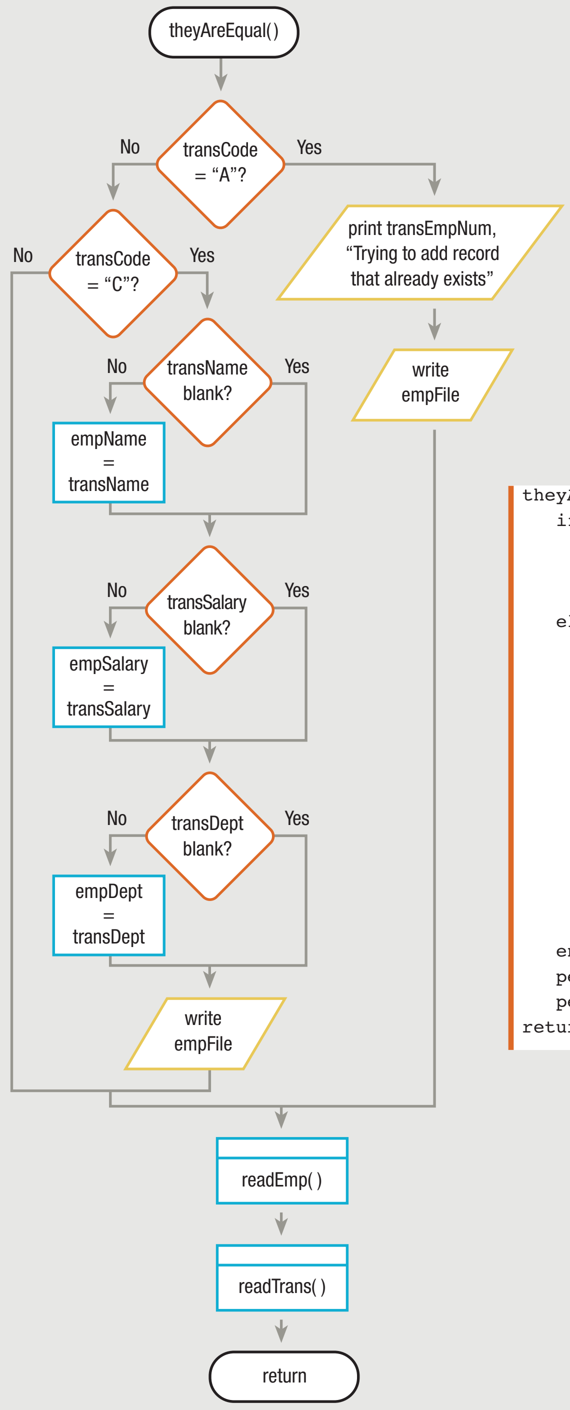
Various programming languages have different ways of checking a field to determine if it is blank. In some languages, you compare the field to an empty string, as in `transName = ""`. The quotation marks with nothing between them indicate an empty or null string. In some systems, you might need to compare the field to a space character, as in `transName = " "`, in which a literal space is inserted between the quotation marks. In other languages, you can use a predefined language-specific constant such as `BLANK`, as in `transName = BLANK`.

TIP

To keep the illustration simple here, you can assume that all the transaction records have been checked by a previous program, and all `transCode` values are “A”, “C”, or “D”. If this were not the case, you could simply add one more decision to the `theyAreEqual()` module. If `transCode` is not “C”, instead of assuming it is “D”, ask if it is “D”. If so, delete the record (by not writing it); if not, it must be something other than “A”, “C”, or “D”, so print an error message.

Finally, at the end of the `theyAreEqual()` module, after the appropriate action has been taken with the matching master and transaction file records, you read one new record from each of the two input files.

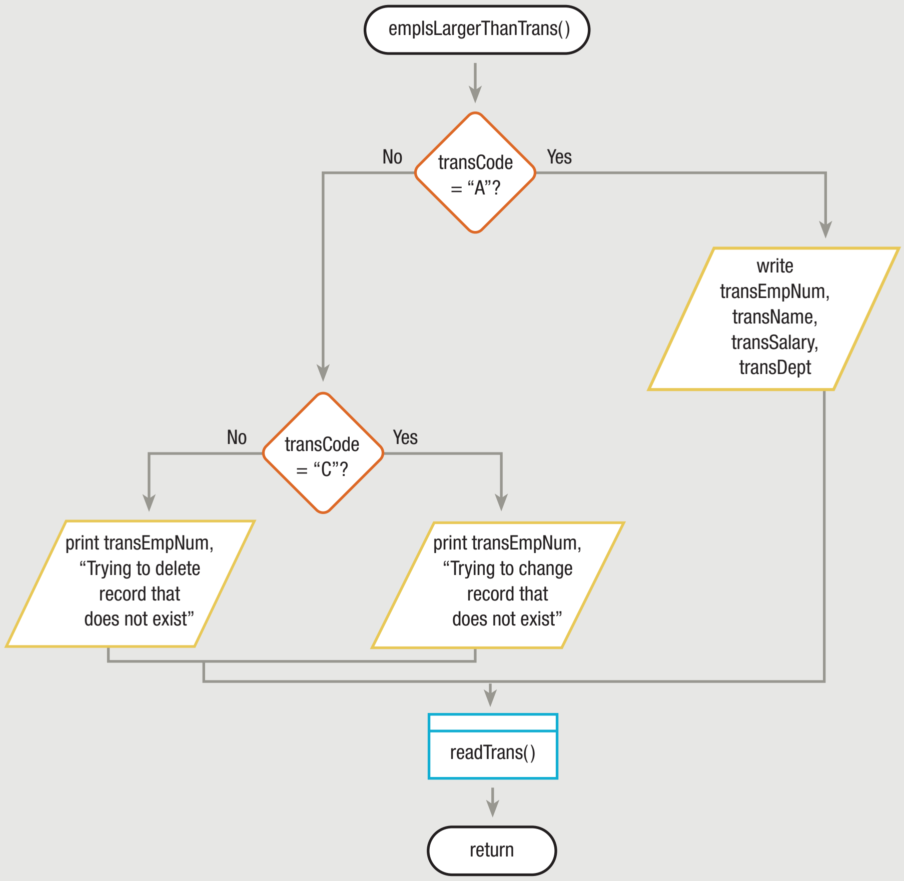
FIGURE 11-24: THE `theyAreEqual()` MODULE FOR THE UPDATE PROGRAM



```
theyAreEqual()
   if transCode = "A" then
         print transEmpNum, "Trying to add
             record that already exists"
         write empFile
   else
        if transCode = "C" then
             if transName not blank then
                  empName = transName
             endif
             if transSalary not blank then
                  empSalary = transSalary
             endif
             if transDept not blank then
                  empDept = transDept
             endif
             write empFile
        endif
   endif
   perform readEmp()
   perform readTrans()
return
```

Suppose that within the `updateMaster()` module in Figure 11-23, the master file record and the transaction file record do *not* match. If the master file record has a higher number than the transaction file record, this means you have read a transaction record for which there is no master file record, so you execute the `empIsLargerThanTrans()` module. See Figure 11-25.

FIGURE 11-25: THE `empIsLargerThanTrans()` MODULE


empIsLargerThanTrans()
No
transCode = "A"?
Yes
write transEmpNum, transName, transSalary, transDept
No
transCode = "C"?
Yes
print transEmpNum, "Trying to delete record that does not exist"
print transEmpNum, "Trying to change record that does not exist"
readTrans()
return

```
empIsLargerThanTrans()
   if transCode = "A" then
      write transEmpNum, transName, transSalary, transDept
   else
      if transCode = "C" then
         print transEmpNum, "Trying to change record that does not exist"
      else
         print transEmpNum, "Trying to delete record that does not exist"
      endif
   endif
   perform readTrans()
return
```

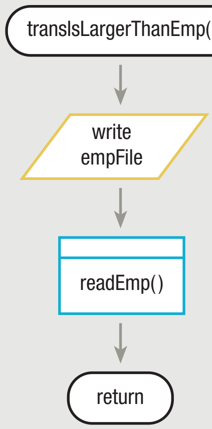
Within the `empIsLargerThanTrans()` module, if the transaction record contains code "A", that's fine because an addition transaction shouldn't have a master record. The transaction record data simply become the data for the new master record, so each of its fields is written to the new output file.

However, if the transaction code is "C" or "D" in the `empIsLargerThanTrans()` module, an error has occurred. Either you are attempting to make a change to a record that does not exist or you are attempting to delete a record that does not exist. Either way, a mistake has been made, and you must print an error message.

At the end of the `empIsLargerThanTrans()` module, you should not read another master file record. After all, there could be several more transactions that represent new additions to the master file. You want to keep reading transactions until a transaction matches or is greater than a master record. Therefore, only a transaction record should be read.

The final possibility in the `updateMaster()` module in Figure 11-23 is that a master file record's `empNum` field is smaller than the transaction file record's `transEmpNum` field in memory. If there is no transaction for a given master file record, it just means that the master file record has no changes or deletions; therefore, when you perform the `transIsLargerThanEmp()` module, you simply write the new master record out exactly like the old master record and read another master record. See Figure 11-26.

FIGURE 11-26: THE `transIsLargerThanEmp()` MODULE



```
transIsLargerThanEmp()
     write empFile
     perform readEmp()
return
```

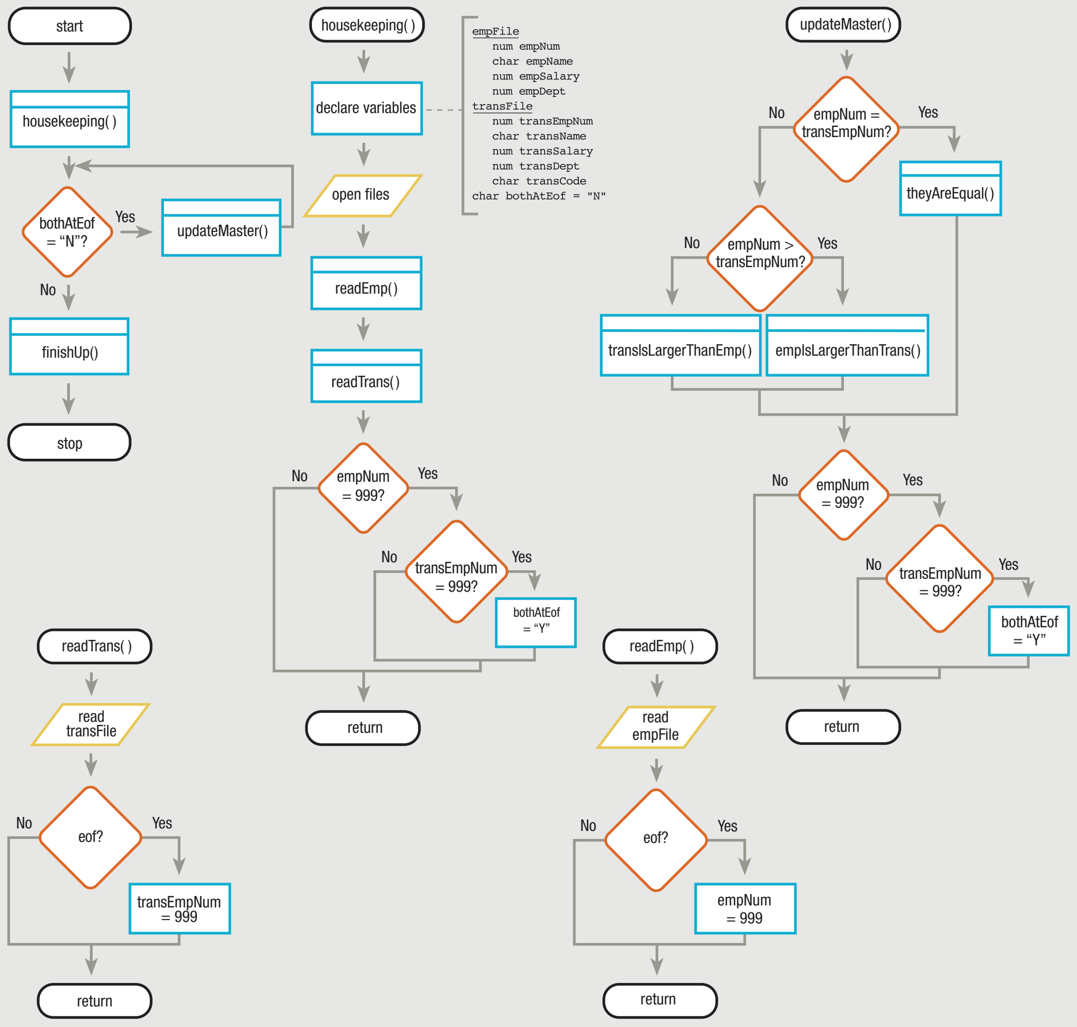
At some point, one of the files will reach `eof`. If the transaction file reaches the end first, `transEmpNum` is set to 999 in the `readTrans()` module. Each time the `updateMaster()` module is entered after `transEmpNum` is set to 999, `empNum` will be lower than `transEmpNum` and the `transIsLargerThanEmp()` module will execute. That module writes records from the master file without alteration, and this is exactly what you want to happen. Obviously, there were no transactions for these final records in the master file, because all the records in the transaction file were used to apply to earlier master file records.

On the other hand, if the master file reaches its end first, `empNum` is set to 999 in the `readEmp()` module. Now, each time the program enters the `updateMaster()` module, `transEmpNum` will be lower than `empNum`. The `empIsLargerThanTrans()` module will execute for all remaining transaction records. In turn, each remaining transaction will be compared to the possible code values. If any remaining transaction records are additions, they will write to the new master as new records. However, if the remaining transaction records represent changes or deletions, a mistake has been made, because there are no corresponding master file records. In other words, error messages will then be printed for the remaining change and deletion transaction records as they go through the `updateMaster()` process.

Whichever file reaches the end first, the other continues to be read and processed. When that file reaches `eof`, the `bothAtEof` flag will finally be set to "Y". Then, you can perform the `finishUp()` module, as shown with the complete program in Figure 11-27.

Merging files, matching files, and updating a master file from a transaction file require a significant number of steps, because as you read each new input record you must account for many possible scenarios. Planning the logic for programs like these takes a fair amount of time, but by planning the logic carefully, you can create programs that perform valuable work for years to come. Separating the various outcomes into manageable modules keeps the program organized and allows you to develop the logic one step at a time.

FIGURE 11-27: COMPLETE PROGRAM THAT UPDATES MASTER FILE USING TRANSACTION RECORDS THAT CONTAIN ADD, CHANGE, OR DELETE CODES



```
empFile
    num empNum
    char empName
    num empSalary
    num empDept
transFile
    num transEmpNum
    char transName
    num transSalary
    num transDept
    char transCode
char bothAtEof = "N"
```

```
start
     perform housekeeping()
     while bothAtEof = "N"
        perform updateMaster()
     endwhile
     perform finishUp()
stop
```

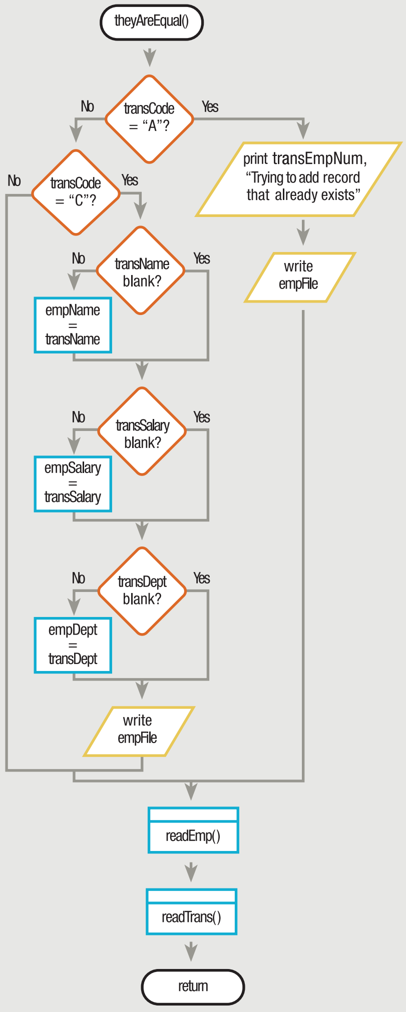
```
housekeeping()
     declare variables
     open files
     perform readEmp()
     perform readTrans()
     if empNum = 999 then
          if transEmpNum = 999 then
               bothAtEof = "Y"
          endif
     endif
return
```

```
updateMaster()
     if empNum = transEmpNum then
          perform theyAreEqual()
     else
          if empNum > transEmpNum then
               perform empIsLargerThanTrans()
          else
               perform transIsLargerThanEmp()
          endif
     endif
     if empNum = 999 then
          if transEmpNum = 999 then
               bothAtEof = "Y"
          endif
     endif
return
```

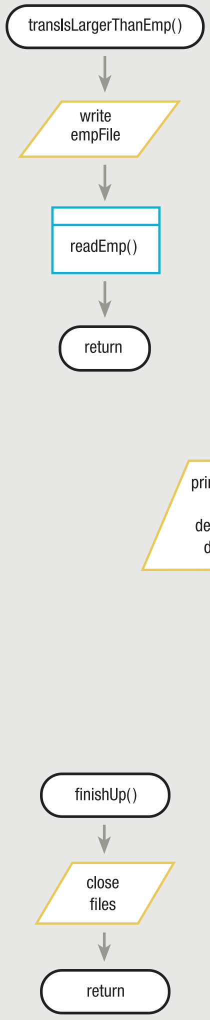
```
readTrans()
     read transFile
     if eof then
          transEmpNum = 999
     endif
return
```

```
readEmp()
     read empFile
     if eof then
          empNum = 999
     endif
return
```

FIGURE 11-27: COMPLETE PROGRAM THAT UPDATES MASTER FILE USING TRANSACTION RECORDS THAT CONTAIN ADD, CHANGE, OR DELETE CODES (CONTINUED)

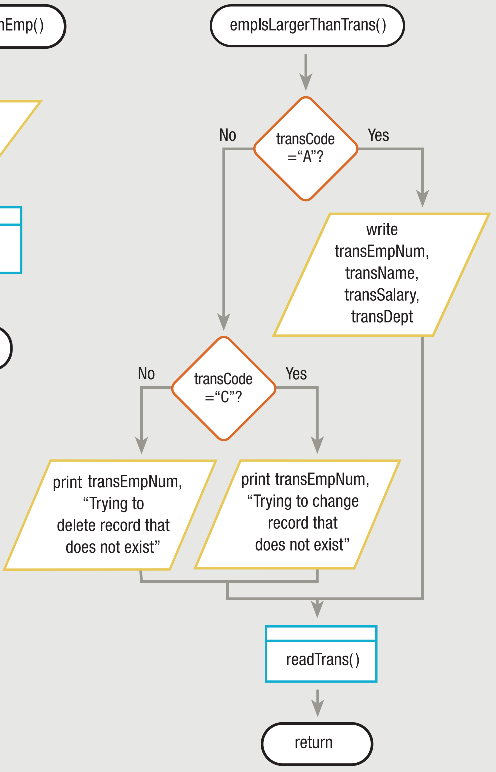


```
theyAreEqual()
     if transCode = "A" then
          print transEmpNum, "Trying to add
             record that already exists"
          write empfile
     else
          if transCode = "C" then
                 if transName not blank then
                      empName = transName
                 endif
                 if transSalary not blank then
                      empSalary = transSalary
                 endif
                 if transDept not blank then
                      empDept = transDept
                 endif
                 write empFile
          endif
     endif
     perform readEmp()
     perform readTrans()
return
```



```
transIsLargerThanEmp()
      write empFile
      perform readEmp()
return
```

```
finishUp()
      close files
return
```


empIsLargerThanTrans()
transCode ="A"?
No
Yes
write transEmpNum, transName, transSalary, transDept
transCode ="C"?
No
Yes
print transEmpNum, "Trying to delete record that does not exist"
print transEmpNum, "Trying to change record that does not exist"
readTrans()
return

```
empIsLargerThanTrans()
    if transCode = "A" then
       write transEmpNum, transName,
          transSalary, transDept
    else
       if transCode = "C" then
          print transEmpNum, "Trying to change
             record that does not exist"
       else
          print transEmpNum, "Trying to delete
             record that does not exist"
       endif
    endif
    perform readTrans()
return
```

CHAPTER SUMMARY

- A sequential file is a file whose records are stored one after another in some order. The field on which you sort records is the key field. Merging files involves combining two or more files while maintaining the sequential order. Each file used in a merge must be sorted in the same order in the same field as the others.
- The mainline logic for a program that merges two files contains a housekeeping module, a module that matches files and repeats until the end of the program, and a final module that performs finishing tasks. The mainline logic checks a flag variable that is turned on when both input files are finished.
- When beginning the repeating module that merges files in a merge program, you compare records from each input file. You write the appropriate record from one of the files to output, and then read a record from the same file. When you encounter `eof` on one of the two input files, set the field on which the merge is based to a high value.
- You use a master file to hold relatively permanent data, and a transaction file to hold more temporary data that corresponds to records in the master file. When you update a master file, you can take two approaches: you can actually change the information in the master file, or you can create a copy of the master file, making the changes in the new version.
- The logic you use to perform a match between master and transaction file records involves comparing the files to determine whether there is a transaction for each master record; when there is, you update the master record. When a master record has no transaction, you write the master record as is; when a transaction record has no corresponding master, you have an error.
- Using the logic that allows multiple transactions per master file record, whenever a transaction matches a master file record, you process the transaction and then you read only from the transaction file. Only when a transaction file key field is greater than a master file key field do you write the master record.
- A sophisticated update program allows you to make changes to data in a record and update a master file by adding new records or eliminating records you no longer want. For this kind of program, it's common to have a transaction file in which each record contains all the same fields as the master file, with an additional code that indicates the type of transaction.

KEY TERMS

A **sequential file** is a file in which records are stored one after another in some order.

Records that are stored in **temporal order** are stored in order based on their creation time.

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

A **data file** contains only data for another computer program to read, not headings or other formatting.

A **high value** is one that is greater than any possible value in a field.

You use a **master file** to hold relatively permanent data.

A **transaction file** holds more temporary data generated by the entities represented in the master file.

A **matching record** is a transaction file record that contains data about the same entity in a master file record.

To **update a master file** means to make changes to the values in its fields based on transaction records.

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

An **addition record** in a transaction file is one that represents a new master record.

A **deletion record** in a transaction file flags a record that should be removed from a master file.

A **change record** in a transaction file indicates an alteration that should be made to a master file record.

REVIEW QUESTIONS

1. **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

2. **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

3. **When you write a program that combines two sorted files into one larger, sorted file, you must create an additional work variable whose purpose is to ________.**
    a. count the files
    b. count the records
    c. flag when both files encounter `eof`
    d. flag when two records in the files contain identical data

4. **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

5. **In a program that merges two sorted files, the first task in the main `mergeFiles()` module is to ________.**

   a. read a record from each input file
   b. compare the values of the fields on which the files are sorted
   c. output the record with the lower value in the field on which the files are sorted
   d. output the record with the higher value in the field on which the files are sorted

6. **Assume 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

7. **A value that is greater than any possible legal value in a field is called a(n) ________ value.**

   a. great
   b. illegal
   c. merging
   d. high

8. **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

9. **A file that holds more permanent data than a transaction file is a ________ file.**

   a. master
   b. primary
   c. key
   d. mega-

10. **A transaction file is often used to ________ another file.**

   a. augment
   b. remove
   c. verify
   d. update

11. 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

12. Ambrose Bierce High School maintains a master file containing records for each student in a class. Each record contains fields such as student ID, student name, home phone number, and grade on each exam. A transaction file is created after each exam; it contains records that each hold a test number, a student ID, and the student's grade on the exam. You would write a matching program to match the records in the __________ field.

a. student ID
b. student name
c. test number
d. grade on the exam

13. Larry's Service Station maintains a master file containing records for each vehicle Larry services. Each record contains fields such as vehicle ID, owner name, date of last oil change, date of last tire rotation, and so on. A transaction file is created each day; it contains records that hold a vehicle ID and the name of the service performed. When Larry performs a match between these two files so that the most recent date can be inserted into the master file, which of the following should cause an error condition?

a. A specific vehicle is represented in each file.
b. A specific vehicle is represented in the master file, but not in the transaction file.
c. A specific vehicle is represented in the transaction file, but not in the master file.
d. A specific vehicle is not represented in either file.

14. Sally's Sandwich Shop maintains a master file containing records for each preferred customer. Each record contains a customer ID, name, e-mail address, and number of sandwiches purchased. A transaction file record is created each time a customer makes a purchase; the fields include customer ID and number of sandwiches purchased as part of the current transaction. After a customer surpasses 12 sandwiches, Sally e-mails the customer a coupon for a free sandwich. When Sally runs the match program with these two files so that the master file can be updated with the most recent purchases, which of the following should indicate an error condition?

a. master ID is greater than transaction ID
b. master ID is equal to transaction ID
c. master ID is less than transaction ID
d. none of the above

15. **Which of the following is true of master-transaction file-matching processing?**
    a. A master file record must never match more than one transaction record.
    b. A transaction file record must never match any master records.
    c. When master and transaction file records match, you must always immediately read another record from each file.
    d. A transaction record must match, at most, one master file record.

16. **Which of the following is true of master-transaction file-matching processing?**
    a. A master file's records must be sorted in some sequential order.
    b. A transaction file's records must be sorted on a different field than the master file's records.
    c. A master file's records must contain more fields than a transaction file's records.
    d. A transaction file's records must contain more fields than a master file's records.

17. **In a program that updates a master file, a transaction file record might cause a master file record to be ____________.**
    a. modified
    b. deleted
    c. either of these
    d. neither a nor b

18. **In a program that updates a master file, if a transaction record indicates a change, then it is an error when the transaction record's matching field is ____________ the field in a master file's record.**
    a. greater than
    b. less than
    c. both of these
    d. neither a nor b

19. **In a program that updates a master file, if a master and transaction file match, then it is an error if the transaction record is a(n) ____________ record.**
    a. addition
    b. change
    c. deletion
    d. two of the above

20. **In a program that updates a master file, if a master file's comparison field is larger than a transaction file's comparison field, then it is an error if the transaction record is a(n) ____________ record.**
    a. addition
    b. change
    c. deletion
    d. two of the above

FIND THE BUGS

The following pseudocode contains one or more bugs that you must find and correct.

1. **Each time a salesperson sells a car at the Pardeeville New and Used Auto Dealership, a record is created containing the salesperson's name and the amount of the sale. Sales of new and used cars are kept in separate files because several reports are created for one type of sale or the other. However, management has requested a merged file so that all of a salesperson's sales, whether the vehicle was new or used, are displayed together. The following code is intended to merge the files that have already been sorted by salesperson ID number.**

```
start
     perform housekeeping()
     while bothAtEOF = "Y"
          perform mergeModule()
     endwhile
     perform finish()
return

housekeeping()
     declare variables
         newFile
           char newIdNumber
           num newSalePrice
         usedFile
           char usedIdNumber
           num usedSalePrice
         char bothAtEof
     open files
     read newFile
     if eof then
          newIdNumber = 9999999
     endif
     read usedFile
     if eof then
          usedIdNumber = 9999
     endif
     if newIdNumber = 9999999 then
          if usedIdNumber = 9999999 then
               bothAtEof = "Y"
          endif
     endif
return
```

```
mergeModule()
     if newIdNumber = usedIdNumber then
          write newIdNumber, newSalePrice
          read newFile
          if eof then
               newIdNumber = 9999999
          endif
     else
          write usedIdNumber, usedSalePrice
          read usedFile
          if eof then
               usedIdNumber = 999
          endif
     endif
     if newIdNumber = 9999999 then
          if usedIdNumber = 9999999 then
               bothAtEof = "X"
          endif
     endif
return

finish()
     close files
return
```

EXERCISES

1. **The Springwater Township School District has two high schools—Jefferson and Audubon. Each school maintains a student file with fields containing student ID, last name, first name, and address. Each file is in student ID number order. Write the flowchart or pseudocode for a program that merges the two files into one file containing a list of all students in the district, maintaining student ID number order.**

2. **The Redgranite Library keeps a file of all books borrowed every month. Each file is in Library of Congress number order and contains additional fields for author and title.**
   a. Write the flowchart or pseudocode for a program that merges the files for January and February to create a list of all books borrowed in the two-month period.
   b. Modify the program from Exercise 2a so that if there is more than one record for a book number, you print the book information only once.
   c. Modify the program from Exercise 2b so that if there is more than one record for a book number, you not only print the book information only once, you print a count of the total number of times the book was borrowed.

3. **Hearthside Realtors keeps a transaction file for each salesperson in the office. Each transaction record contains the salesperson's first name, date of the sale, and sale price. The records for the year are sorted in descending sale price order. Two salespeople, Diane and Mark, have formed a partnership. Write the flowchart or pseudocode that produces a merged list of their transactions (including name of salesperson, date, and price) in descending order by price.**

4. **Dartmoor Medical Associates maintains two patient files—one for the Lakewood office and one for the Hanover office. Each record contains the name, address, city, state, and zip code of a patient, with the file maintained in zip code order. Write the flowchart or pseudocode that merges the two files to produce one master name and address file that the Dartmoor office staff can use for addressing the practice's monthly Healthy Lifestyles newsletter mailing in zip code order.**

5. **The Willmington Walking Club maintains a master file that contains a record for each of its members. Fields in the master file include the walker's ID number, first name, last name, and total miles walked to the nearest one-tenth of a mile. Every week, a transaction file is produced; the transaction file contains a walker's ID number and the number of miles the walker has logged that week. Each file is sorted in walker ID number order.**

   a. Create the flowchart or pseudocode for a program that matches the master and transaction file records and updates the total miles walked for each club member by adding the current week's miles to the cumulative total for each walker. Not all walkers submit walking reports each week. The output is the updated master file and an error report listing any transaction records for which no master record exists.

   b. Modify the program in Exercise 5a to print a certificate of achievement each time a walker exceeds the 500-mile mark. That is, the certificate—containing the walker's name and an appropriate congratulatory message—is printed during the run of the update program when a walker's mile total changes from a value below 500 to one that is 500 or greater.

6. **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 of the temporary employees it sends out on assignments. 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.

   a. Create the flowchart or pseudocode for a program that matches the master and transaction file records, and print one line for each transaction, indicating job number, employee ID number, hours worked, hourly rate, and gross pay. Assume each temporary worker works at most one job per week; print one line for each worker who has worked that week.

   b. Modify Exercise 6a so that any individual temporary worker can work any number of separate jobs in a week. Print one line for each job that week.

   c. Modify Exercise 6b so that, although any worker can work any number of jobs in a week, you accumulate the worker's total pay for all jobs and print one line per worker.

7. **Claypool College maintains a student master file that contains a student ID number, last name, first name, address, total credit hours completed, and cumulative grade point average for each of the students who attend the college. The file has been sorted in student ID number order.**

   Each semester, a transaction file is created with the student's ID, the number of credits completed during the new semester, and the grade point average for the new semester. The transaction file is also sorted in student ID order.

   Create the flowchart or pseudocode for a program that matches the master and transaction file records and updates the total credit hours completed and the cumulative grade point average on a new master record. Calculate the new grade point average as follows:

   - ☐ Multiply the credits in the master file by the grade point average in the master file, giving master honor points—that is, honor points earned prior to any transaction. The honor points value is useful because it is weighted—the value of the honor points is more for a student who has accumulated 100 credits with a 3.0 grade point average than it is for a student who has accumulated only 20 credits with a 3.0 grade point average.
   - ☐ Multiply the credits in the transaction file by the grade point average in the transaction file, giving transaction honor points.
   - ☐ Add the two honor point values, giving total honor points.
   - ☐ Add master and transaction credit hours, giving total credit hours.
   - ☐ Divide total honor points by total credit hours, giving the new grade point average.

8. **The Amelia Earhart High School basketball team maintains a record for each team player, including player number, first and last name, minutes played during the season, baskets attempted, baskets made, free throws attempted, free throws made, shooting average from the floor, and shooting average from the free throw line. (Shooting average from the floor is calculated by dividing baskets made by baskets attempted; free throw average is calculated by dividing free throws made by free throws attempted.) The master records are maintained in player number order.**

   After each game, a transaction record is produced for any player who logged playing time. Fields in each transaction record contain player number, minutes played during the game, baskets attempted, baskets made, free throws attempted, and free throws made.

   Design the flowchart or pseudocode for a program that updates the master file with the transaction file, including recalculating shooting averages, if necessary.

9. **The Tip-Top Talent Agency books bands for social functions. The agency maintains a master file in which the records are stored in order by band code. The records have the following format:**

TALENT FILE DESCRIPTION

File name: BANDS

FIELD DESCRIPTION	DATA TYPE	COMMENTS	EXAMPLE
Band Code	Numeric	3-digit number	176
Band Name	Character	20 characters	The Polka Pals
Contact Person	Character	20 characters	Jay Sakowicz
Phone	Numeric	10 digits	5554556012
Musical Style	Character	8 characters	Polka
Hourly Rate	Numeric	2 decimal places	75.00

The agency has asked you to write an update program, so that once a month the agency can make changes to the file, using transaction records with the same format as the master records, plus one additional field that holds a transaction code. The transaction code is "A" if the agency is adding a new band to the file, "C" if it is changing some of the data in an existing record, and "D" if it is deleting a band from the file.

An addition transaction record contains a band code, an "A" in the transaction code field, and the new band's data. During processing, an error can occur if you attempt to add a band code that already exists in the file. This is not allowed, and an error message is printed.

A change transaction record contains a band code, a "C" in the transaction code field, and data for only those fields that are changing. For example, a band that is raising its hourly rate from $75 to $100 would contain empty fields for the band name, contact person information, and style of music, but the hourly rate field would contain the new rate. During processing, an error can occur if you attempt to change data for a band number that doesn't exist in the master file; print an error message.

A deletion transaction record contains a band code, a "D" in the transaction code field, and no other data. During processing, an error can occur if you attempt to delete a band number that doesn't exist in the master file; print an error message.

Two forms of output are created. One is the updated master file with all changes, additions, and deletions. The other is a printed report of errors that occurred during processing. Rather than just a list of error messages, each line of the printed output should list the appropriate band code along with the corresponding message.

a. Design the print chart or create sample output for the error report.
b. Design the hierarchy chart for the program.
c. Create either a flowchart or pseudocode for the program.

10. **Cozy Cottage Realty maintains a master file in which records are stored in order by listing number, in the following format:**

```
REALTY FILE DESCRIPTION
File name: HOUSES
FIELD DESCRIPTION    DATA TYPE    COMMENTS          EXAMPLE
Listing Number       Numeric      6-digit number    200719
Address              Character    20 characters     348 Alpine Road
List Price           Numeric      0 decimals        139900
Bedrooms             Numeric      0 decimals        3
Baths                Numeric      1 decimal         1.5
```

The realty company has asked you to write an update program so that, every day, the company can make changes to the file, using transaction records with the same format as the master records, plus one additional field that holds a transaction code. The transaction code is "A" to add a new listing, "C" to change some of the data in an existing record, and "D" to delete a listing that is sold or no longer on the market.

An addition transaction record contains a listing number, an "A" in the transaction code field, and the new house listing's data. During processing, an error can occur if you attempt to add a listing number that already exists in the file. This is not allowed, and an error message is printed.

A change transaction record contains a listing number, a "C" in the transaction code field, and data for only those fields that are changing. For example, a listing that is dropping in price from $139,900 to $133,000 would contain empty fields for the address, bedrooms, and baths, but the price field would contain the new list price. During processing, an error can occur if you attempt to change data for a listing number that doesn't exist in the master file; print an error message.

A deletion transaction record contains a listing code number, a "D" in the transaction code field, and no other data. During processing, an error can occur if you attempt to delete a listing number that doesn't exist in the master file; print an error message.

Two forms of output are created. One is the updated master file with all changes, additions, and deletions. The other is a printed report of errors that occurred during processing. Rather than just a list of error messages, each line of the printed output should list the appropriate house listing number along with the corresponding message.

a. Design the print chart or create sample output for the error report.
b. Design the hierarchy chart for the program.
c. Create either a flowchart or pseudocode for the program.

11. Crown Greeting Cards maintains a master file of its customers stored in order by customer number, in the following format:

```
CROWN CUSTOMER FILE DESCRIPTION
File name: CUSTS
FIELD DESCRIPTION        DATA TYPE   COMMENTS           EXAMPLE
Customer Number          Numeric     5 digits           34492
Name                     Character   20 characters      Roberta Branch
Address                  Character   20 characters      32 Pinetree Lane
Phone Number             Numeric     10 digits          5554448935
Value of Merchandise
  Purchased This Year    Numeric     2 decimal places   525.99
```

The card store has asked you to write an update program so that, every week, the store can make changes to the file, using transaction records with the same format as the master records, plus one additional field that holds a transaction code. The transaction code is "A" to add a new customer, "C" to change some of the data in an existing record, and "D" to delete a customer. In a transaction record, the amount field represents a new transaction instead of the total value of merchandise purchased.

An addition transaction record contains a customer number, an "A" in the transaction code field, and the new customer's name, address, phone number, and first purchase amount. During processing, an error can occur if you attempt to add a customer number that already exists in the file. This is not allowed, and an error message is printed.

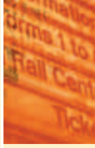
A change transaction record contains a customer number, a "C" in the transaction code field, and data for only those fields that are changing. For example, a customer might have a new address or phone number. In a change record, if a value appears in the merchandise value field, it represents an amount that should be added to the total merchandise value in the master record. During processing, an error can occur if you attempt to change data for a customer number that doesn't exist in the master file; print an error message.

A deletion transaction record contains a customer number, a "D" in the transaction code field, and no other data. During processing, an error can occur if you attempt to delete a customer number that doesn't exist in the master file; print an error message.

Three forms of output are created. One is the updated master file with all changes, additions, and deletions. The second output is a printed report of errors that occurred during processing. Rather than just list error messages, each line of the printed output should list the appropriate customer number along with the corresponding message. The third output is a report listing all customers who have currently met or exceeded the $1,000 purchase threshold for the year.

a. Design the print chart or create sample output for the error report, along with the hierarchy chart and either a flowchart or pseudocode for the program.

b. Modify the program in Exercise 11a so that the third output is not a report of all customers who have met or exceeded the $1,000 purchase threshold this year, but a report listing all customers who have just passed the $100 purchase threshold this week.



DETECTIVE WORK

1. What is a random file and how does it differ from a sequential file? In what types of applications are sequential files most useful? In what types of applications are they least useful?
2. What is FIFO and how does it relate to file processing?

UP FOR DISCUSSION

1. In Chapter 5, you considered criteria to use for a program that selects possible candidates for organ transplants. Suppose you are hired by a large hospital that has decided to avoid public criticism of how potential recipients are chosen; they will display recipients sequentially in alphabetical order. The hospital's doctors will consult this list if they have an organ that can be transplanted. If more than 10 patients are waiting for a particular organ, the first 10 are displayed; the user can either select one of these or move on to view the next set of 10 patients. You worry that this system gives an unfair advantage to patients with last names that start with A, B, C, and D. Should you write and install the program? If you do not, many transplant opportunities will be missed while the hospital searches for another programmer who will write the program.
2. Suppose 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. 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.) Your friend could look up police records at the courthouse, but it takes many hours per week. As a convenience, should you provide your friend with outcomes of any arrest records of job applicants?