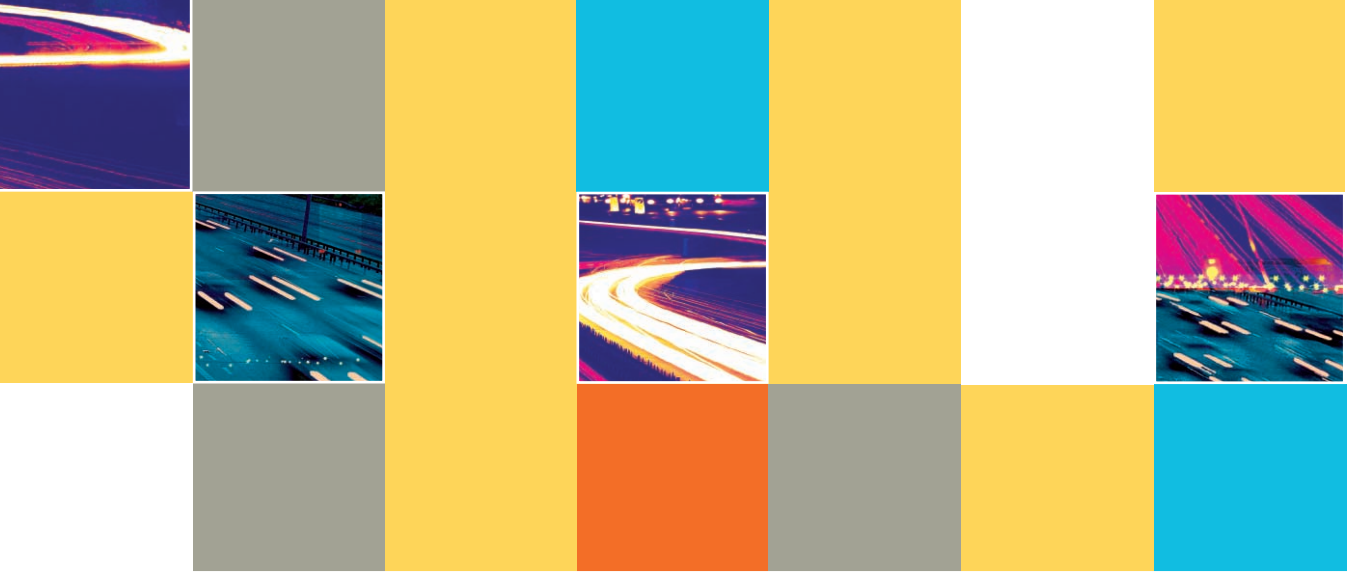


7 CONTROL BREAKS

After studying Chapter 7, you should be able to:

- Understand control break logic
- Perform single-level control breaks
- Use control data within a heading in a control break module
- Use control data within a footer in a control break module
- Perform control breaks with totals
- Perform multiple-level control breaks
- Perform page breaks

UNDERSTANDING CONTROL BREAK LOGIC

A **control break** is a temporary detour in the logic of a program. In particular, programmers refer to a program as a **control break program** when a change in the value of a variable initiates special actions or causes special or unusual processing to occur. You usually write control break programs to organize output for programs that handle data records that are organized logically in groups based on the value in a field. As you read records, you examine the same field in each record, and when you encounter a record that contains a different value from the ones that preceded it, you perform a special action. If you have ever read a report that lists items in groups, with each group followed by a subtotal, then you have read a type of **control break report**. For example, you might generate a report that lists all company clients in order by state of residence, with a count of clients after each state's client list. See Figure 7-1 for an example of a report that breaks after each change in state.

FIGURE 7-1: A CONTROL BREAK REPORT WITH TOTALS AFTER EACH STATE

```
Company Clients by State of Residence

Name                    City                    State

Albertson               Birmingham              Alabama
Davis                   Birmingham              Alabama
Lawrence                Montgomery              Alabama
                                                Count for Alabama    3

Smith                   Anchorage               Alaska
Young                   Anchorage               Alaska
Davis                   Fairbanks               Alaska
Mitchell                Juneau                  Alaska
Zimmer                  Juneau                  Alaska
                                                Count for Alaska     5

Edwards                 Phoenix                 Arizona
                                                Count for Arizona    1
```

Some other examples of control break reports produced by control break programs include:

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

Each of these reports shares two traits:

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

To generate a control break report, your input records must be organized in sequential order based on the field that will cause the breaks. In other words, if you are going to write a program that produces a report that lists customers by state, like the one in Figure 7-1, then the records must be grouped by state before you begin processing. Frequently, grouping by state will mean placing the records in alphabetical order by state, although they could just as easily be placed in order by population, governor's last name, or any other factor as long as all of one state's records are together. As you grow more proficient in programming logic, you will learn techniques for writing programs that sort records before you proceed with creating a program that contains control break logic. Programs that **sort** records take records that are not in order and rearrange them to be in order, according to the data in some field. For now, assume that a sorting program has already been used to presort your records before you begin the part of a program that determines control breaks.

TIP To use control break logic, either the records must arrive already in order in the input file or you must sort the records yourself. You will learn techniques for processing unsorted records in Chapter 8. In Chapter 9, you will learn to sort records. It is easier to work with sorted records than unsorted ones, so you are learning the easier techniques first.

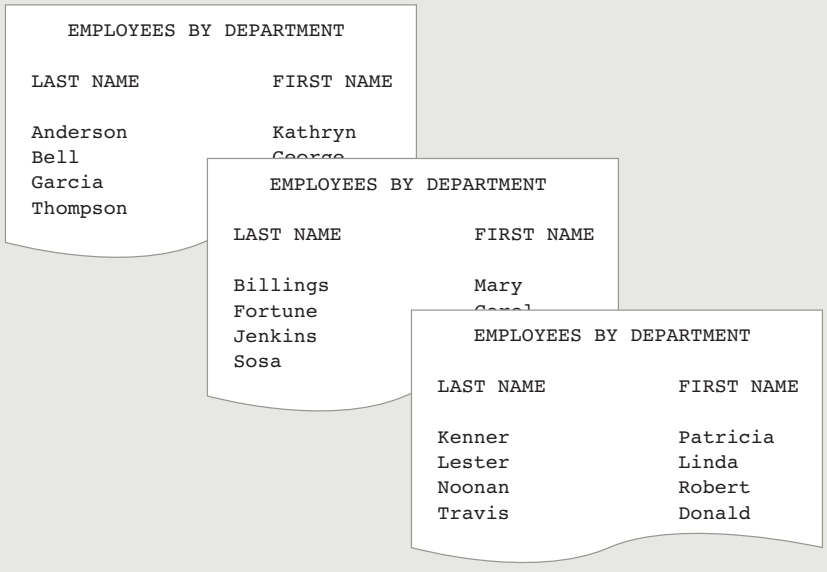
PERFORMING A SINGLE-LEVEL CONTROL BREAK TO START A NEW PAGE

Suppose you want to print a list of employees, advancing to a new page for each department. Figure 7-2 shows the input file description, from which you can see that the employee department is a numeric field, and that the file has been presorted so that the records will arrive in a program in department-number order. Figure 7-3 shows a sample report with the desired output—a simple list of employee names, with one department per page.

FIGURE 7-2: EMPLOYEE FILE DESCRIPTION

```
File name: EMPSBYDEPT
Sorted by: Department
FIELD DESCRIPTION        DATA TYPE     COMMENTS
Department               Numeric       0 decimals
Last Name                Character     15 characters
First Name               Character     15 characters
```

FIGURE 7-3: SAMPLE CONTROL BREAK REPORT LISTING EMPLOYEES, WITH EACH DEPARTMENT ON A NEW PAGE



TIP In the report in Figure 7-3, each new page contains employees from a new department. Later in this chapter, department numbers will be added to the headings, making this point clearer to those who read the report.

The basic logic of the program works like this: Each time you read an employee record from the input file, you will determine whether the employee belongs to the same department as the previous employee. If so, you simply print the employee record and read another record, without any special processing. If there are 20 employees in a department, these steps are repeated 20 times in a row—read an employee record and print the employee record. However, eventually you will read an employee record that does not belong to the same department. At that point, before you print the employee record from the new department, you must print headings at the top of a new page. Then, you can proceed to read and print employee records that belong to the new department, and you continue to do so until the next time you encounter an employee in a different department. This type of program contains a **single-level control break**, a break in the logic of the program (pausing or detouring to print new headings) that is based on the value of a single variable (the department number).

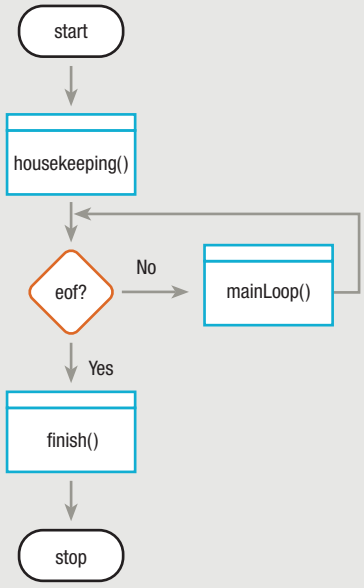
However, there is a slight problem you must solve before you can determine whether a new input record contains the same department number as the previous input record. When you read a record from an input file, you copy the data from storage locations (for example, from a disk) to temporary computer memory locations. After they are read, the data items that represent department, last name, and first name occupy specific physical locations in computer memory. For each new record that is read from storage, new data must occupy the same positions in memory as the previous record occupied, and the previous set of data is lost. For example, if you read a record containing data for Donald Travis in Department 1, when you read the next record for Mary Billings in Department 2, “Mary” replaces “Donald”, “Billings” replaces “Travis”, and 2 replaces 1. After you read a new record into memory, there is no way to look back at the

previous record to determine whether that record had a different department number. The previous record's data has been replaced in memory by the new record's data.

The technique you must use to "remember" the old department number is to create a special variable, called a **control break field**, to hold the previous department number. With a control break field, every time you read a record and print it, you also can save the crucial part of the record that will signal the change or control the program break. In this case, you want to store the department number in this specially created variable. Comparing the new and old department-number values will determine when it is time to print headings at the top of a new page.

The mainline logic for the Employees by Department report is the same as the mainline logic for all the other programs you've analyzed so far. It performs a `housekeeping()` module, after which an `eof` question controls execution of a `mainLoop()` module. At `eof`, a `finish()` module executes. See Figure 7-4.

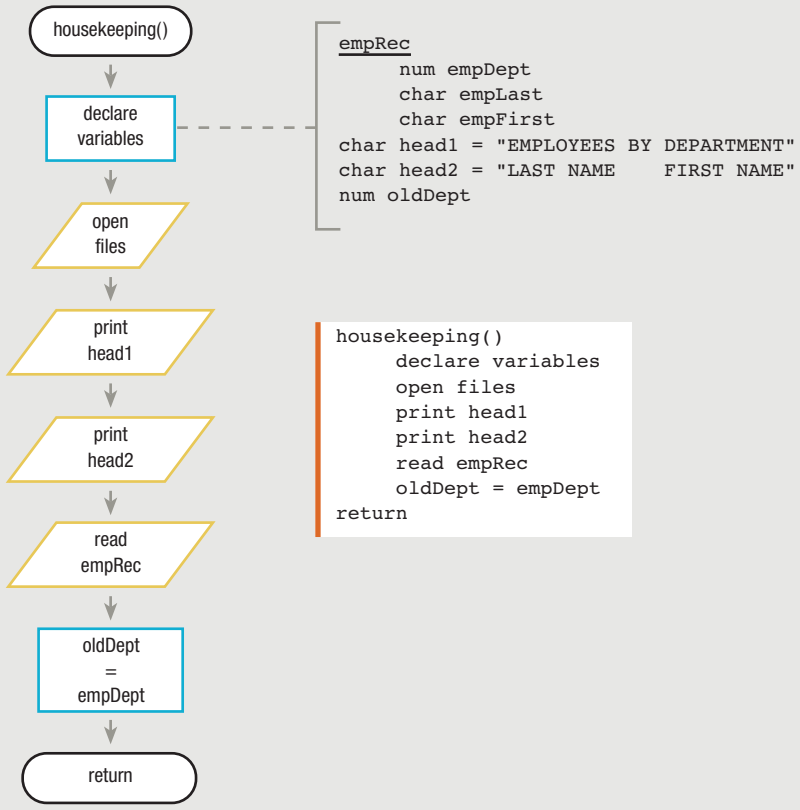
FIGURE 7-4: MAINLINE LOGIC FOR EMPLOYEES BY DEPARTMENT REPORT PROGRAM




```
start
     perform housekeeping()
     while not eof
          perform mainLoop()
     endwhile
     perform finish()
stop
```

The `housekeeping()` module for this program begins like others you have seen. You declare variables as shown in Figure 7-5, including those you will use for the input data: `empDept`, `empLast`, and `empFirst`. You can also declare variables to hold the headings, and an additional variable that is named `oldDept` in this example. The purpose of `oldDept` is to serve as the control break field. Every time you read a record from a new department, you can save its department number in `oldDept` before you read the next record. The `oldDept` field provides you with a comparison for each new department so you can determine whether there has been a change in value.

FIGURE 7-5: THE `housekeeping()` MODULE FOR EMPLOYEES BY DEPARTMENT REPORT PROGRAM



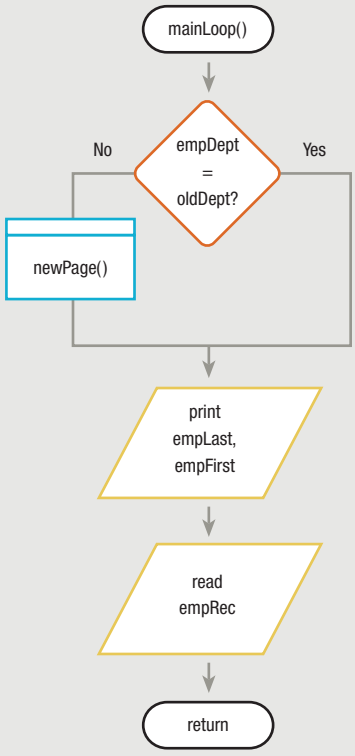
Note that it would be incorrect to initialize `oldDept` to the value of `empDept` when you declare `oldDept` in the `housekeeping()` module. When you declare variables at the beginning of the `housekeeping()` module, you have not yet read the first record; therefore, `empDept` does not yet have any usable value. You use the value of the first `empDept` variable at the end of the module, only after you read the first input record.

In the `housekeeping()` module, after declaring variables, you also open files, print headings, and read the first input record. Before you leave the `housekeeping()` module, you can set the `oldDept` variable to equal the `empDept` value in the first input record. You will write the `mainLoop()` module of the program to check for any change in department number; that's the signal to print headings at the top of a new page. Because you just printed headings and read the first record, you do not want to print headings again for this first record, so you want to ensure that `empDept` and `oldDept` are equal when you enter `mainLoop()`.

TIP As an alternative to the `housekeeping()` logic shown here, you can remove printing headings from the `housekeeping()` module and set `oldDept` to any impossible value—for example, –1. Then, in `mainLoop()`, the first record will force the control break, and the headings will print in the `newPage()` control break routine.

The first task within the `mainLoop()` module is to check whether `empDept` holds the same value as `oldDept`. For the first record, on the first pass through `mainLoop()`, the values are equal; you set them to be equal in the `housekeeping()` module. Therefore, you proceed without performing the `newPage()` module, printing the first employee's record and reading a second record. At the end of the `mainLoop()` module, shown in Figure 7-6, the logical flow returns to the mainline logic, shown in Figure 7-4. If it is not `eof`, the flow travels back into the `mainLoop()` module. There, you compare the second record's `empDept` to `oldDept`. If the second record holds an employee from the same department as the first employee, then you simply print that second employee's record and read a third record into memory. As long as each new record holds the same `empDept` value, you continue reading and printing, never pausing to perform the `newPage()` module.

FIGURE 7-6: THE `mainLoop()` MODULE FOR EMPLOYEES BY DEPARTMENT REPORT PROGRAM




```
mainLoop()
     if empDept not = oldDept then
          perform newPage()
     endif
     print empLast, empFirst
     read empRec
return
```

TIP

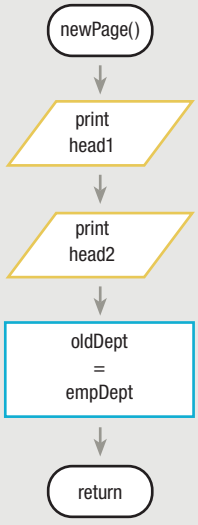
In the flowchart in Figure 7-6, you could change the decision to `empDept not = oldDept`. Then, the Yes branch of the decision structure would perform the `newPage()` module, and the No branch would be null. This format would more closely resemble the pseudocode in Figure 7-6, but the logic would be identical to the version shown here. In other words, you perform `newPage()` when `empDept = oldDept` is false or when `empDept not = oldDept` is true.

Eventually, you will read in an employee whose `empDept` is not the same as `oldDept`. That's when the control break routine, `newPage()`, executes. The `newPage()` module must perform two tasks:

- It must print headings at the top of a new page.
- It must update the control break field.

Figure 7-7 shows the `newPage()` module.

FIGURE 7-7: THE `newPage()` MODULE FOR EMPLOYEES BY DEPARTMENT REPORT PROGRAM




```
newPage()
      print head1
      print head2
      oldDept = empDept
return
```

TIP Notice that the steps in the `newPage()` module mimic steps in the `housekeeping()` module. You take advantage of this coincidence later in this chapter.

TIP In Chapter 4, you learned that specific programming languages each provide you with a means to physically advance printer paper to the top of a page. Usually, you insert a language-specific code just before the first character in the first heading that will appear on a page. For this book, if a sample report or print chart shows a heading printing at the top of the page, then you can assume that printing the heading causes the paper in the printer to advance to the top of a new page. The appropriate language-specific codes can be added when you code the program.

When you read an employee record in which `empDept` is not the same as `oldDept`, you cause a break in the normal flow of the program. The new employee record must "wait" while headings print and the control break field `oldDept` acquires a new value. After the `oldDept` field has been updated, and before the `mainLoop()` module ends, the waiting employee record prints on the new page. When you read the *next* employee record (and it is not `eof`), the `mainLoop()` module is reentered and the next employee's `empDept` field is compared to the updated `oldDept` field. If the new employee works in the same department as the one just preceding, then normal processing continues with the print-and-read statements.

The `newPage()` module in the employee report program performs two tasks required in all control break modules:

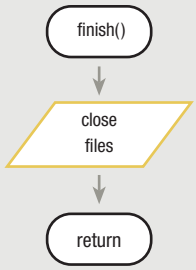
- It performs any necessary processing for the new group—in this case, it prints headings.
- It updates the control break field—in this case, the `oldDept` field.

TIP

As an alternative to updating the control break field within the control break routine, you could set `oldDept` equal to `empDept` just before you read each record. However, if there are 200 employees in Department 55, then you set `oldDept` to the same value 200 times. It's more efficient to set `oldDept` to a different value only when there is a change in the value of the department.

The `finish()` module for the Employees by Department report program requires only that you close the files. See Figure 7-8.

FIGURE 7-8: THE `finish()` MODULE FOR EMPLOYEES BY DEPARTMENT REPORT PROGRAM

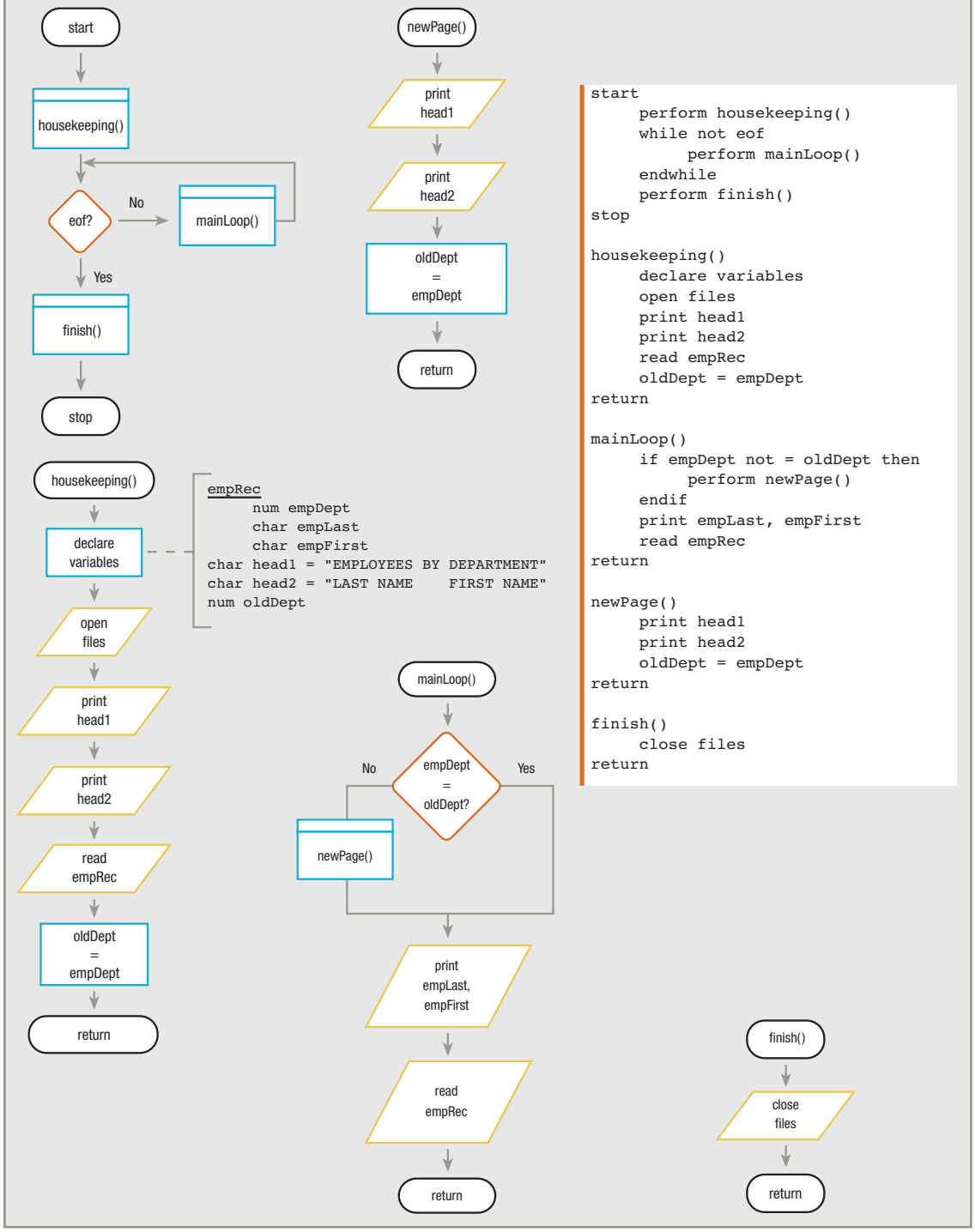



```
finish()
     close files
return
```

Notice that in the control break program described in Figures 7-4 through 7-8, the department numbers of employees in the input file do not have to follow each other incrementally. That is, the departments might be 1, 2, 3, and so on, but they also might be 1, 4, 12, 35, and so on. A control break occurs when there is a change in the control break field; the change does not necessarily have to be a numeric change of 1.

Figure 7-9 shows the entire Employees by Department control break program.

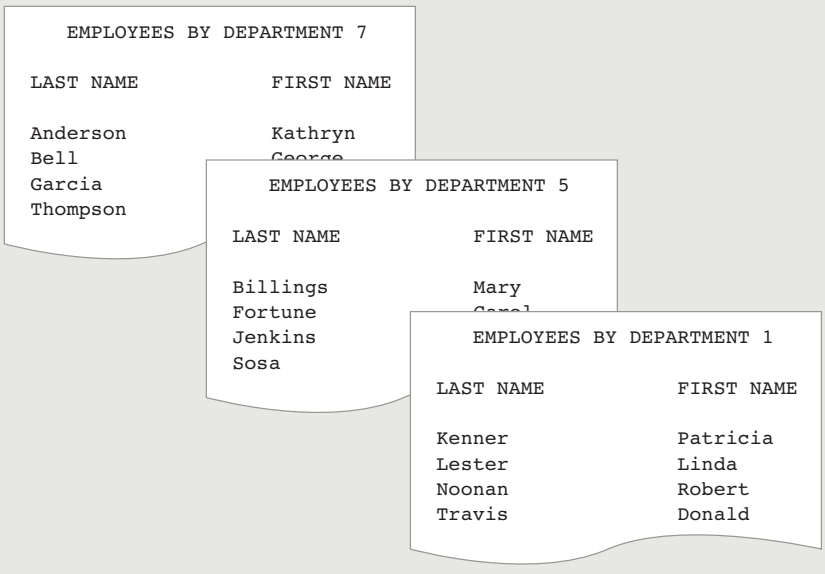
FIGURE 7-9: THE EMPLOYEES BY DEPARTMENT CONTROL BREAK PROGRAM



USING CONTROL DATA WITHIN A HEADING IN A CONTROL BREAK MODULE

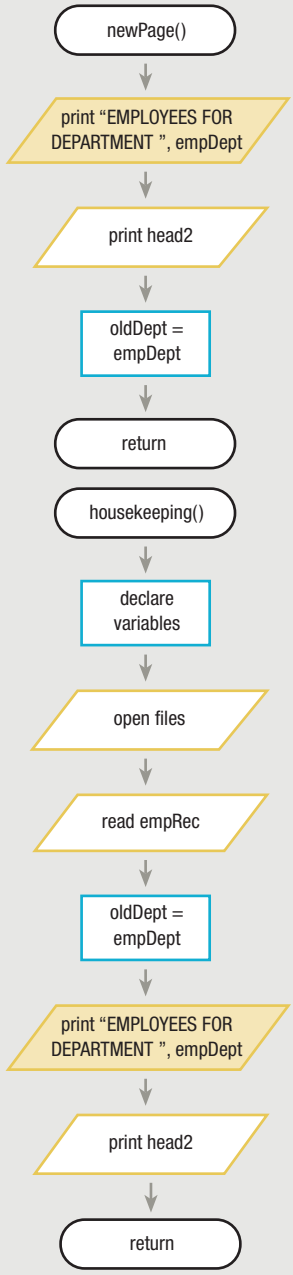
In the Employees by Department report program example in Figure 7-9, the control break module printed constant headings at the top of each new page; in other words, each page heading was the same. However, sometimes you need to use control data within the heading. For example, consider the sample report shown in Figure 7-10.

FIGURE 7-10: SAMPLE REPORT FOR EMPLOYEES BY DEPARTMENT IN WHICH DEPARTMENT NUMBERS APPEAR IN THE HEADING



The difference between Figure 7-3 and Figure 7-10 lies in the heading. Figure 7-10 shows variable data in the heading—a different department number prints at the top of each page of employees. To create this kind of program, you must make two changes in the existing program. First, you modify the `newPage()` module, as shown in Figure 7-11. Instead of printing a fixed heading on each new page, you print a heading that contains two parts: a constant beginning ("EMPLOYEES FOR DEPARTMENT") and a variable ending (the department number for the employees who appear on the page). Notice that you use the `empDept` number that belongs to the employee record that is waiting to be printed while this control break module executes. Additionally, you must modify the `housekeeping()` module to ensure that the first heading on the report prints correctly. As Figure 7-11 shows, you must modify the `housekeeping()` module from Figure 7-5 so that you read the first `empRec` prior to printing the headings. The reason is that you must know the first employee's department number before you can print the heading for the top of the first page.

FIGURE 7-11: MODIFIED `newPage()` AND `housekeeping()` MODULES FOR EMPLOYEES BY DEPARTMENT REPORT THAT DISPLAYS THE DEPARTMENT NUMBER IN THE HEADING




```
newPage()
     print "EMPLOYEES FOR
          DEPARTMENT ", empDept
     print head2
     oldDept = empDept
return
```

```
housekeeping()
     declare variables
     open files
     read empRec
     oldDept = empDept
     print "EMPLOYEES FOR
          DEPARTMENT ", empDept
     print head2
return
```

USING CONTROL DATA WITHIN A FOOTER IN A CONTROL BREAK MODULE

In the previous section, you learned how to use control break data in a heading. Figure 7-12 shows a different report format. For this report, the department number prints *following* the employee list for the department. A message that prints at the end of a page or other section of a report is called a **footer**. Headings usually require information about the *next* record; footers usually require information about the *previous* record.

FIGURE 7-12: SAMPLE REPORT FOR EMPLOYEES BY DEPARTMENT IN WHICH DEPARTMENT NUMBERS APPEAR IN THE FOOTER

```
    EMPLOYEES BY DEPARTMENT

LAST NAME            FIRST NAME

Anderson             Kathryn
Bell                 George
Garcia               Maria
Thompson             Olivia

END OF DEPARTMENT 7
```

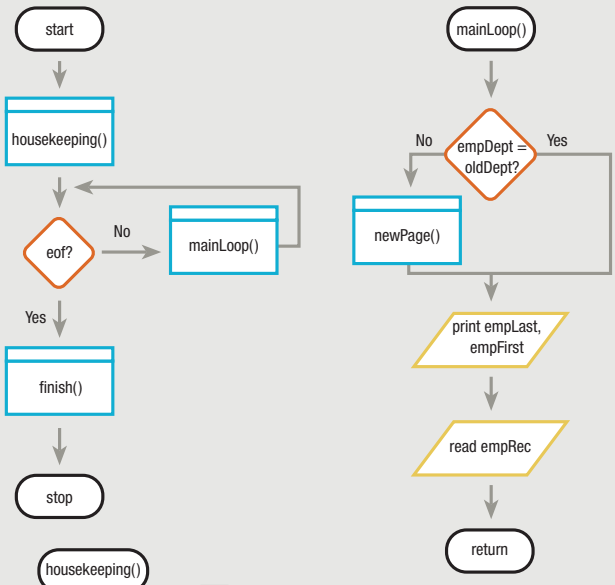
```
    EMPLOYEES BY DEPARTMENT

LAST NAME            FIRST NAME

Billings             Mary
Fortune              Carol
Jenkins              Justin
Sosa                 Charles

END OF DEPARTMENT 5
```

```
    EMPLOYEES BY DEPARTMENT

LAST NAME            FIRST NAME

Kenner               Patricia
Lester               Linda
Noonan               Robert
Travis               Donald

END OF DEPARTMENT 1
```

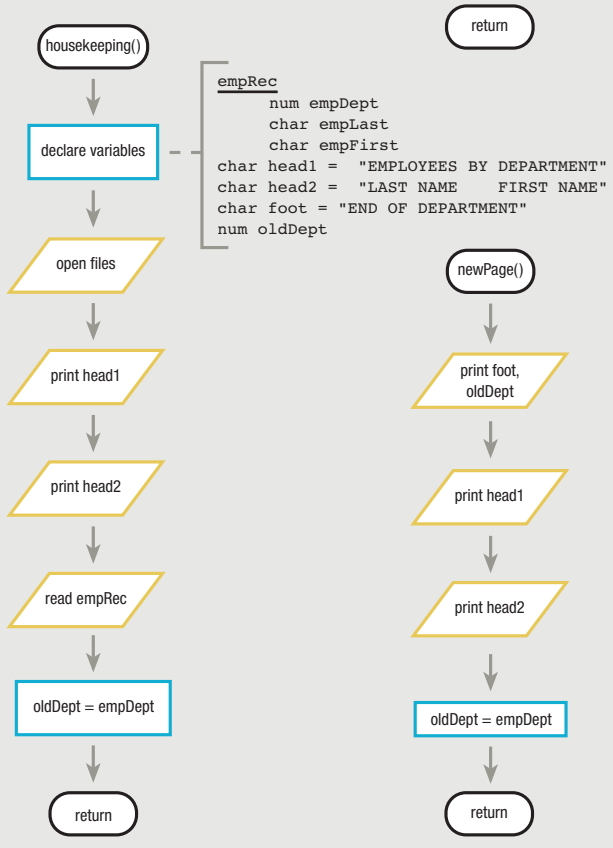
Figure 7-13 shows a program that prints a list of employees by department, including a footer that displays the department number at the end of each department's list. When you write a program that produces the report like the one shown in Figure 7-12, you continuously read records with `empLast`, `empFirst`, and `empDept` fields. Each time `empDept` does not equal `oldDept`, it means that you have reached a department break and that you should perform the `newPage()` module. The `newPage()` module has three tasks:

- It must print the footer for the previous department at the bottom of the employee list.
- It must print headings at the top of a new page.
- It must update the control break field.

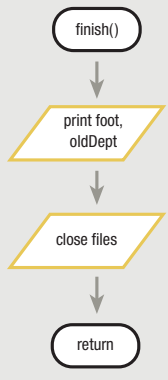
FIGURE 7-13: PROGRAM THAT LISTS EMPLOYEES BY DEPARTMENT, INCLUDING DEPARTMENT NUMBER IN THE FOOTER




```
start
     perform housekeeping()
     while not eof
          perform mainLoop()
     endwhile
     perform finish()
stop



housekeeping()
     declare variables
     open files
     print head1
     print head2
     read empRec
     oldDept = empDept
return

mainLoop()
     if empDept not = oldDept then
          perform newPage()
     endif
     print empLast, empFirst
     read empRec
return

newPage()
     print foot, oldDept
     print head1
     print head2
     oldDept = empDept
return

finish()
     print foot, oldDept
     close files
return
```




```
empRec
     num empDept
     char empLast
     char empFirst
char head1 =  "EMPLOYEES BY DEPARTMENT"
char head2 =  "LAST NAME    FIRST NAME"
char foot = "END OF DEPARTMENT"
num oldDept
```

When the `newPage()` module prints the footer at the bottom of the old page, you must use the `oldDept` number. For example, assume you have printed several employees from Department 12. When you read a record with an employee from Department 13 (or any other department), the first thing you must do is print "END OF DEPARTMENT 12". You print the correct department number by accessing the value of `oldDept`, not `empDept`. Then, you can print the other headings at the top of a new page and update `oldDept` to the current `empDept`, which in this example is 13.

The `newPage()` module in Figure 7-13 performs three tasks required in all control break routines: it processes the previous group, processes the new group, and updates the control break field.

When you printed the department number in the header in the example in the previous section, you needed a special step in the `housekeeping()` module. When you print the department number in the footer, the `finish()` module requires an extra step. Imagine that the last five records in the input file include two employees from Department 78, Amy and Bill, and three employees from Department 85, Carol, Don, and Ellen. The logical flow proceeds as follows:

1. After the first Department 78 employee (Amy) prints, you read the second Department 78 employee (Bill).
2. At the top of the `mainLoop()` module, Bill's department is compared to `oldDept`. The departments are the same, so the second Department 78 employee (Bill) is printed. Then, you read the first Department 85 employee (Carol).
3. At the top of `mainLoop()`, Carol's `empDept` and `oldDept` are different, so you perform the `newPage()` module while Carol's record waits in memory.
4. In the `newPage()` module, you print "END OF DEPARTMENT 78". Then, you print headings at the top of the next page. Finally, you set `oldDept` to 85, and then return to `mainLoop()`.
5. Back in `mainLoop()`, you print a line of data for the first Department 85 employee (Carol), whose record waited while `newPage()` executed. Then, you read the record for the second Department 85 employee (Don).
6. At the top of `mainLoop()`, you compare Don's department number to `oldDept`. The numbers are the same, so you print Don's employee data and read in the last Department 85 employee (Ellen).
7. At the top of `mainLoop()`, you determine that Ellen has the same department number, so you print Ellen's data and attempt to read from the input file, where you encounter `eof`.
8. The `eof` decision in the mainline logic sends you to the `finish()` module.

You have printed the last Department 85 employee (Ellen), but the department footer for Department 85 has not printed. That's because every time you attempt to read an input record, you don't know whether there will be more records. The mainline logic checks for the `eof` condition, but if it determines that it is `eof`, the logic does not flow back into the `mainLoop()` module, where the `newPage()` module can execute.

To print the footer for the last department, you must print a footer one last time within the `finish()` routine. The `finish()` module that is part of the complete program in Figure 7-13 illustrates this point. Taking this action is similar to printing the first heading in the `housekeeping()` module. The very first heading prints separately from all the others at the beginning; the very last footer must print separately from all the others at the end.

PERFORMING CONTROL BREAKS WITH TOTALS

Suppose you run a bookstore, and one of the files you maintain is called BOOKFILE, which has one record for every book title that you carry. Each record has fields such as `bookTitle`, `bookAuthor`, `bookCategory` (fiction, reference, self-help, and so on), `bookPublisher`, and `bookPrice`, as shown in the file description in Figure 7-14.

FIGURE 7-14: BOOKFILE FILE DESCRIPTION

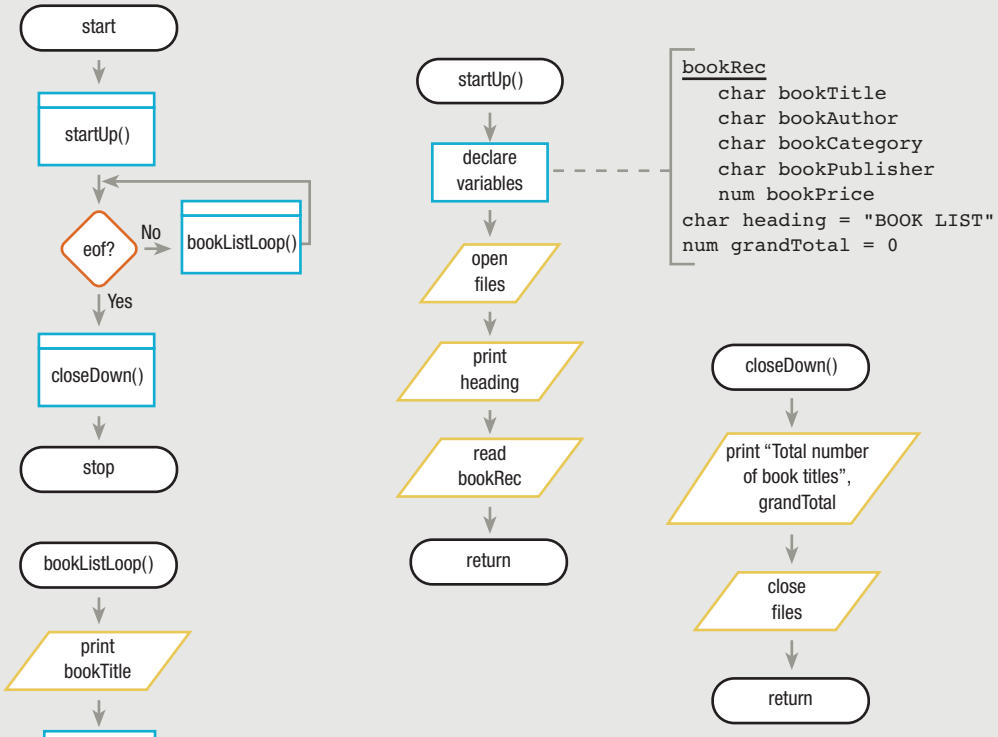
```
File name: BOOKFILE
Sorted by: Category
FIELD DESCRIPTION       DATA TYPE          COMMENTS
Title                   Character          30 characters
Author                  Character          15 characters
Category                Character          15 characters
Publisher               Character          15 characters
Price                   Numeric            2 decimals
```

Suppose you want to print a list of all the books that your store carries, with a total number of books at the bottom of the list, as shown in the sample report in Figure 7-15. You can use the logic shown in Figure 7-16. In the main loop module, named `bookListLoop()`, you print a book title, add 1 to `grandTotal`, and read the next record. At the end of the program, in the `closeDown()` module, you print `grandTotal` before you close the files. You can't print `grandTotal` any earlier in the program because the `grandTotal` value isn't complete until the last record has been read.

FIGURE 7-15: SAMPLE BOOK LIST REPORT

```
BOOK LIST

A Brief History of Time
The Scarlet Letter
Math Magic
She's Come Undone
The Joy of Cooking
Walden
A Bridge Too Far
The Time Traveler's Wife
The DaVinci Code

Programming Logic and Design
Forever Amber

Total number of book titles  512
```

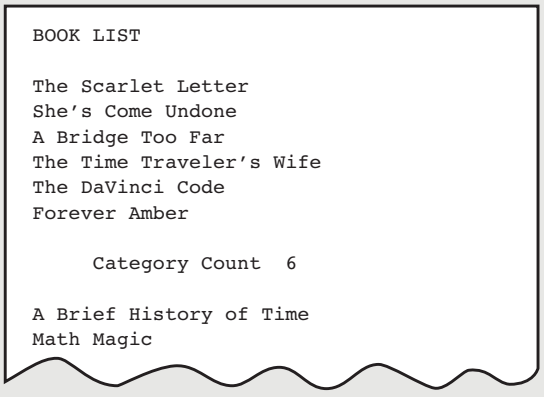
FIGURE 7-16: FLOWCHART AND PSEUDOCODE FOR BOOKSTORE PROGRAM




```
start
   perform startUp()
   while not eof
      perform bookListLoop()
   endwhile
   perform closeDown()
stop

startUp()
   declare variables
   open files
   print heading
   read bookRec
return

bookListLoop()
   print bookTitle
   grandTotal = grandTotal + 1
   read bookRec
return

closeDown()
   print "Total number of book titles", grandTotal
   close files
return
```

The logic of the book list report program is pretty straightforward. Suppose, however, that you decide you want a count for each category of book rather than just one grand total. For example, if all the book records contain a category that is either fiction, reference, or self-help, then the book records might be sorted in alphabetical order by category, and the output would consist of a list of all fiction books first, followed by a count; then all reference books, followed by a count; and finally all self-help books, followed by a count. The report is a control break report, and the control break field is `bookCategory`. See Figure 7-17 for a sample report.

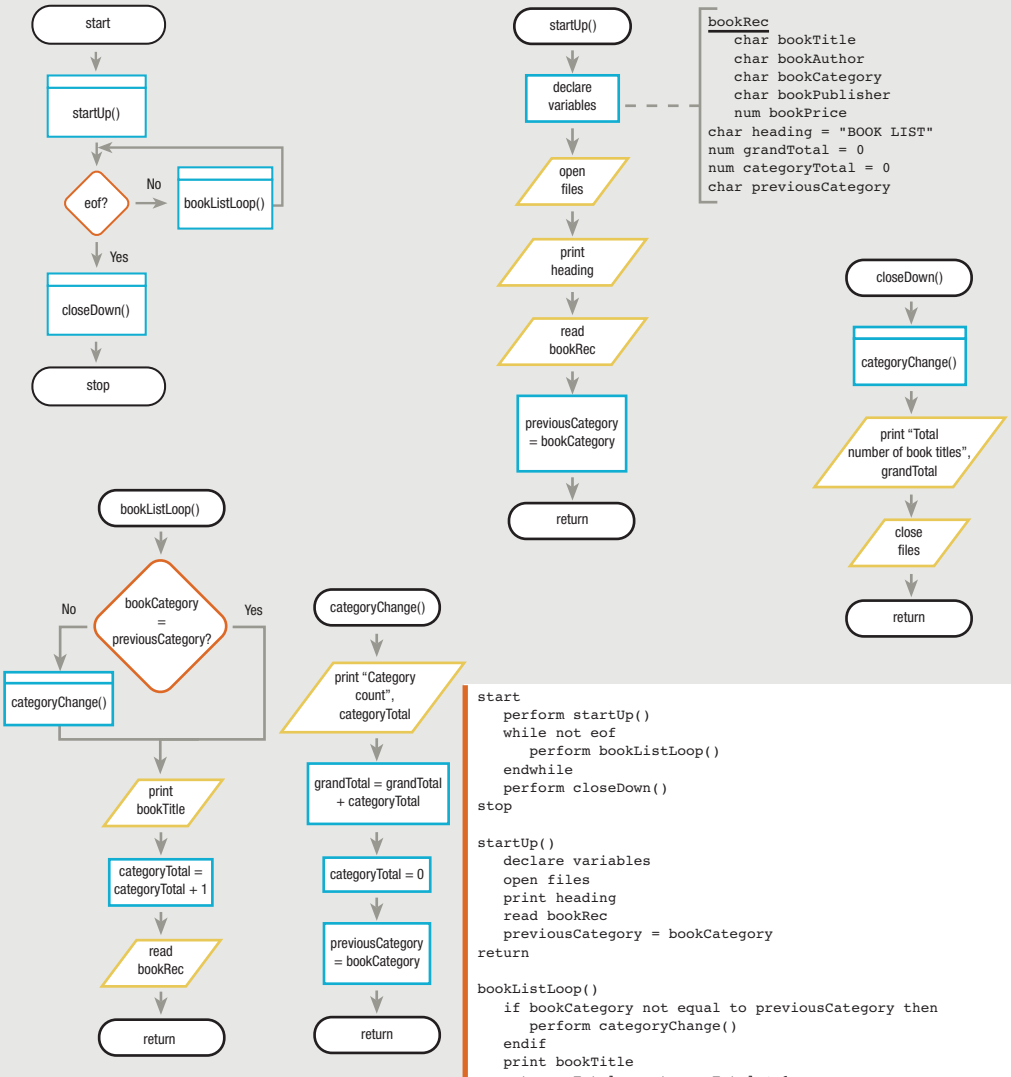
FIGURE 7-17: SAMPLE REPORT LISTING BOOKS BY CATEGORY WITH CATEGORY COUNTS



```
BOOK LIST

The Scarlet Letter
She's Come Undone
A Bridge Too Far
The Time Traveler's Wife
The DaVinci Code
Forever Amber

     Category Count  6

A Brief History of Time
Math Magic
```

To produce the report with subtotals by category, you must declare two new variables: `previousCategory` and `categoryTotal`. Every time you read a book record, you compare `bookCategory` to `previousCategory`; when there is a category change, you print the count of books for the previous category. The `categoryTotal` variable holds that count. See Figure 7-18.

FIGURE 7-18: FLOWCHART AND PSEUDOCODE FOR BOOKSTORE PROGRAM CONTAINING A COUNT AFTER EACH BOOK CATEGORY GROUP




```
start
   perform startUp()
   while not eof
      perform bookListLoop()
   endwhile
   perform closeDown()
stop

startUp()
   declare variables
   open files
   print heading
   read bookRec
   previousCategory = bookCategory
return

bookListLoop()
   if bookCategory not equal to previousCategory then
      perform categoryChange()
   endif
   print bookTitle
   categoryTotal = categoryTotal + 1
   read bookRec
return

categoryChange()
   print "Category count", categoryTotal
   grandTotal = grandTotal + categoryTotal
   categoryTotal = 0
   previousCategory = bookCategory
return

closeDown()
   perform categoryChange()
   print "Total number of book titles", grandTotal
   close files
return
```

TIP When you draw a flowchart, it usually is clearer to ask questions positively, as in "`bookCategory = previousCategory?`", and draw appropriate actions on the Yes or No side of a decision. In pseudocode, when action occurs only on the No side of a decision, it is usually clearer to ask negatively, as in "`bookCategory not equal to previousCategory?`" Figure 7-18 uses these tactics.

When you read the first record from the input file in the `startUp()` module of the program in Figure 7-18, you save the value of `bookCategory` in the `previousCategory` variable. Every time a record enters the `bookListLoop()` module, the program checks to see if the current record represents a new category of work, by comparing `bookCategory` to `previousCategory`. When you process the first record, the categories match, so the book title prints, the `categoryTotal` increases by 1, and you read the next record. If this next record's `bookCategory` value matches the `previousCategory` value, processing continues as usual: printing a line and adding 1 to `categoryTotal`.

At some point, `bookCategory` for an input record does not match `previousCategory`. At that point, you perform the `categoryChange()` module. Within the `categoryChange()` module, you print the count of the previous category of books. Then, you add `categoryTotal` to `grandTotal`. Adding a total to a higher-level total is called **rolling up the totals**.

You could write `bookListLoop()` so that as you process each book, you add 1 to `categoryTotal` and add 1 to `grandTotal`. Then, there would be no need to roll totals up in the `categoryChange()` module. If there are 120 fiction books, you add 1 to `categoryTotal` 120 times; you also would add 1 to `grandTotal` 120 times. This technique would yield correct results, but you can eliminate executing 119 addition instructions by waiting until you have accumulated all 120 category counts before adding the total figure to `grandTotal`.

This control break report containing totals performs the five tasks required in all control break routines that include totals:

- It performs any necessary processing for the previous group—in this case, it prints `categoryTotal`.
- It rolls up the current-level totals to the next higher level—in this case, it adds `categoryTotal` to `grandTotal`.
- It resets the current level's totals to zero—in this case, `categoryTotal` is set to zero.
- It performs any necessary processing for the new group—in this case, there is none.
- It updates the control break field—in this case, `previousCategory`.

The `closeDown()` routine for this type of program is more complicated than it might first appear. It seems as though you should print `grandTotal`, close the files, and return to the mainline logic. However, when you read the last record, the mainline `eof` decision sends the logical flow to the `closeDown()` routine. You have not printed the last `categoryTotal`, nor have you added the count for the last category to `grandTotal`. You must take care of both these tasks before printing `grandTotal`. You can perform these two tasks as separate steps in `closeDown()`, but it is often simplest just to remember to perform the control break routine `categoryChange()` one last time. The `categoryChange()` module already executes after every previous category completes—that is, every time you encounter a new category during the execution of the program. You also

can execute this module after the final category completes, at the end of the file. Encountering the end of the file is really just another form of break; it signals that the last category has finally completed. The `categoryChange()` module prints the category total and rolls the totals up to the `grandTotal` level.

TIP

When you call the `categoryChange()` module from within `closeDown()`, it performs a few tasks you don't need, such as setting the value of `previousCategory`. You have to weigh the convenience of calling the already-written `categoryChange()` module, and executing a few unneeded statements, against taking the time to write a new module that would execute only the statements that are absolutely necessary.

It is very important to note that this control break program works whether there are three categories of books or 300. Note further that it does not matter what the categories of books are. For example, the program never asks `bookCategory = "fiction"?`. Instead, the control of the program breaks when the category field *changes*, and it is in no way dependent on *what* that change is.

PERFORMING MULTIPLE-LEVEL CONTROL BREAKS

Let's say your bookstore from the last example is so successful that you have a chain of them across the country. Every time a sale is made, you create a record with the fields `bookTitle`, `bookPrice`, `bookCity`, and `bookState`. You want a report that prints a summary of books sold in each city and each state, similar to the one shown in Figure 7-19. A report such as this one, which does not include any information about individual records, but instead includes only group totals, is a **summary report**.

This program contains a **multiple-level control break**—that is, the normal flow of control (reading records and counting book sales) breaks away to print totals in response to more than just one change in condition. In this report, a control break occurs in response to either (or both) of two conditions: when the value of the `bookCity` variable changes, as well as when the value of the `bookState` variable changes.

Just as the file you use to create a single-level control break report must be presorted, so must the input file you use to create a multiple-level control break report. The input file that you use for the book sales report must be sorted by `bookCity` *within* `bookState`. That is, all of one state's records—for example, all records from IA—come first; then all of the records from another state, such as IL, follow. Within any one state, all of one city's records come first; then all of the next city's records follow. For example, the input file that produces the report shown in Figure 7-19 contains 200 records for book sales in Ames, IA, followed by 814 records for book sales in Des Moines, IA. The basic processing entails reading a book sale record, adding 1 to a counter, and reading the next book sale record. At the end of any city's records, you print a total for that city; at the end of a state's records, you print a total for that state.

FIGURE 7-19: SAMPLE RUN OF BOOK SALES BY CITY AND STATE REPORT

```
BOOK SALES BY CITY AND STATE

Ames                                 200
Des Moines                           814
Iowa City                            291
     Total for IA                   1305
Chicago                             1093
Crystal Lake                         564
McHenry                              213
Springfield                          365
     Total for IL                   2235
Springfield                          289
Worcester                            100
     Total for MA                    389
     Grand Total                    3929
```

The `housekeeping()` module of the Book Sales by City and State report program looks similar to the `housekeeping()` module in the previous control break program, in which there was a single control break for change in category of book. In each program, you declare variables, open files, and read the first record. This time, however, there are multiple fields to save and compare to the old fields. Here, you declare two special variables, `prevCity` and `prevState`, as shown in Figure 7-20. In addition, the Book Sales report shows three kinds of totals, so you declare three new variables that will serve as holding places for the totals in the Book Sales report: `cityCounter`, `stateCounter`, and `grandTotal`, which are all initialized to zero.

FIGURE 7-20: FLOWCHART AND PSEUDOCODE FOR `housekeeping()` MODULE IN BOOK SALES BY CITY AND STATE REPORT PROGRAM

This program prints both `bookState` and `bookCity` totals, so you need two control break modules, `cityBreak()` and `stateBreak()`. Every time there is a change in the `bookCity` field, the `cityBreak()` routine performs these standard control break tasks:

- It performs any necessary processing for the previous group—in this case, it prints totals for the previous city.
- It rolls up the current-level totals to the next higher level—in this case, it adds the city count to the state count.
- It resets the current level's totals to zero—in this case, it sets the city count to zero.
- It performs any necessary processing for the new group—in this case, there is none.
- It updates the control break field—in this case, it sets `prevCity` to `bookCity`.

Within the `stateBreak()` module, you must perform one new type of task, as well as the control break tasks you are familiar with. The new task is the first task: Within the `stateBreak()` module, you must first perform

`cityBreak()` automatically (because if there is a change in the state, there must also be a change in the city). The `stateBreak()` module does the following:

- It processes the lower-level break—in this case, `cityBreak()`.
- It performs any necessary processing for the previous group—in this case, it prints totals for the previous state.
- It rolls up the current-level totals to the next higher level—in this case, it adds the state count to the grand total.
- It resets the current level's totals to zero—in this case, it sets the state count to zero.
- It performs any necessary processing for the new group—in this case, there is none.
- It updates the control break field—in this case, it sets `prevState` to `bookState`.

The `mainLoop()` module of this multiple-level control break program checks for any change in two different variables: `bookCity` and `bookState`. When `bookCity` changes, a city total is printed, and when `bookState` changes, a state total is printed. As you can see from the sample report in Figure 7-19, all city totals for each state print before the state total for the same state, so it might seem logical to check for a change in `bookCity` before checking for a change in `bookState`. However, the opposite is true. For the totals to be correct, you must check for any `bookState` change first. You do so because when `bookCity` changes, `bookState` also *might* be changing, but when `bookState` changes, it means `bookCity` *must* be changing.

Consider the sample input records shown in Figure 7-21, which are sorted by `bookCity` within `bookState`. When you get to the point in the program where you read the first Illinois record (*The Scarlet Letter*), "Iowa City" is the value stored in the field `prevCity`, and "IA" is the value stored in `prevState`. Because the values in the `bookCity` and `bookState` variables in the new record are both different from the `prevCity` and `prevState` fields, both a city and state total will print. However, consider the problem when you read the first record for Springfield, MA (*Walden*). At this point in the program, `prevState` is IL, but `prevCity` is the same as the current `bookCity`; both contain Springfield. If you check for a change in `bookCity`, you won't find one at all, and no city total will print, even though Springfield, MA, is definitely a different city from Springfield, IL.

FIGURE 7-21: SAMPLE DATA FOR BOOK SALES BY CITY AND STATE REPORT

TITLE	PRICE	CITY	STATE
A Brief History of Time	20.00	Iowa City	IA
The Scarlet Letter	15.99	Chicago	IL
Math Magic	4.95	Chicago	IL
She's Come Undone	12.00	Springfield	IL
The Joy of Cooking	2.50	Springfield	IL
Walden	9.95	Springfield	MA
A Bridge Too Far	3.50	Springfield	MA

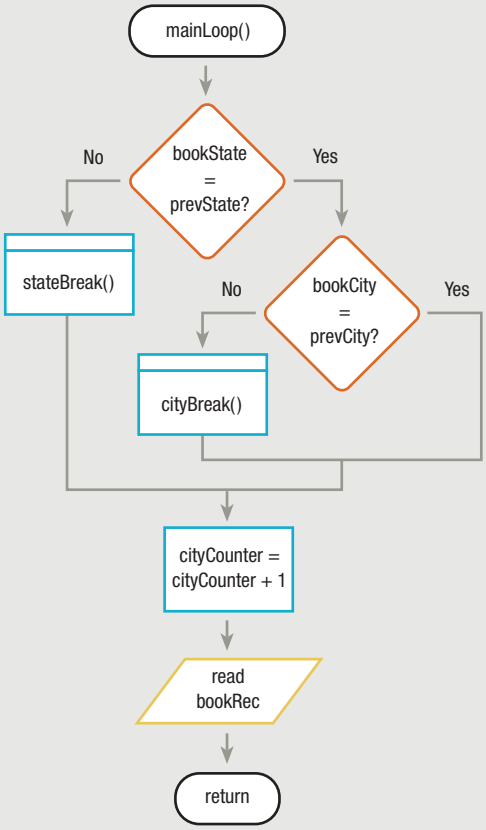
Cities in different states can have the same name; if two cities with the same name follow each other in your control break program and you have written it to check for a change in city name first, the program will not recognize that you are working with a

new city. Instead, you should always check for the major-level break first. If the records are sorted by `bookCity` within `bookState`, then a change in `bookState` causes a **major-level break**, and a change in `bookCity` causes a **minor-level break**. When the `bookState` value "MA" is not equal to the `prevState` value "IL", you force `cityBreak()`, printing a city total for Springfield, IL, before a state total for IL and before continuing with the Springfield, MA, record. You check for a change in `bookState` first, and if there is one, you perform `cityBreak()`. In other words, if there is a change in `bookState`, there is an implied change in `bookCity`, even if the cities happen to have the same name.

TIP If you needed totals to print by `bookCity` within a field defined as `bookCounty` within `bookState`, you could say you have minor-, intermediate-, and major-level breaks.

Figure 7-22 shows the `mainLoop()` module for the Book Sales by City and State report program. You check for a change in the `bookState` value. If there is no change, you check for a change in the `bookCity` value. If there is no change there either, you add 1 to the counter for the city and read the next record. When there is a change in the `bookCity` value, you print the city total and add the city total to the state total. When there is a change in the `bookState` value, you perform the break routine for the last city in the state, and then you print the state total and add it to the grand total.

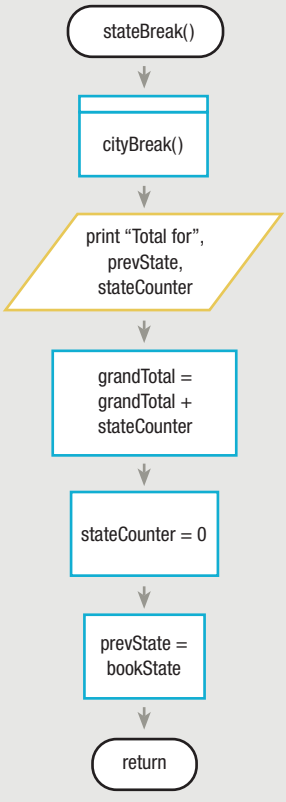
FIGURE 7-22: FLOWCHART AND PSEUDOCODE FOR `mainLoop()` FOR BOOK SALES BY CITY AND STATE REPORT PROGRAM




```
mainLoop()
     if bookState not equal prevState then
           perform stateBreak()
     else
           if bookCity not equal prevCity then
                 perform cityBreak()
           endif
     endif
     cityCounter = cityCounter + 1
     read bookRec
return
```

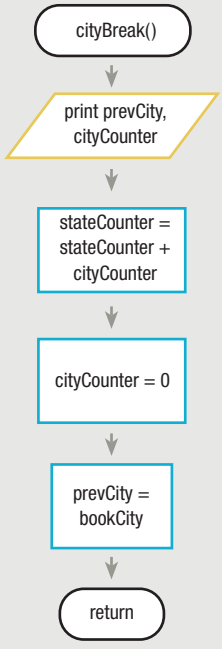
Figures 7-23 and 7-24 show the `stateBreak()` and `cityBreak()` modules. The two modules are very similar; the `stateBreak()` routine contains just one extra type of task. When there is a change in `bookState`, you perform `cityBreak()` automatically before you perform any of the other necessary steps to change states.

FIGURE 7-23: FLOWCHART AND PSEUDOCODE FOR `stateBreak()` MODULE




```
stateBreak()
      perform cityBreak()
      print "Total for", prevState, stateCounter
      grandTotal = grandTotal + stateCounter
      stateCounter = 0
      prevState = bookState
return
```

FIGURE 7-24: FLOWCHART AND PSEUDOCODE FOR `cityBreak()` MODULE



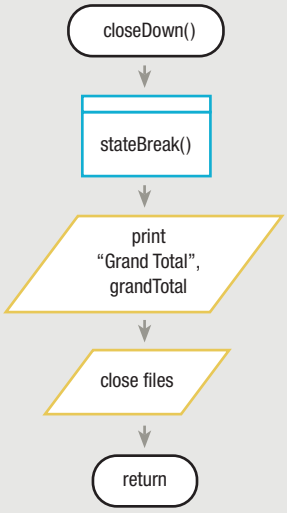

```
cityBreak()
      print prevCity, cityCounter
      stateCounter = stateCounter + cityCounter
      cityCounter = 0
      prevCity = bookCity
return
```

The sample report containing book sales by city and state shows that you print the grand total for all book sales, so within the `closeDown()` module, you must print the `grandTotal` variable. Before you can do so, however, you must perform both the `cityBreak()` and the `stateBreak()` modules one last time. You can accomplish this by performing `stateBreak()`, because the first step within `stateBreak()` is to perform `cityBreak()`.

Consider the sample data shown in Figure 7-21. While you continue to read records for books sold in Springfield, MA, you continue to add to the `cityCounter` for that city. At the moment you attempt to read one more record past the

end of the file, you do not know whether there will be more records; therefore, you have not yet printed either the `cityCounter` for Springfield or the `stateCounter` for MA. In the `closeDown()` module, you perform `stateBreak()`, which immediately performs `cityBreak()`. Within `cityBreak()`, the count for Springfield prints and rolls up to the `stateCounter`. Then, after the logic transfers back to the `stateBreak()` module, the total for MA prints and rolls up to `grandTotal`. Finally, you can print `grandTotal`, as shown in Figure 7-25.

FIGURE 7-25: FLOWCHART AND PSEUDOCODE FOR `closeDown()` MODULE




```
closeDown()
     perform stateBreak()
     print "Grand Total", grandTotal
     close files
return
```

Every time you write a program where you need control break routines, you should check whether you need to complete each of the following tasks within the modules:

- Performing the lower-level break, if any
- Performing any control break processing for the previous group
- Rolling up the current-level totals to the next higher level
- Resetting the current level's totals to zero
- Performing any control break processing for the new group
- Updating the control break field

PERFORMING PAGE BREAKS

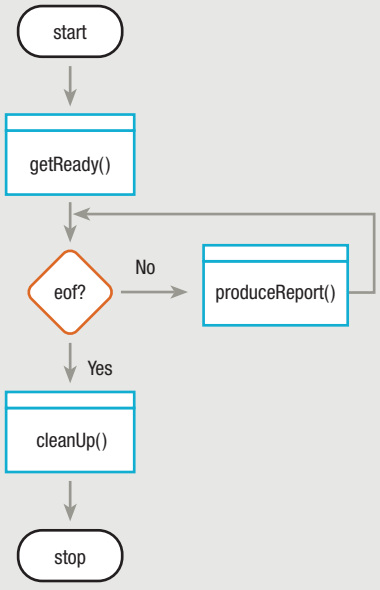
Many business programs use a form of control break logic to start a new page when a printed page fills up with output. In other words, you might want the change to a new page to be based on the number of lines already printed, rather than on the contents of an input field, such as department number. The logic in these programs involves counting the lines printed, pausing to print headings when the counter reaches some predetermined value, and then going on. This common business task is just another example of providing a break in the usual flow of control.

TIP

Some programmers may prefer to reserve the term *control break* for situations in which the break is based on the contents of one of the fields in an input record, rather than on the contents of a work field such as a line counter.

Let's say you have a file called CUSTOMERFILE containing 1000 customers, with two character fields that you have decided to call `custLast` and `custFirst`. You want to print a list of these customers, 60 detail lines to a page. The mainline logic of the program is familiar (see Figure 7-26). The only new feature is a variable called a line counter. You will use a **line-counter** variable to keep track of the number of printed lines, so that you can break to a new page after printing 60 lines.

FIGURE 7-26: MAINLINE LOGIC OF CUSTOMER REPORT PROGRAM




```
start
     perform getReady()
     while not eof
          perform produceReport()
     endwhile
     perform cleanUp()
stop
```

TIP

You first learned about detail lines in Chapter 3. Detail lines contain individual record data, as opposed to summary lines, which typically contain counts, totals, or other group information culled from multiple records.

TIP When creating a printed report, you need to clarify whether the user wants a specific number of *total* lines per page, including headings, or a specific number of *detail* lines per page following the headings. In other words, you must determine whether headings should "count" as part of the number of lines requested.

TIP Although you might require any specific number of lines per page, this example uses 60 because it represents a commonly used limit. Printing is most legible with the least waste at about six lines per inch, so 60 lines fit comfortably on standard 11-inch paper.

Within the `getReady()` module (Figure 7-27), you declare the variables, open the files, print the headings, and read the first record. Within the `produceReport()` module (Figure 7-28), you compare `lineCounter` to 60. When you process the first record, `lineCounter` is 0, so you print the record, add 1 to `lineCounter`, and read the next record.

FIGURE 7-27: THE `getReady()` MODULE FOR CUSTOMER REPORT PROGRAM

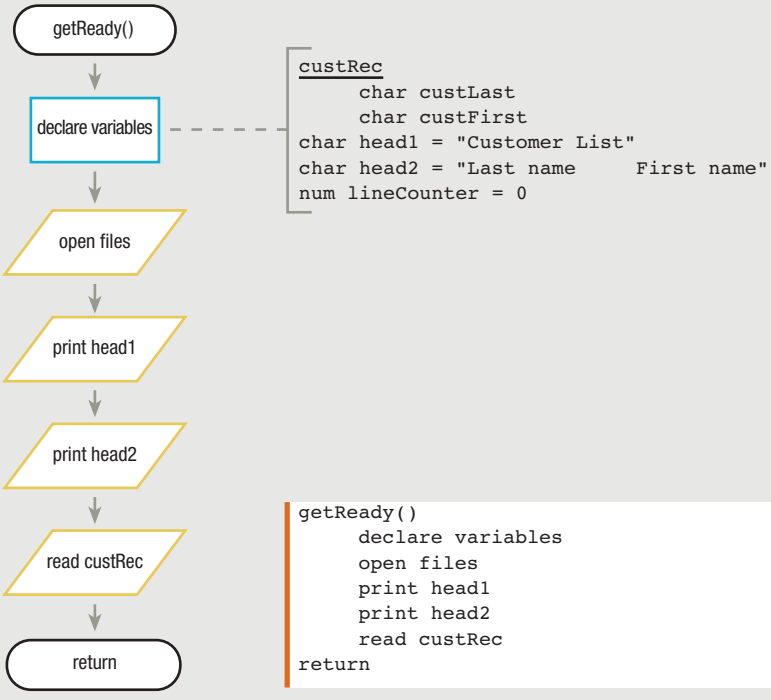
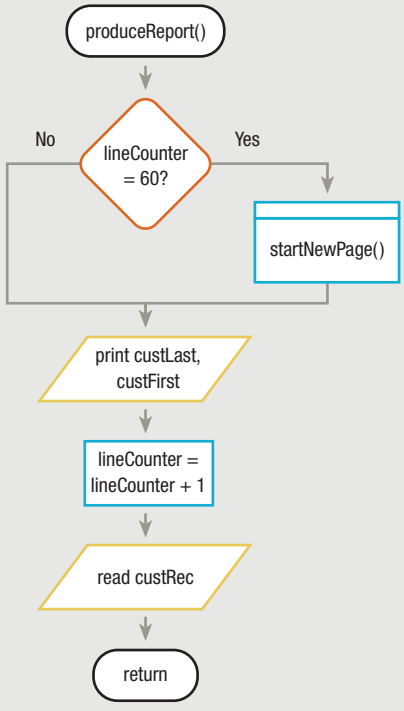


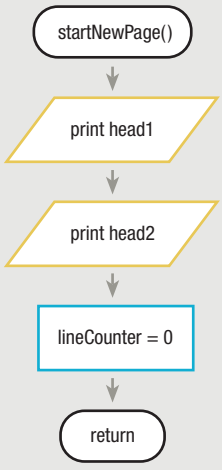
FIGURE 7-28: THE `produceReport()` MODULE FOR CUSTOMER REPORT PROGRAM




```
produceReport()
     if lineCounter = 60 then
          perform startNewPage()
     endif
     print custLast, custFirst
     lineCounter = lineCounter + 1
     read custRec
return
```

In Figure 7-27, instead of printing `head1` and `head2`, you could perform a module that starts a new page. Figure 7-29 shows a `startNewPage()` module that the `getReady()` module could call.

FIGURE 7-29: THE `startNewPage()` MODULE FOR CUSTOMER REPORT PROGRAM




```
startNewPage()
     print head1
     print head2
     lineCounter = 0
return
```

On every cycle through the `produceReport()` module, you check the line counter to see if it is 60 yet. When the first record is printed, `lineCounter` is 1. You read the second record, and if there is a second record (that is, if it is not `eof`), you return to the top of the `produceReport()` module. In that module, you compare `lineCounter` to 60, print another line, and add 1 to `lineCounter`, making it equal to 2.

After 60 records are read and printed, `lineCounter` holds a value of 60. When you read the 61st record (and if it is not `eof`), you enter the `produceReport()` module for the 61st time. The answer to the question `lineCounter = 60?` is Yes, and you break to perform the `startNewPage()` module. The `startNewPage()` module is a control break routine.

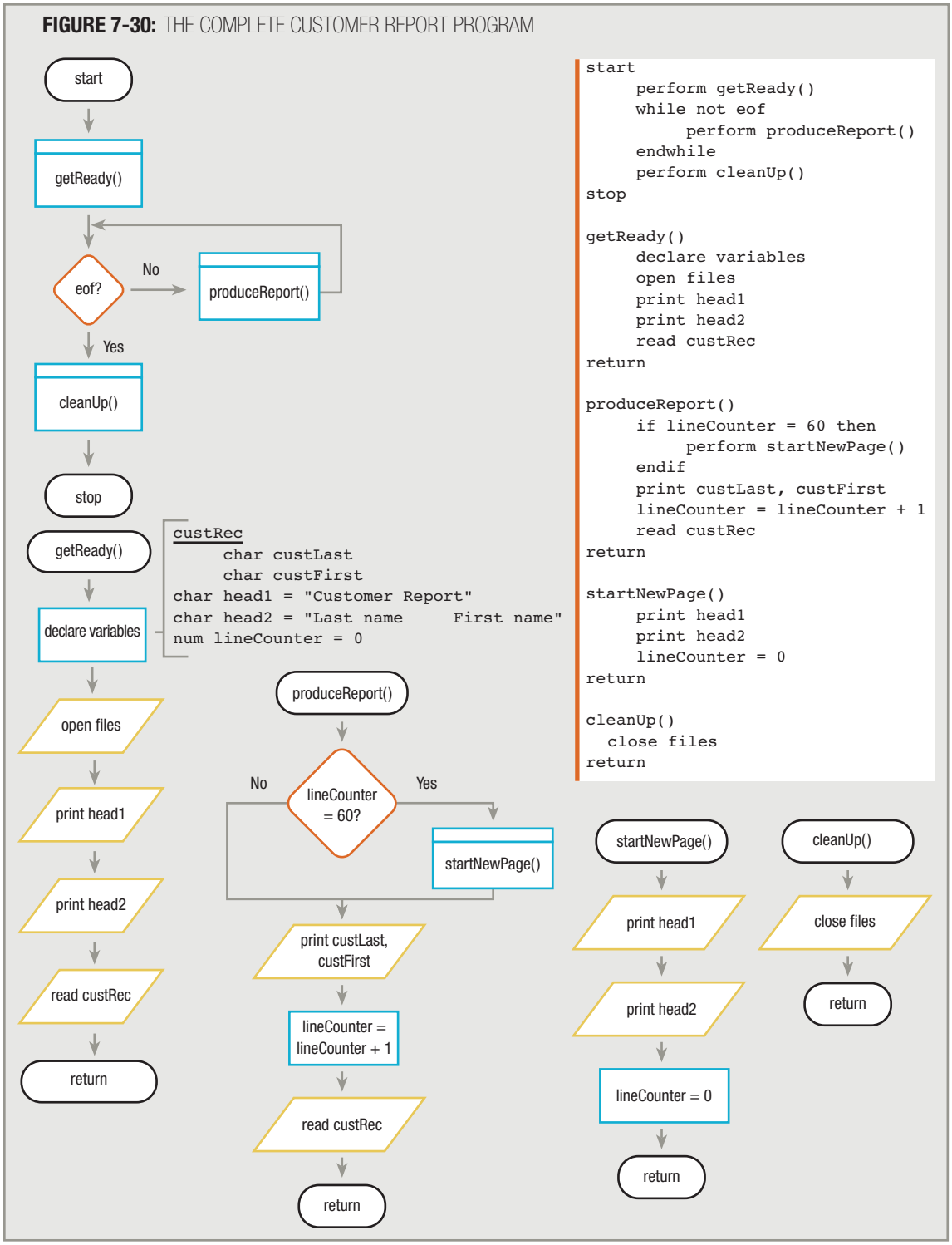
The `startNewPage()` module, shown in Figure 7-29, must print the headings that appear at the top of a new page, and it must set `lineCounter` back to zero. If you neglect to reset `lineCounter`, its value will increase with each successive record and never be equal to 60 again. When resetting `lineCounter` for a new page, you force execution of the `startNewPage()` module after 60 more records (120 total) print.

The `startNewPage()` module is simpler than many control break modules because no record counters or accumulators are being maintained. In fact, the `startNewPage()` module must perform only two of the tasks you have seen required by control break routines.

- It does not perform the lower-level break, because there is none.
- It does not perform any control break processing for the previous group, because there is none.
- It does not roll up the current-level totals to the next higher level, because there are no totals.
- It does not reset the current level's totals to zero, because there are no totals (other than `lineCounter`, which is the control break field).
- It does perform control break processing for the new group by printing headings at the top of the new page.
- It does update the control break field—the line counter.

You might want to employ one little trick to remove the statements that print the headings from the `getReady()` module. If you initialize `lineCounter` to 60 when defining the variables at the beginning of the program, on the first pass through `produceReport()`, you can "fool" the computer into printing the first set of headings automatically. When you initialize `lineCounter` to 60, you can remove the statements `print head1` and `print head2` from the `getReady()` module. With this change, when you enter the `produceReport()` module for the first time, `lineCounter` is already set to 60, and the `startNewPage()` module prints the headings and resets `lineCounter` to zero before processing the first record from the input file and starting to count the first page's detail lines. Figure 7-30 shows the entire program.

FIGURE 7-30: THE COMPLETE CUSTOMER REPORT PROGRAM



TIP

In the program in Figure 7-30, you might prefer to create a constant named LINES_PER_PAGE and set it to be equal to 60. Then, in the `produceReport()` module, you would compare `lineCounter` to this constant. Doing this would provide you with two advantages. First, the meaning of LINES_PER_PAGE would be clearer than the number 60. Second, if you needed to change the number of lines per page, you could do so using the declaration list instead of searching through the program to find the reference.

As with control break report programs that break based on the contents of one of a record's fields, in any program that starts new pages based on a line count, you always must update the line-counting variable that causes the unusual action. Using page breaks or control breaks (or both) within reports adds a new degree of organization to your printed output and makes it easier for the user to interpret and use.

CHAPTER SUMMARY

- A control break is a temporary detour in the logic of a program; programmers refer to a program as a control break program when a change in the value of a variable initiates special actions or causes special or unusual processing to occur. To generate a control break report, your input records must be organized in sorted order based on the field that will cause the breaks.
- You use a control break field to hold data from a previous record. You decide when to perform a control break routine by comparing the value in the control break field to the corresponding value in the current record. At minimum, the simplest control break routines perform necessary processing for the new group and update the control break field.
- Sometimes, you need to use control data within a control break module, such as in a heading that requires information about the next record, or in a footer that requires information about the previous record. The very first heading prints separately from all the others at the beginning; the very last footer must print separately from all the others at the end.
- A control break report contains and prints totals for the previous group, rolls up the current-level totals to the next higher level, resets the current level's totals to zero, performs any other needed control break processing, and updates the control break field.
- In a program containing a multiple-level control break, the normal flow of control breaks away for special processing in response to a change in more than one field. You should always test for a major-level break before a minor-level break, and include a call to the minor break routine within the major break module.
- Every time you write a program in which you need control break routines, you should check whether you need to perform each of the following tasks within the routines: any lower-level break, any control break processing for the previous group, rolling up the current-level totals to the next higher level, resetting the current level's totals to zero, any control break processing for the new group, and updating the control break field.
- To perform page breaks, you count the lines printed and pause to print headings when the counter reaches some predetermined value.

KEY TERMS

A **control break** is a temporary detour in the logic of a program.

A **control break program** is one in which a change in the value of a variable initiates special actions or causes special or unusual processing to occur.

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

Programs that **sort** records take records that are not in order and rearrange them to be in order based on some field.

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

A **control break field** is a variable that holds the value that signals a break in a program.

A **footer** is a message that prints at the end of a page or other section of a report.

Rolling up the totals is the process of adding a total to a higher-level total.

A **summary report** is one that does not include any information about individual records, but instead includes only group totals.

A **multiple-level control break** is one in which the normal flow of control breaks away for special processing in response to a change in more than one field.

A **major-level break** is a break in the flow of logic that is caused by a change in the value of a higher-level field.

A **minor-level break** is a break in the flow of logic that is caused by a change in the value of a lower-level field.

A **line-counter** variable keeps track of the number of printed lines on a page.

REVIEW QUESTIONS

1. A control break occurs when a program _____.

a. takes one of two alternate courses of action for every record
b. pauses to perform special processing based on the value of a field
c. ends prematurely, before all records have been processed
d. passes logical control to a module contained within another program

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

a. a list of all employees in a company, with a message "Retain" or "Dismiss" following each employee record
b. a list of all students in a school, arranged in alphabetical order, with a total count at the end of the report
c. a list of all customers of a business in zip code order, with a count of the number of customers who reside in each zip code
d. a list of some of the patients of a medical clinic—those who have not seen a doctor for at least two years

3. Placing records in sequential order based on the value in one of the fields is called _____.

a. sorting
b. collating
c. merging
d. categorizing

4. In a program with a single-level control break, _____.

a. the input file must contain a variable that contains a single digit
b. the hierarchy chart must contain a single level below the main level
c. special processing occurs based on the value in a single field
d. the control break module must not contain any submodules

5. A control break field _____.

a. always prints prior to any group of records on a control break report
b. always prints after any group of records on a control break report
c. never prints on a report
d. causes special processing to occur

6. The value stored in a control break field _____.

a. can be printed at the end of each group of records
b. can be printed with each record
c. both of these
d. neither a nor b

7. Within any control break module, you must _____.

a. declare a control break field
b. set the control break field to zero
c. print the control break field
d. update the value in the control break field

8. An insurance agency employs 10 agents and wants to print a report of claims based on the insurance agent who sold each policy. The agent's name should appear in a heading prior to the list of each agent's claims. In the housekeeping module for this program, you should _____.

a. read the first record before printing the first heading
b. print the first heading before reading the first record
c. read all the records that represent clients of the first agent before printing the heading
d. print the first heading, but do not read the first record until the main loop

9. In contrast to using control break data in a heading, when you use control break data in a footer, you usually need data from the _____ record in the input data file.

a. previous
b. next
c. first
d. priming

10. An automobile dealer wants a list of cars sold, grouped by model, with a total dollar amount sold at the end of each group. The program contains four modules, appropriately named `housekeeping()`, `mainLoop()`, `modelBreak()`, and `finish()`. The total for the last car model group should be printed in the _____.

a. `mainLoop()` module, after the last time the control break module is called
b. `mainLoop()` module, as the last step in the module
c. `modelBreak()` module when it is called from within the `mainLoop()` module
d. `modelBreak()` module when it is called from within the `finish()` module

11. The Hampton City Zoo has a file that contains information about each of the animals it houses. Each animal record contains such information as the animal's ID number, date acquired by the zoo, and species. The zoo wants to print a list of animals, grouped by species, with a count after each group. As an example, a typical summary line might be "Species: Giraffe Count: 7". Which of the following happens within the control break module that prints the count?

a. The previous species count prints, and then the previous species field is updated.
b. The previous species field is updated, and then the previous species count prints.
c. Either of these will produce the desired results.
d. Neither a nor b will produce the desired results.

12. Adding a total to a higher-level total is called _____ the totals.

a. sliding
b. advancing
c. rolling up
d. replacing

13. The Academic Dean of Creighton College wants a count of the number of students who have declared each of the college's 45 major courses of study, as well as a grand total count of students enrolled in the college. Individual student records contain each student's name, ID number, major, and other data, and are sorted in alphabetical order by major. A control break module executes when the program encounters a change in student major. Within this module, what must occur?

a. The total count for the previous major prints.
b. The total count for the previous major prints, and the total count is added to the grand total.
c. The total count for the previous major prints, the total count for the major is added to the grand total, and the total count for the major is reset to zero.
d. The total count for the previous major prints, the total count for the major is added to the grand total, the total count for the major is reset to zero, and the grand total is reset to zero.

14. In a control break program containing printed group totals and a grand total, the final module that executes must _____.

a. print the group total for the last group
b. roll up the total for the last group
c. both of these
d. neither a nor b

15. A summary report _____.

a. contains detail lines
b. contains total lines
c. both of these
d. neither a nor b

16. The Cityscape Real Estate Agency wants a list of all housing units sold last year, including a subtotal of sales that occurred each month. Within each month group, there are also subtotals of each type of property—single-family homes, condominiums, commercial properties, and so on. This report is a _____ control break report.

a. single-level
b. multiple-level
c. semilevel
d. trilevel

17. The Packerville Parks Commission has a file that contains picnic permit information for the coming season. They need a report that lists each day's picnic permit information, including permit number and name of permit holder, starting on a separate page each day of the picnic season. (Figure 7-31 shows a sample page of output for the Packerville Parks report.) Within each day's permits, they want subtotals that count permits in each of the city's 30 parks. The permit records have been sorted by park name within date. In the main loop of the report program, the first decision should check for a change in _____.

FIGURE 7-31: SAMPLE PARKS REPORT

```
     Packerville Parks Commission - Daily Count of Permits by Park
Day: June 24

     Permit Number        Permit Holder
     200501932            Paul Martin
     200502003            Brownie Troop 176
     200502015            Dorothy Wintergreen
                                       Alcott Park Count - 3
     200500080            YMCA Day Camp
     200501200            Packerville Rotary Club
     200501453            Harold Martinez
     200502003            Wendy Sudo
                                       Browning Park Count - 4
```

a. park name
b. date
c. permit number
d. any of these

18. Which of the following is *not* a task you need to complete in any control break module that has multiple levels and totals at each level?

a. Perform lower-level breaks.
b. Roll up the totals.
c. Update the control break field.
d. Reset the current-level totals to the previous-level totals.

19. The election commission for the state of Illinois maintains a file that contains the name of each registered voter, the voter's county, and the voter's precinct within the county. The commission wants to produce a report that counts the voters in each precinct and county. The file should be sorted in _____.

a. county order within precinct
b. last name order within precinct
c. last name order within county
d. precinct order within county

20. A variable that determines when a new page should start based on the number of detail lines printed on a page is a _____.

a. detail counter
b. line counter
c. page counter
d. break counter

FIND THE BUGS

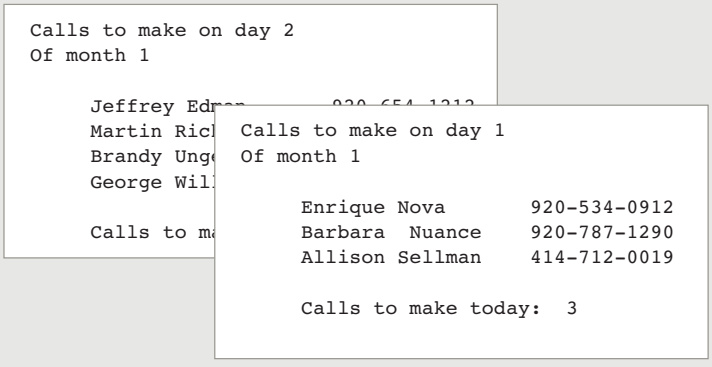
Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. **This application prints a student report for an elementary school. Students have been sorted by grade level. A new page is started for each grade level, and the numeric grade level prints as part of the heading of the page.**

```
start
     perform getReady()
     while not eof
          perform produceReport()
     endwhile
     perform finishUp()
stop

getReady()
     declare variables
         studentRec
             num studentID
             char name
             num gradeLevel
          char heading1 = "Student Report by Grade Level"
          char heading2 = "Students in Grade "
     open files
     print heading1
     print heading2, gradeLevel
     read studentRec
return

produceReport()
     if gradeLevel = holdGradeLevel then
         perform newGrade()
     endif
     print studentId, name
     read studentRec
return

newGrade()
     print heading1
     print heading2, holdGradeLevel
      holdGradeLevel = gradeLevel
return

finishUp()
     close files
return
```

2. **The Friendly Insurance Company makes a point to phone a birthday greeting to each of its clients on his or her birthday. The following program is intended to produce a report that lists the clients a salesperson should call each day for the coming year. Input records include the client's name and phone number as well as a numeric month and day. The records have been sorted by day within month, and each day's list appears on a new page. (It is very likely that some days of the year do not have a client birthday.) At the end of each page is a count of the number of calls that should be made that day. Two pages of a sample report are shown in Figure 7-32.**

FIGURE 7-32: SAMPLE REPORT




```
start
     perform prepare()
     while not eof
          perform produceReport()
     endwhile
     perform finish()
stop

prepare()
     declare variables
         appointmentRec
             char clientName
             char phoneNumber
             num month
             num day
          num oldMonth
          num oldDay
          char heading1 = "Calls to make on day "
          char heading2 = "of month "
          char footer = "Calls to make today: "
          num countAppointments
```

```
     open files
     read appointmentRec
     print head1, day
     print heading2, month
     month = oldMonth
     day = oldDay
return

produceReport()
     if day not = oldDay then
         perform newDay()
      else
            if month not = oldMonth
                perform newMonth()
            endif
     endif
     print clientName, phoneNumber
     countAppointments = countAppointments + 1
return

newMonth()
     perform newDay()
     oldMonth = month
return

newDay()
    perform newMonth()
    print footer, countAppointments
    print heading1, day
    print heading1, month
    oldDay = day
return

finish()
     close files
return
```

EXERCISES

1. **What fields might you want to use as the control break fields to produce a report that lists all inventory items in a grocery store? (For example, you might choose to group items by grocery store department.) Design a sample report.**

2. **What fields might you want to use as the control break fields to produce a report that lists all the people you know? (For example, you might choose to group friends by city of residence.) Design a sample report.**

3. **Cool's Department Store keeps a record of every sale in the following format:**

```
DEPARTMENT STORE SALES FILE DESCRIPTION
File name: DEPTSALES
Sorted by: Department
FIELD DESCRIPTION       DATA TYPE           COMMENTS
Transaction Number      Numeric             a 6-digit number
Amount                  Numeric             2 decimal places
Department              Numeric             a 3-digit number
```

Create the logic for a program that would print each transaction's details, with a total at the end of each department.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Create the flowchart.
d. Create the pseudocode.

4. **A used-car dealer keeps track of sales in the following format:**

```
AUTO SALES FILE DESCRIPTION
File name: AUTO
Sorted by: Salesperson
FIELD DESCRIPTION       DATA TYPE       EXAMPLE
Salesperson             Character       Miller
Make of Car             Character       Ford
Vehicle Type            Character       Sedan
Sale Price              Numeric         0 decimal places; for example, 15000
```

By the end of the week, a salesperson may have sold no cars, one car, or many cars. Create the logic of a program that would print one line for each salesperson, with that salesperson's total sales for the week and commission earned, which is 4 percent of the total sales.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Create the flowchart.
d. Create the pseudocode.

5. A community college maintains student records in the following format:

```
STUDENT FILE DESCRIPTION
File name: STUDENTS
Sorted by: Hour of First Class
FIELD DESCRIPTION       DATA TYPE         EXAMPLE
Student Name            Character         Amy Lee
City                    Character         Woodstock
Hour of First Class     Numeric           08 for 8 a.m. or 14 for 2 p.m.
Phone Number            Numeric           8154379823
```

The records have been sorted by hour of the day. The Hour of First Class is a two-digit number based on a 24-hour clock (for example, a 1 p.m. first class is recorded as 13).

Create a report that students can use to organize carpools. The report lists the names and phone numbers of students from the city of Huntley. Note that some students come from cities other than Huntley; these students should not be listed on the report.

Start a new page for each hour of the day, so that all students starting classes at the same hour are listed on the same page. Include the hour that each page represents in the heading for that page.

a. Design the output for this program; create either sample output or a print chart.
b. Create the hierarchy chart.
c. Create the flowchart.
d. Create the pseudocode.

6. **The Stanton Insurance Agency needs a report summarizing the counts of life, health, and other types of insurance policies it sells. Input records contain policy number, name of insured, policy value, and type of policy, and have been sorted in alphabetical order by type of policy. At the end of the report, display a count of all the policies.**

   a. Design the output for this program; create either sample output or a print chart.
   b. Create the hierarchy chart.
   c. Create the flowchart.
   d. Create the pseudocode.

7. **If a university is organized into colleges (such as Liberal Arts), divisions (such as Languages), and departments (such as French), what would constitute the major, intermediate, and minor control breaks in a report that prints all classes offered by the university?**

8. **A zoo keeps track of the expense of feeding the animals it houses. Each record holds one animal's ID number, name, species (elephant, rhinoceros, tiger, lion, and so on), zoo residence (pachyderm house, large-cat house, and so on), and weekly food budget. The records take the following form:**

```
ANIMAL FEED RECORDS
File name: ANIMFOOD
Sorted by: Species within house
FIELD DESCRIPTION       DATA TYPE       EXAMPLE
Animal ID               Numeric         4116
Animal Name             Character       Elmo
Species                 Character       Elephant
House                   Character       Pachyderm
Weekly Food             Numeric         0 decimals, whole dollars; for example, 75
   Budget in Dollars
```

   Design a report that lists each animal's ID, name, and budgeted food amount. At the end of each species group, print a total budget for the species. At the end of each house (for example, the species lion, tiger, and leopard are all in the large-cat house), print the house total. At the end of the report, print the grand total.

   a. Design the output for this program; create either sample output or a print chart.
   b. Create the hierarchy chart.
   c. Create the flowchart.
   d. Create the pseudocode.

9. **A soft-drink manufacturer produces several flavors of drink—for example, cola, orange, and lemon. Additionally, each flavor has several versions, such as regular, diet, and caffeine-free. The manufacturer operates factories in several states.**

   Assume you have input records that list version, flavor, yearly production in gallons, and state (for example: Regular Cola 5000 Kansas). The records have been sorted in alphabetical order by version within flavor within state. Design the report that lists each version and flavor, with minor total production figures for each flavor and major total production figures for each state.

   a. Design the output for this program; create either sample output or a print chart.
   b. Create the hierarchy chart.
   c. Create the flowchart.
   d. Create the pseudocode.

10. **An art shop owner maintains records for each item in the shop, including the title of the work, the artist who made the item, the medium (for example, watercolor, oil, or clay), and the monetary value. The records are sorted by artist within medium. Design a report that lists all items in the store, with a minor total value following each artist's work, and a major total value following each medium. Allow only 40 detail lines per page.**

    a. Design the output for this program; create either sample output or a print chart.
    b. Create the hierarchy chart.
    c. Create the flowchart.
    d. Create the pseudocode.

DETECTIVE WORK

1. **Control break reports are just one type of frequently printed business report. Has paper consumption increased or decreased since computers became common office tools? How soon do experts predict we will have the "paperless office"?**

UP FOR DISCUSSION

1. **Suppose your employer asks you to write a control break program that lists all the company's employees, their salaries, and their ages, with breaks at each department to list a count of employees in that department. You are provided with the personnel file to use as input. You decide to take the file home with you so you can work on creating the report over the weekend. Is this acceptable? What if the file contained only employees' names and departments, and not more sensitive data such as salaries and ages?**

2. **Suppose your supervisor asks you to create a report that lists all employees by department and includes a break after each department to display the highest-paid employee in that department. Suppose you also know that your employer will use this report to lay off the highest-paid employee in each department. Would you agree to write the program? Instead, what if the report's purpose was to list the worst performer in each department in terms of sales? What if the report grouped employees by gender? What if the report grouped employees by race?**

3. **Suppose your supervisor asks you to write a control break report that lists employees in groups by the dollar value of medical insurance claims they have in a year. You fear the employer will use the report to eliminate workers who are driving up the organization's medical insurance policy costs. Do you agree to write the report? What if you know for certain that the purpose of the report is to eliminate workers?**