6 LOOPING

After studying Chapter 6, you should be able to:

- Understand the advantages of looping
- Control a `while` loop using a loop control variable
- Increment a counter to control a loop
- Loop with a variable sentinel value
- Control a loop by decrementing a loop control variable
- Avoid common loop mistakes
- Use a `for` statement
- Use `do while` and `do until` loops
- Recognize the characteristics shared by all loops
- Nest loops
- Use a loop to accumulate totals

UNDERSTANDING THE ADVANTAGES OF LOOPING

If making decisions is what makes computers seem intelligent, it's looping that makes computer programming worthwhile. When you use a loop within a computer program, you can write one set of instructions that operates on multiple, unique sets of data. Consider the following set of tasks required for each employee in a typical payroll program:

- Determine regular pay.
- Determine overtime pay, if any.
- Determine federal withholding tax based on gross wages and number of dependents.
- Determine state withholding tax based on gross wages, number of dependents, and state of residence.
- Determine insurance deduction based on insurance code.
- Determine Social Security deduction based on gross pay.
- Subtract federal tax, state tax, Social Security, and insurance from gross pay.

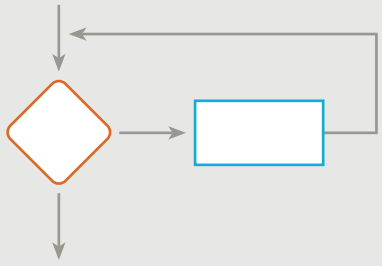
In reality, this list is too short—companies deduct stock option plans, charitable contributions, union dues, and other items from checks in addition to the items mentioned in this list. Also, they might pay bonuses and commissions and provide sick days and vacation days that must be taken into account and handled appropriately. As you can see, payroll programs are complicated.

The advantage of having a computer perform payroll calculations is that all of the deduction instructions need to be written *only once* and can be repeated over and over again for each paycheck using a **loop**, the structure that repeats actions while some condition continues.

USING A WHILE LOOP WITH A LOOP CONTROL VARIABLE

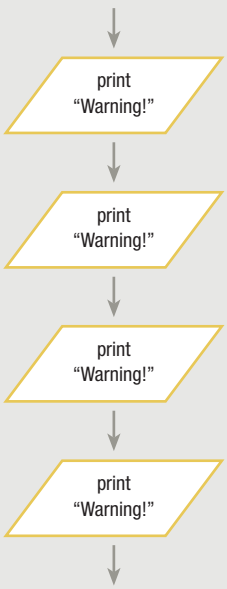
Recall the loop, or `while` structure, that you learned about in Chapter 2. (See Figure 6-1.) In Chapter 4, you learned that almost every program has a **main loop**, or a basic set of instructions that is repeated for every record. The main loop is a typical loop—within it, you write one set of instructions that executes repeatedly while records continue to be read from an input file. Several housekeeping tasks execute at the start of most programs, and a few cleanup tasks execute at the end. However, most of a program's tasks are located in a main loop; these tasks repeat over and over for many records (sometimes hundreds, thousands, or millions).

FIGURE 6-1: THE `while` LOOP



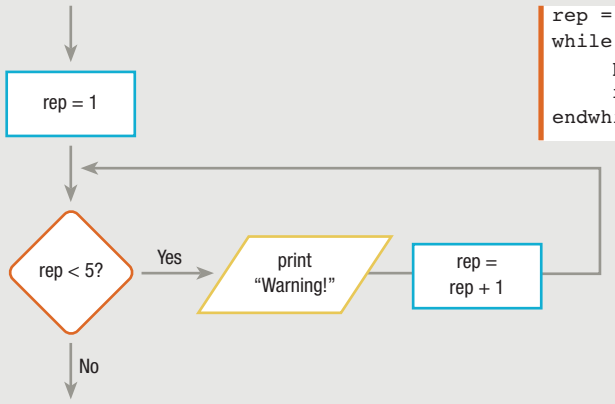
In addition to this main loop, loops also appear within a program's modules. They are used any time you need to perform a task several times and don't want to write identical or similar instructions over and over. Suppose, for example, as part of a much larger program, you want to print a warning message on the computer screen when the user has made a potentially dangerous menu selection (for example, "Delete all files"). To get the user's attention, you want to print the message four times. You can write this program segment as a sequence of four steps, as shown in Figure 6-2, but you can also use a loop, as shown in Figure 6-3.

FIGURE 6-2: PRINTING FOUR WARNING MESSAGES IN SEQUENCE




```
print "Warning!"
print "Warning!"
print "Warning!"
print "Warning!"
```

FIGURE 6-3: PRINTING FOUR WARNING MESSAGES IN A LOOP




```
rep = 1
while rep < 5
     print "Warning!"
     rep = rep + 1
endwhile
```

The flowchart and pseudocode segments in Figure 6-3 show three steps that should occur in every loop:

1. You initialize a variable that will control the loop. The variable in this case is named `rep`.
2. You compare the variable to some value that controls whether the loop continues or stops. In this case, you compare `rep` to the value 5.
3. Within the loop, you alter the variable that controls the loop. In this case, you alter `rep` by adding 1 to it.

On each pass through the loop, the value in the `rep` variable determines whether the loop will continue. Therefore, variables like `rep` are known as **loop control variables**. Any variable that determines whether a loop will continue to execute is a loop control variable. To stop a loop's execution, you compare the loop control value to a **sentinel value** (also known as a limit or ending value), in this case the value 5. The decision that controls every loop is always based on a Boolean comparison. You can use any of the six comparison operators that you learned about in Chapter 5 to control a loop—equal to, greater than, less than, greater than or equal to, less than or equal to, and not equal to.

TIP Just as with a selection, the Boolean comparison that controls a `while` loop must compare same-type values: numeric values are compared to other numeric values, and character values to other character values.

The statements that execute within a loop are known as the **loop body**. The body of a loop might contain any number of statements, including method calls, sequences, decisions, and other loops. Once your program enters the body of a structured loop, the entire loop body must execute. Your program can leave a structured loop only at the comparison that tests the loop control variable.

USING A COUNTER TO CONTROL LOOPING

Suppose you own a factory and have decided to place a label on every product you manufacture. The label contains the words "Made for you personally by " followed by the first name of one of your employees. For one week's production, suppose you need 100 personalized labels for each employee.

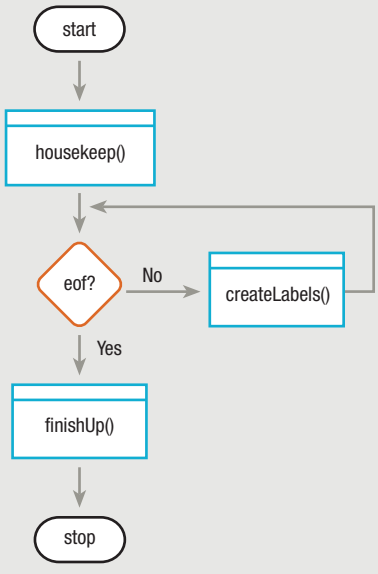
Assume you already have a personnel file that can be used for input. This file has more information than you'll need for this program: an employee last name, first name, Social Security number, address, date hired, and salary. The important feature of the file is that it does contain each employee's name stored in a separate record. The input file description appears in Figure 6-4.

FIGURE 6-4: EMPLOYEE FILE DESCRIPTION

```
File Name: EMPFILE
FIELD DESCRIPTION          DATA TYPE      COMMENTS
Employee Last Name         Character      20 characters
Employee First Name        Character      15 characters
Social Security Number     Numeric        0 decimal places
Address                    Character      15 characters
Date Hired                 Numeric        8 digits, YYYYMMDD
Hourly Salary              Numeric        2 decimal places
```

In the mainline logic of this program, you call three modules: a housekeeping module (`housekeep()`), a main loop module (`createLabels()`), and a finish routine (`finishUp()`). See Figure 6-5.

FIGURE 6-5: MAINLINE LOGIC FOR LABEL-MAKING PROGRAM




```
start
     perform housekeep()
     while not eof
          perform createLabels()
     endwhile
     perform finishUp()
stop
```

The first task for the label-making program is to name the fields in the input record so you can refer to them within the program. As a programmer, you can choose any variable names you like, for example: `inLastName`, `inFirstName`, `inSSN`, `inAddress`, `inDate`, and `inSalary`.

TIP

In Chapter 4 you learned that starting all field names in the input record with the same prefix, such as `in`, is a common programming technique to help identify these fields in a large program and differentiate them from work areas and output areas that will have other names. Another benefit to using a prefix like `in` is that some language compilers produce a dictionary of variable names when you compile your program. These dictionaries show at which lines in the program each data name is referenced. If all your input field names start with the same prefix, they will be together alphabetically in the dictionary, and perhaps be easier to find and work with.

You also can set up a variable to hold the characters "Made for you personally by " and name it `labelLine`. You eventually will print this `labelLine` variable followed by the employee's first name (`inFirstName`).

You will need one more variable: a location to be used as a counter. A **counter** is any numeric variable you use to count the number of times an event has occurred; in this example, you need a counter to keep track of how many labels have been printed at any point. Each time you read an employee record, the counter variable is set to 0. Then every time a label is printed, you add 1 to the counter. Adding to a variable is called **incrementing** the variable; programmers often use the term "incrementing" specifically to mean "increasing by one." Before the next employee label is printed, the program checks the variable to see if it has reached 100 yet. When it has, that means 100 labels have been printed, and the job is done for that employee. While the counter remains below 100, you continue to print labels. As with all variables, the programmer can choose any name for a counter; this program uses `labelCounter`. In this example, `labelCounter` is the loop control variable.

The `housekeep()` module for the label program, shown in Figure 6-6, includes a step to open the files: the employee file and the printer. Unlike a program that produces a report, this program produces no headings, so the next and last task performed in `housekeep()` is to read the first input record.

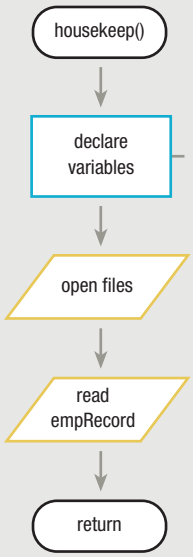
TIP

Remember, you can give any name to modules within your programs. This program uses `housekeep()` for its first routine, but `housekeeping()`, `startUp()`, `prep()`, or any other name with the same general meaning could be used.

TIP

If you don't know why the first record is read in the `housekeep()` module, go back and review the concept of the priming read, presented in Chapter 2.

FIGURE 6-6: THE `housekeep()` MODULE FOR THE LABEL PROGRAM




```
empRecord
     char inLastName
     char inFirstName
     num inSSN
     char inAddress
     num inDate
     num inSalary
char labelLine = "Made for you
          personally by "
num labelCounter
```

```
housekeep()
     declare variables
     open files
     read empRecord
return
```

TIP

The label-making program could be interactive instead of reading data from a file. An easy way to make the program interactive would be to replace the `read empRecord` statement with a series of statements or a call to a module that provides a prompt and a read statement for each of the six data fields needed for each employee. A user could then enter these values from the keyboard. (If this were an interactive program, the programmer would likely require the user to enter data only in the field that is necessary for output—the employee's name.) Also, if this were an interactive program, the user might be asked to type a sentinel value, such as "XXX", when finished. This program is discussed as one that reads from a file to reduce the number of statements you must view to understand the logical process.

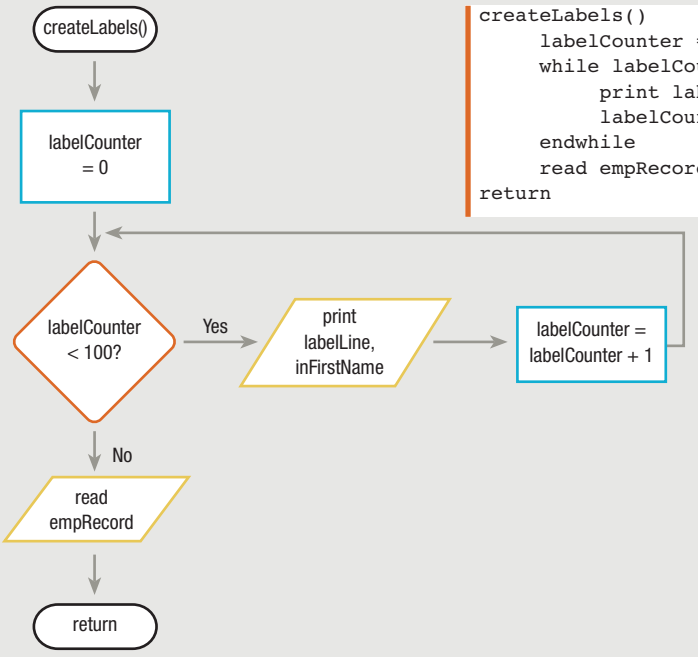
TIP

In previous chapters, the list of declared variables was shown with both the flowchart and the pseudocode. To save space in the rest of the chapters in this book, the variable list will be shown only with the flowchart.

When the `housekeep()` module is done, the logical flow returns to the `eof` question in the mainline logic. If you attempt to read the first record at the end of `housekeep()` and for some reason there is no record, the answer to `eof?` is Yes, so the `createLabels()` module is never entered; instead, the logic of the program flows directly to the `finishUp()` module.

Usually, however, employee records will exist and the program will enter the `createLabels()` module, which is shown in Figure 6-7. When this happens, the first employee record is sitting in memory waiting to be processed. During one execution of the `createLabels()` module, 100 labels will be printed for one employee. As the last event within the `createLabels()` module, the program reads the next employee record. Control of the program then returns to the `eof` question. If the new read process has not resulted in the `eof` condition, control reenters the `createLabels()` module, where 100 more labels print for the new employee.

FIGURE 6-7: THE `createLabels()` MODULE FOR THE LABEL PROGRAM




```
createLabels()
     labelCounter = 0
     while labelCounter < 100
          print labelLine, inFirstName
          labelCounter = labelCounter + 1
     endwhile
     read empRecord
return
```

The `createLabels()` method of this label-making program contains three parts:

- Set `labelCounter` to 0.
- Compare `labelCounter` to 100.
- While `labelCounter` is less than 100, print `labelLine` and `inFirstName`, and add 1 to `labelCounter`.

When the first employee record enters the `createLabels()` module, `labelCounter` is set to 0. The `labelCounter` value is less than 100, so the record enters the label-making loop. One label prints for the first employee, `labelCounter` increases by one, and the logical flow returns to the question `labelCounter < 100?`. After the first label is printed, `labelCounter` holds a value of only 1. It is nowhere near 100 yet, so the value of the Boolean expression is true, and the loop is entered for a second time, thus printing a second label.

After the second printing, `labelCounter` holds a value of 2. After the third printing, it holds a value of 3. Finally, after the 100th label prints, `labelCounter` has a value of 100. When the question `labelCounter < 100?` is asked, the answer will finally be No, and the loop will exit.

Before leaving the `createLabels()` method, and after the program prints 100 labels for an employee, there is one final step: the next input record is read from the EMPLOYEES file. When the `createLabels()` method is over, control returns to the `eof` question in the main line of the logic. If it is not `eof` (if another employee record is present), the program enters the `createLabels()` method again, resets `labelCounter` to 0, and prints 100 new labels with the next employee's name.

TIP

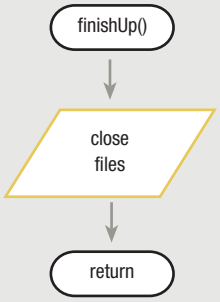
Setting `labelCounter` to 0 when the `createLabels()` module is entered is important. With each new record, `labelCounter` must begin at 0 if 100 labels are to print. When the first employee's set of labels is complete, `labelCounter` holds the value 100. If it is not reset to 0 for the second employee, then no labels will ever print for that employee.

TIP

In this example, the label-making loop executes as `labelCounter` varies from 0 to 100. The program would work just as well if you decided to vary the counter from 1 to 101 or use any other pair of values that differs by 100.

At some point while attempting to read a new record, the program encounters the end of the file, the `createLabels()` module is not entered again, and control passes to the `finishUp()` module. In this program, the `finishUp()` module simply closes the files. See Figure 6-8.

FIGURE 6-8: THE `finishUp()` MODULE FOR THE LABEL PROGRAM




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

LOOPING WITH A VARIABLE SENTINEL VALUE

Sometimes you don't want to be forced to repeat every pass through a loop the same number of times. For example, instead of printing 100 labels for each employee, you might want to vary the number of labels based on how many items a worker actually produces. That way, high-achieving workers won't run out of labels, and less productive workers won't have too many. Instead of printing the same number of labels for every employee, a more sophisticated program prints a different number for each employee, depending on that employee's production the previous week. For example, you might decide to print enough labels to cover 110 percent of each employee's production rate from the previous week; this ensures that the employee will have enough labels for the week, even if his or her production level improves.

For example, assume that employee production data exists in an input file called EMPPRODUCTION in the format shown in Figure 6-9.

A real-life production file would undoubtedly have more fields in each record, but these fields supply more than enough information to produce the labels. You need the first name to print on the label, and you need the field that holds production for the last week in order to calculate the number of labels to print for each employee. Assume this field can contain any number from 0 through 999.

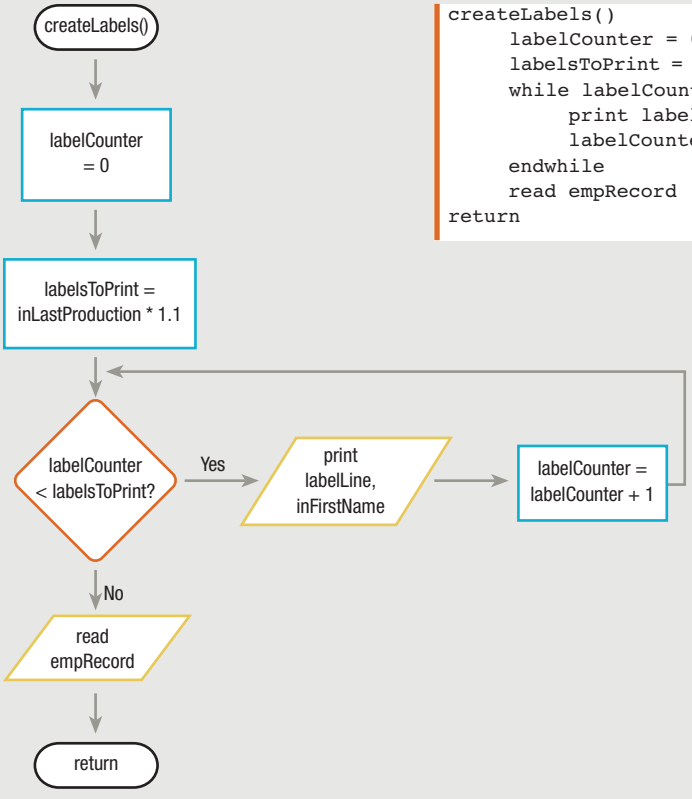
FIGURE 6-9: EMPLOYEE PRODUCTION FILE DESCRIPTION

```
File Name: EMPPRODUCTION
FIELD DESCRIPTION          DATA TYPE      COMMENTS
Last Name                  Character      20 characters
First Name                 Character      15 characters
Production Last Week       Numeric        0 decimal places
```

To write a program that produces an appropriate number of labels for each employee, you can make some minor modifications to the original label-making program. For example, the input file variables have changed; you must declare a variable for an `inLastProduction` field. Additionally, you might want to create a numeric field named `labelsToPrint` that can hold a value equal to 110 percent of a worker's `inLastProduction`.

The major modification to the original label-making program is in the question that controls the label-producing loop. Instead of asking if `labelCounter < 100`, you now can ask if `labelCounter < labelsToPrint`. The sentinel, or limit, value can be a variable like `labelsToPrint` just as easily as it can be a constant like 100. See Figure 6-10 for the flowchart as well as the pseudocode.

FIGURE 6-10: FLOWCHART AND PSEUDOCODE FOR LABEL-MAKING `createLabels()` MODULE




```
createLabels()
     labelCounter = 0
     labelsToPrint = inLastProduction * 1.1
     while labelCounter < labelsToPrint
          print labelLine, inFirstName
          labelCounter = labelCounter + 1
     endwhile
     read empRecord
return
```

TIP

The statement `labelsToPrint = inLastProduction * 1.1` calculates `labelsToPrint` as 110 percent of `inLastProduction`. Alternatively, you can perform the calculation as `labelsToPrint = inLastProduction + 0.10 * inLastProduction`. The mathematical result is the same.

LOOPING BY DECREMENTING

Rather than increasing a loop control variable until it passes some sentinel value, sometimes it is more convenient to reduce a loop control variable on every cycle through a loop. For example, again assume you want to print enough labels for every worker to cover 110 percent production. As an alternative to setting a `labelCounter` variable to 0 and increasing it after each label prints, you initially can set `labelCounter` equal to the number of labels to print (`inLastProduction * 1.1`), and subsequently reduce the `labelCounter` value every time a label prints. You continue printing labels and reducing `labelCounter` until you have counted down to zero. Decreasing a variable is called **decrementing** the variable; programmers most often use the term to mean a decrease by one.

For example, when you write the following, you produce enough labels to equal 110 percent of `inLastProduction`:

```
labelCounter = inLastProduction * 1.1
while labelCounter > 0
     print labelLine, inFirstName
     labelCounter = labelCounter - 1
endwhile
```

TIP

Many languages provide separate numeric data types for whole number (integer) values and floating-point values (those with decimal places). Depending on the data type you choose for `labelCounter`, you might end up calculating a fraction of a label to print. For example, if `inLastProduction` is 5, then the number of labels to produce is 5.5. The logic shown here would print the additional label.

When you decrement, you can avoid declaring a special variable for `labelsToPrint`. The `labelCounter` variable starts with a value that represents the labels to print, and works its way down to zero.

Yet another alternative allows you to eliminate the `labelCounter` variable. You could use the `inLastProduction` variable itself to keep track of the labels. For example, the following pseudocode segment also produces a number of labels equal to 110 percent of each worker's `inLastProduction` value:

```
inLastProduction = inLastProduction * 1.1
while inLastProduction > 0
     print labelLine, inFirstName
     inLastProduction = inLastProduction - 1
endwhile
```

In this example, `inLastProduction` is first increased by 10 percent. Then, while it remains above 0, there are more labels to print; when it is eventually reduced to hold the value 0, all the needed labels will have been printed. With this method, you do not need to create any new counter variables such as `labelCounter`, because `inLastProduction` itself acts as a counter. However, you can't use this method if you need to use the value of `inLastProduction` for this record later in the program. By decrementing the variable, you are changing its value on every cycle through the loop; when you have finished, the original value in `inLastProduction` has been lost.

TIP Do not think the value of `inLastProduction` is gone forever when you alter it. If the data is being read from a file, then the original value still exists within the data file. It is the main memory location called `inLastProduction` that is being reduced.

AVOIDING COMMON LOOP MISTAKES

The mistakes that programmers make most often with loops are:

- Neglecting to initialize the loop control variable
- Neglecting to alter the loop control variable
- Using the wrong comparison with the loop control variable
- Including statements inside the loop that belong outside the loop
- Initializing a variable that does not require initialization

NEGLECTING TO INITIALIZE THE LOOP CONTROL VARIABLE

It is always a mistake to fail to initialize a loop's control variable. For example, assume you remove the statement `labelCounter = 0` from the program illustrated in Figure 6-10. When `labelCounter` is compared to `labelsToPrint` at the start of the `while` loop, it is impossible to predict whether any labels will print. Because uninitialized values contain unknown, unpredictable garbage, comparing such a variable to another value is meaningless. Even if you initialize `labelCounter` to 0 in the `housekeep()` module of the program, you must reset `labelCounter` to 0 for each new record that is processed within the `while` loop. If you fail to reset `labelCounter`, it never surpasses 100 because after it reaches 100, the answer to the question `labelCounter < 100` is always No, and the logic never enters the loop where a label can be printed.

NEGLECTING TO ALTER THE LOOP CONTROL VARIABLE

A different sort of error occurs if you remove the statement that adds 1 to `labelCounter` from the program in Figure 6-10. This error results in the following code:

```
while labelCounter < labelsToPrint
     print labelLine, inFirstName
endwhile
```

Following this logic, if `labelCounter` is 0 and `labelsToPrint` is, for example, 110, then `labelCounter` will be less than `labelsToPrint` forever. Nothing in the loop changes either variable, so when `labelCounter` is less than `labelsToPrint` once, then `labelCounter` is less than `labelsToPrint` forever, and labels will continue to print. A loop that never stops executing is called an **infinite loop**. It is unstructured and incorrect to create a loop that cannot terminate on its own.

TIP Although most programmers advise that infinite loops must be avoided, some programmers argue that there are legitimate uses for them. Intentional uses for infinite loops include programs that are supposed to run continuously, such as product demonstrations, or in programming for embedded systems.

USING THE WRONG COMPARISON WITH THE LOOP CONTROL VARIABLE

Programmers must be careful to use the correct comparison in the statement that controls a loop. Although there is only a one-keystroke difference between the following two code segments, one performs the loop 10 times and the other performs the loop 11 times.

```
counter = 0
while counter < 10
     perform someModule()
     counter = counter + 1
endwhile
```

and

```
counter = 0
while counter <= 10
     perform someModule()
     counter = counter + 1
endwhile
```

The seriousness of the error of using <= or >= when only < or > is needed depends on the actions performed within the loop. For example, if such an error occurred in a loan company program, each customer might be charged a month's additional interest; if the error occurred in an airline's program, it might overbook a flight; and if it occurred in a pharmacy's drug-dispensing program, each patient might receive one extra (and possibly harmful) unit of medication.

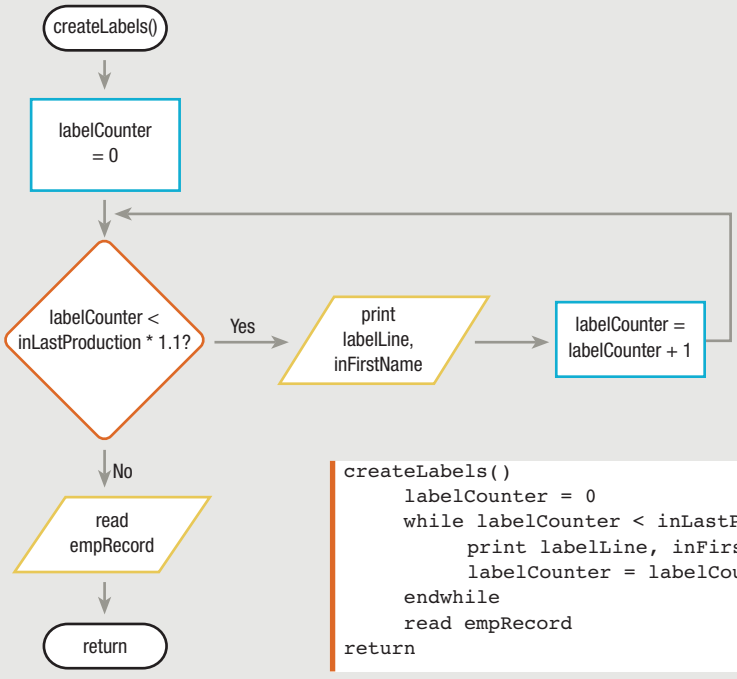
INCLUDING STATEMENTS INSIDE THE LOOP THAT BELONG OUTSIDE THE LOOP

When you run a computer program that uses the loop in Figure 6-10, hundreds or thousands of employee records might pass through the `createLabels()` method. If there are 100 employee records, then `labelCounter` is set to 0 exactly 100 times; it must be reset to 0 once for each employee, in order to count each employee's labels correctly. Similarly, `labelsToPrint` is reset (to 1.1 times the current `inLastProduction` value) once for each employee.

If the average employee produces 100 items during a week, then the loop within the `createLabels()` method, the one controlled by the statement `while labelCounter < labelsToPrint`, executes 11,000 times—110 times each for 100 employees. This number of repetitions is necessary in order to print the correct number of labels.

A repetition that is *not* necessary would be to execute 11,000 separate multiplication statements to recalculate the value to compare to `labelCounter`. See Figure 6-11.

FIGURE 6-11: INEFFICIENT PSEUDOCODE FOR LABEL-MAKING `createLabels()` MODULE




```
createLabels()
     labelCounter = 0
     while labelCounter < inLastProduction * 1.1
          print labelLine, inFirstName
          labelCounter = labelCounter + 1
     endwhile
     read empRecord
return
```

Although the logic shown in Figure 6-11 will produce the correct number of labels for every employee, the statement `while labelCounter < inLastProduction * 1.1` executes an average of 110 times for each employee. That means the arithmetic operation that is part of the question—multiplying `inLastProduction` by 1.1—occurs 110 separate times for each employee. Performing the same calculation that results in the same mathematical answer 110 times in a row is inefficient. Instead, it is superior to perform the multiplication just once for each employee and use the result 110 times, as shown in the original version of the program in Figure 6-10. In the pseudocode in Figure 6-10, you still must recalculate `labelsToPrint` once for each record, but not once for each label, so you have improved the program's efficiency.

The modules illustrated in Figures 6-10 and 6-11 do the same thing: print enough labels for every employee to cover 110 percent of production. As you become more proficient at programming, you will recognize many opportunities to perform the same tasks in alternative, more elegant, and more efficient ways.

INITIALIZING A VARIABLE THAT DOES NOT REQUIRE INITIALIZATION

Another common error made by beginning programmers involves initializing a variable that does not require initialization. When declaring variables for the label-making program, you might be tempted to declare `num labelsToPrint = inLastProduction * 1.1`. It seems as though this declaration statement indicates that the value of `labelsToPrint` will always be 110 percent of the `inLastProduction` figure. However, this approach is incorrect for two reasons. First, at the time `labelsToPrint` is declared, the first employee record has not yet been read into memory, so the value of `inLastProduction` is garbage; therefore, the result in `labelsToPrint` after multiplication will also be garbage. Second, even if you read the first `empRecord` into memory before declaring the `labelsToPrint` variable, the mathematical calculation of `labelsToPrint` within the `housekeep()` module would be valid for the first record only. The value of `labelsToPrint` must be recalculated for each employee record in the input file. Therefore, calculation of `labelsToPrint` correctly belongs within the `createLabels()` module, as shown in Figure 6-10.

USING THE FOR STATEMENT

The label-making programs discussed in this chapter each contain two loops. For example, Figures 6-12 and 6-13 show the loop within the mainline program as well as the loop within the `createLabels()` module for a program that produces exactly 100 labels for each employee. (These flowcharts were shown earlier in this chapter.)

FIGURE 6-12: MAINLINE LOGIC FOR LABEL-MAKING PROGRAM

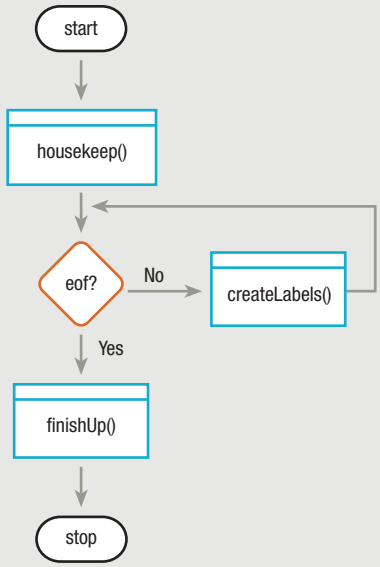
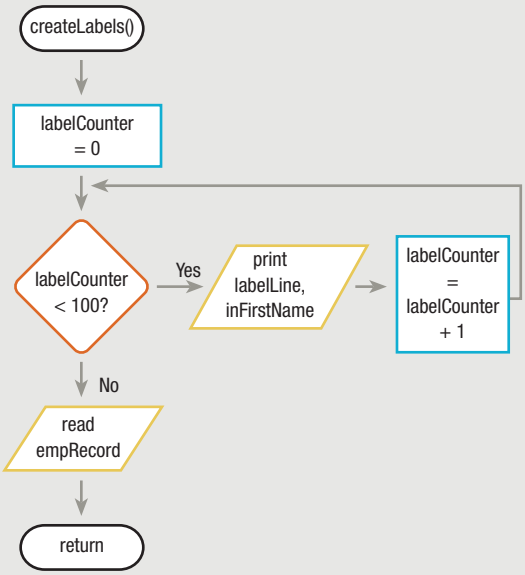


FIGURE 6-13: THE `createLabels()` LOGIC FOR LABEL-MAKING PROGRAM



Entry to the `createLabels()` module in the mainline logic of the label-making program is controlled by the `eof` decision. Within the `createLabels()` method, the loop that produces labels is controlled by the `labelCounter` decision. When you execute the mainline logic, you cannot predict how many times the `createLabels()` module will execute. Depending on the size of the input file (that is, depending on the number of employees who require labels), any number of records might be processed; while the program runs, you don't know what the total number of records finally will be. Until you attempt to read a record and encounter the end of the file, you don't know if more records are going to become available. Of course, not being able to predict the number of input records is valuable—it allows the program to function correctly no matter how many employees exist from week to week or year to year. Because you can't determine ahead of time how many records there might be and, therefore, how many times the loop might execute, the mainline loop in the label-making program is called an **indeterminate**, or **indefinite**, **loop**.

With some loops, you know exactly how many times they will execute. If every employee needs 100 printed labels, then the loop within the `createLabels()` module executes exactly 100 times for each employee. This kind of loop, in which you definitely know the repetition factor, is a **definite loop**.

Every high-level computer programming language contains a `while` **statement** that you can use to code any loop, including indefinite loops (like the mainline loop) and definite loops (like the label-printing loop). You can write statements like the following:

```
while not eof
     perform createLabels()
endwhile
```

and

```
while labelCounter < 100
     print labelLine, inFirstName
     labelCounter = labelCounter + 1
endwhile
```

In addition to the `while` statement, most computer languages also support a `for` statement. You can use the `for` **statement** with definite loops—those for which you know how many times the loop will repeat. The `for` statement provides you with three actions in one compact statement. The `for` statement:

- initializes the loop control variable
- evaluates the loop control variable
- alters the loop control variable (typically by incrementing it)

The `for` statement usually takes the form:

```
for initialValue to finalValue
     do something
endfor
```

For example, to print 100 labels you can write:

```
for labelCounter = 0 to 99
     print labelLine, inFirstName
endfor
```

This `for` statement accomplishes several tasks at once in a compact form:

- The `for` statement initializes `labelCounter` to 0.
- The `for` statement checks `labelCounter` against the limit value 99 and makes sure that `labelCounter` is less than or equal to that value.
- If the evaluation is true, the `for` statement body that prints the label executes.
- After the `for` statement body executes, `labelCounter` increases by 1 and the comparison to the limit value is made again.

TIP As an alternative to using the loop `for labelCounter = 0 to 99`, you can use `for labelCounter = 1 to 100`. You can use any combination of values, as long as there are 100 whole number values between (and including) the two limits.

The `for` statement does not represent a new structure; it simply provides a compact way to write a pretest loop. You are never required to use a `for` statement; the label loop executes correctly using a `while` statement with `labelCounter` as a loop control variable. However, when a loop is based on a loop control variable progressing from a known starting value to a known ending value in equal increments, the `for` statement presents you with a convenient shorthand.

TIP The programmer needs to know neither the starting nor the ending value for the loop control variable; only the program must know those values. For example, you don't know the value of a worker's `inLastProduction`, but when you tell the program to read a record, the program knows. To use this value as a limit value, you can write a `for` statement that begins `for labelCounter = 1 to inLastProduction`.

TIP In most programming languages, you can provide a `for` statement with a step value. A step value is a number you use to increase (or decrease) a loop control variable on each pass through a loop. In most programming languages, the default loop step value is 1. You specify a step value when you want each pass through the loop to change the loop control variable by a value other than 1.

TIP In Java, C++, C#, and other modern languages, the `for` statement is written using the keyword `for` followed by parentheses that contain the increment test, which alters portions of the loop. For example, the following `for` statement could be used in several languages:

```
for(labelCounter = 0; labelCounter < 100; labelCounter = labelCounter + 1)
   print labelLine, inFirstName
```

In this example, the first section within the parentheses initializes the loop control variable, the middle section tests it, and the last section alters it. In languages that use this format, you can use the `for` statement for indefinite loops as well as definite loops.

USING THE DO WHILE AND DO UNTIL LOOPS

When you use either a `while` loop or a `for` statement, the body of the loop may never execute. For example, in the mainline logic in Figure 6-5, the last action in the `housekeep()` module is to read an input record. If the input file contains no records, the result of the `eof` decision is true, and the program executes the `finishUp()` module without ever entering the `createLabels()` module.

Similarly, when you produce labels within the `createLabels()` module shown in Figure 6-10, labels are produced `while labelCounter < labelsToPrint`. Suppose an employee record contains a 0 in the `inLastProduction` field—for example, in the case of a new employee or an employee who was on vacation during the previous week. In such a case, the value of `labelsToPrint` would be 0, and the label-producing body of the loop would never execute. With a `while` loop, you evaluate the loop control variable prior to executing the loop body, and the evaluation might indicate that you can't enter the loop.

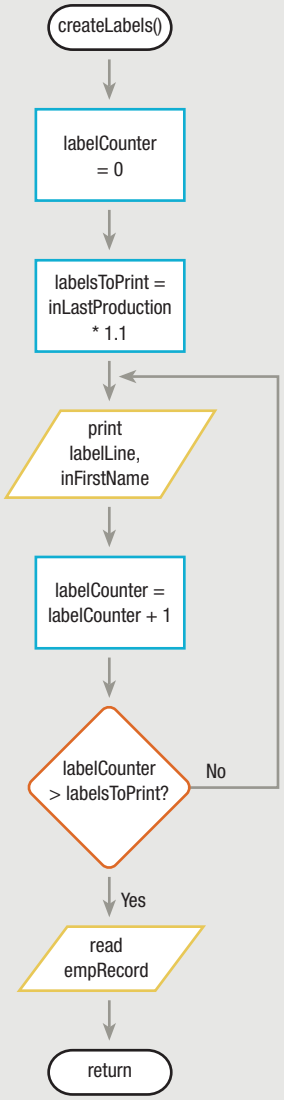
With a `while` loop, the loop body might not execute. When you want to ensure that a loop's body executes at least one time, you can use either a `do while` or a `do until` loop. In both types of loops, the loop control variable is evaluated after the loop body executes, instead of before. Therefore, the body always executes at least one time. Although the loops have similarities, as explained above, they are different in that the `do while` loop continues when the result of the test of the loop control variable is true, but the `do until` loop continues when the result of the test of the loop control variable is false. In other words, the difference between the two loops is simply in how the question at the bottom of the loop is phrased.

TIP You first learned about the `do while` and `do until` loops in Chapter 2. Review Chapter 2 to reinforce your understanding of the differences between a `while` loop and the `do while` and `do until` loops.

TIP Because the question that controls a `while` loop is asked before you enter the loop body, programmers say a `while` loop is a pretest loop. Because the question that controls `do while` and `do until` loops occurs after the loop body executes, programmers say these loops are posttest loops.

For example, suppose you want to produce one label for each employee to wear as identification, before you produce enough labels to cover 110 percent of last week's production. You can write the `do until` loop that appears in Figure 6-14.

FIGURE 6-14: USING A `do until` LOOP TO PRINT ONE IDENTIFICATION LABEL, THEN PRINT ENOUGH TO COVER PRODUCTION REQUIREMENTS



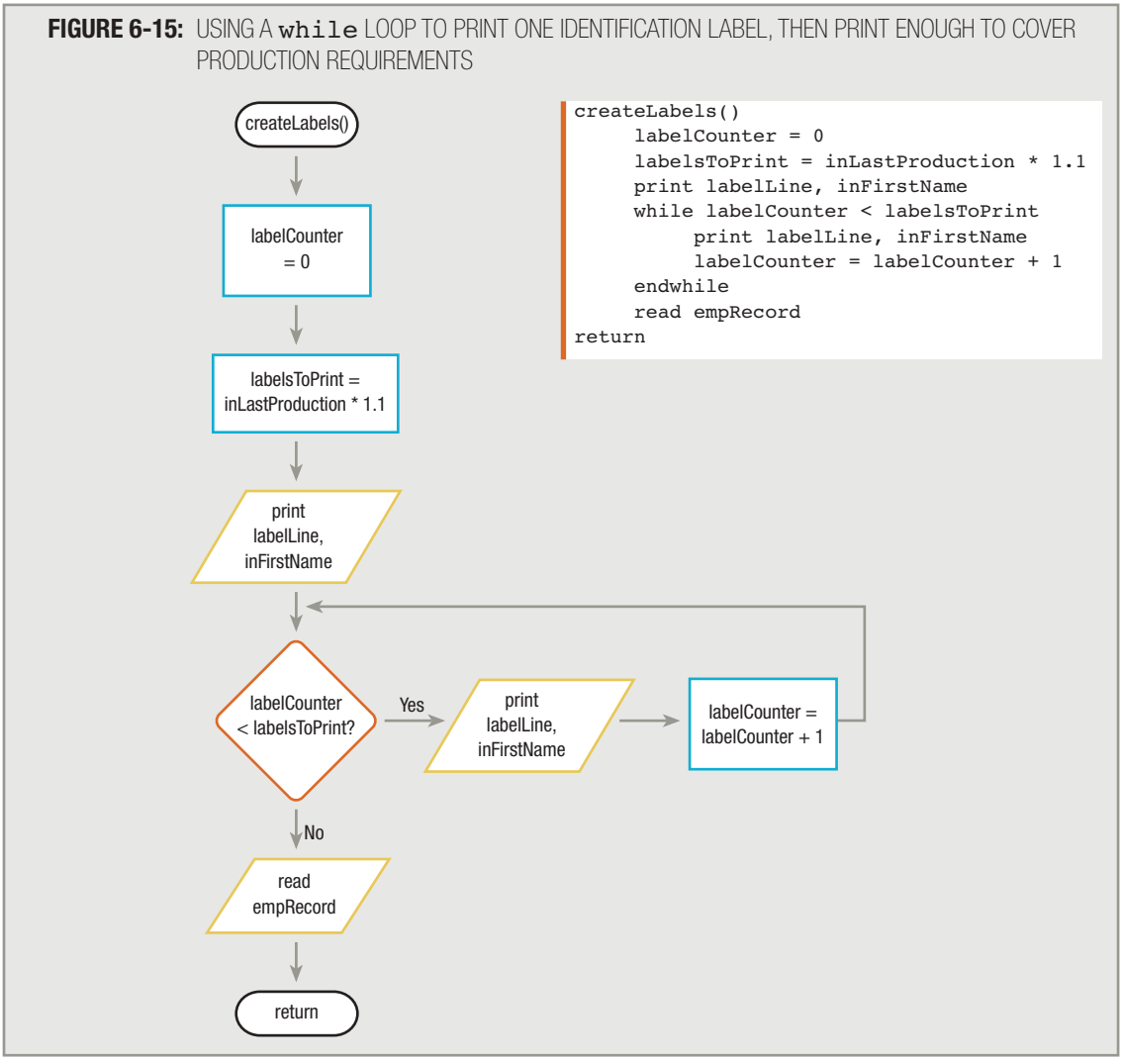

```
createLabels()
      labelCounter = 0
      labelsToPrint = inLastProduction * 1.1
      do
        print labelLine, inFirstName
        labelCounter = labelCounter + 1
      until labelCounter > labelsToPrint
      read empRecord
return
```

In Figure 6-14, the `labelCounter` variable is set to 0 and `labelsToPrint` is calculated. Suppose `labelsToPrint` is computed to be 0. The `do until` loop will be entered, a label will print, 1 will be added to `labelCounter`, and then and only then will `labelCounter` be compared to `labelsToPrint`. Because `labelCounter` is now 1 and `labelsToPrint` is only 0, the loop is exited, having printed a single identification label and no product labels.

As a different example using the logic in Figure 6-14, suppose that for a worker `labelsToPrint` is calculated to be 1. In this case, the loop is entered, a label prints, and 1 is added to `labelCounter`. Now, the value of `labelCounter` is not yet greater than the value of `labelsToPrint`, so the loop repeats, a second label prints, and `labelCounter` is incremented again. This time `labelCounter` (with a value of 2) does exceed `labelsToPrint` (with a value of 1), so the loop ends. This employee gets an identification label as well as one product label.

Of course, you could achieve the same results by printing one label, then entering a `while` loop, as in Figure 6-15. In this example, one label prints before `labelCounter` is compared to `labelsToPrint`. No matter what the value of `labelsToPrint` is, one identification label is produced.

FIGURE 6-15: USING A `while` LOOP TO PRINT ONE IDENTIFICATION LABEL, THEN PRINT ENOUGH TO COVER PRODUCTION REQUIREMENTS



TIP

The logic in Figure 6-15, in which you print one label and then test a value to determine whether you will print more, takes the same form as the mainline logic in most of the programs you have worked with so far. When you read records from a file, you read one record (the priming read) and then test for `eof` before continuing. In effect, the first label printed in Figure 6-15 is a "priming label."

The results of the programs shown in Figures 6-14 and 6-15 are the same. Using either, every employee will receive an identification label and enough labels to cover production. Each module works correctly, and neither is logically superior to the other. There is almost always more than one way to solve the same programming problem. As you learned in Chapter 2, a posttest loop (`do while` or `do until`) can always be replaced by pairing a sequence and a pretest `while` loop. Which method you choose depends on your (or your instructor's or supervisor's) preference.

TIP

There are several additional ways to approach the logic shown in the programs in Figures 6-14 and 6-15. For example, after calculating `labelsToPrint`, you could immediately add 1 to the value. Then, you could use the logic in Figure 6-14, as long as you change the loop-ending question to `labelCounter >= labelsToPrint` (instead of only >). Alternatively, using the logic in Figure 6-15, after adding 1 to `labelsToPrint`, you could remove the lone first label-printing instruction; that way, one identification label would always be printed, even if the last production figure was 0.

RECOGNIZING THE CHARACTERISTICS SHARED BY ALL LOOPS

You can see from Figure 6-15 that you are never required to use posttest loops (either a `do while` loop or a `do until` loop). The same results always can be achieved by performing the loop body steps once before entering a `while` loop. If you follow the logic of either of the loops shown in Figures 6-14 and 6-15, you will discover that when an employee has an `inLastProduction` value of 3, then exactly four labels print. Likewise, when an employee has an `inLastProduction` value of 0, then exactly one label prints. You can accomplish the same results with either type of loop; the posttest `do while` and `do until` loops simply are a convenience when you need a loop's statements to execute at least one time.

TIP

In some languages, the `do until` loop is called a `repeat until` loop.

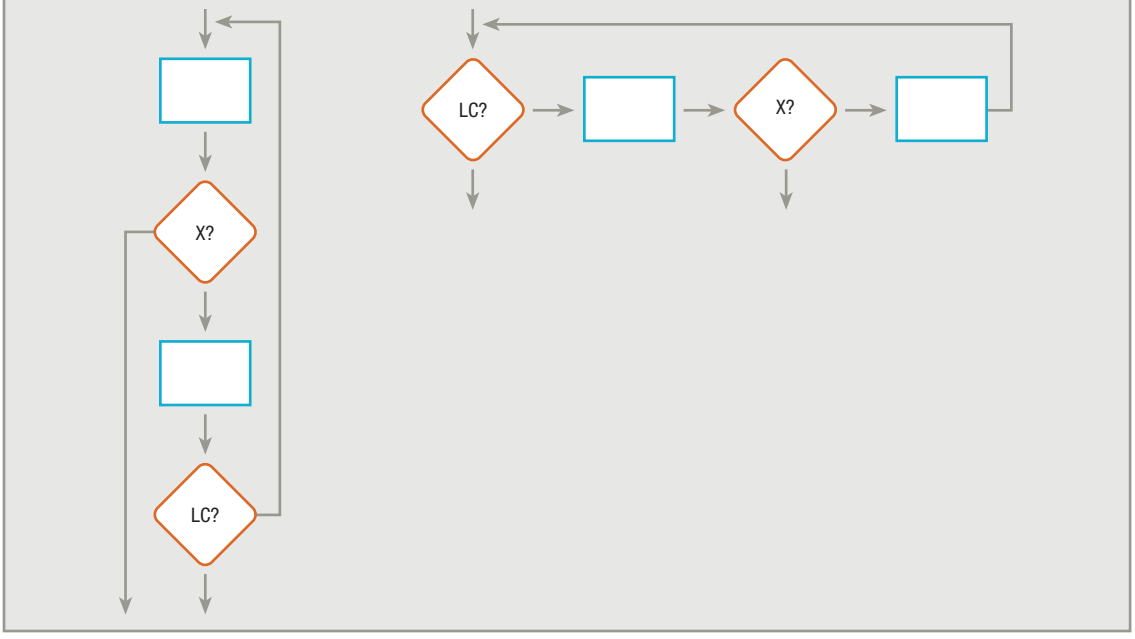
If you can express the logic you want to perform by saying "while a is true, keep doing b," you probably want to use a `while` loop. If what you want to accomplish seems to fit the statement "do a until b is true," you can probably use a `do until` loop. If the statement "do a while b is true" makes more sense, then you might choose to use a `do while` loop.

As you examine Figures 6-14 and 6-15, notice that with the `do until` loop in Figure 6-14, the loop-controlling question is placed at the *end* of the sequence of the steps that repeat. With the `while` loop, the loop-controlling question is placed at the *beginning* of the steps that repeat. All structured loops (whether they are `while` loops, `do while` loops, or `do until` loops) share these characteristics:

- The loop-controlling question provides either entry to or exit from the repeating structure.
- The loop-controlling question provides the *only* entry to or exit from the repeating structure.

You should also notice the difference between *unstructured* loops and the structured `do until` and `while` loops. Figure 6-16 diagrams the outline of two unstructured loops. In each case, the decision labeled X breaks out of the loop prematurely. In each case, the loop control variable (labeled `LC`) does not provide the only entry to or exit from the loop.

FIGURE 6-16: EXAMPLES OF UNSTRUCTURED LOOPS



NESTING LOOPS

Program logic gets more complicated when you must use loops within loops, or **nesting loops**. When one loop appears inside another, the loop that contains the other loop is called the **outer loop**, and the loop that is contained is called the **inner loop**. For example, suppose you work for a company that pays workers twice per month. The company has decided on an incentive plan to provide each employee with a one-fourth of one percent raise for each pay period during the coming year, and it wants a report for each employee like that shown in Figure 6-17. A list will be printed for each employee showing the exact paycheck amounts for each of the next 24 pay periods—two per month for 12 months. A description of the employee input record is shown in Figure 6-18.

FIGURE 6-17: SAMPLE PROJECTED PAYROLL REPORT FOR ONE EMPLOYEE

Projected Payroll for
Roberto Martinez

Month	Check	Amount
1	1	501.25
1	2	502.50
2	1	503.76
2	2	505.02
3	1	506.28

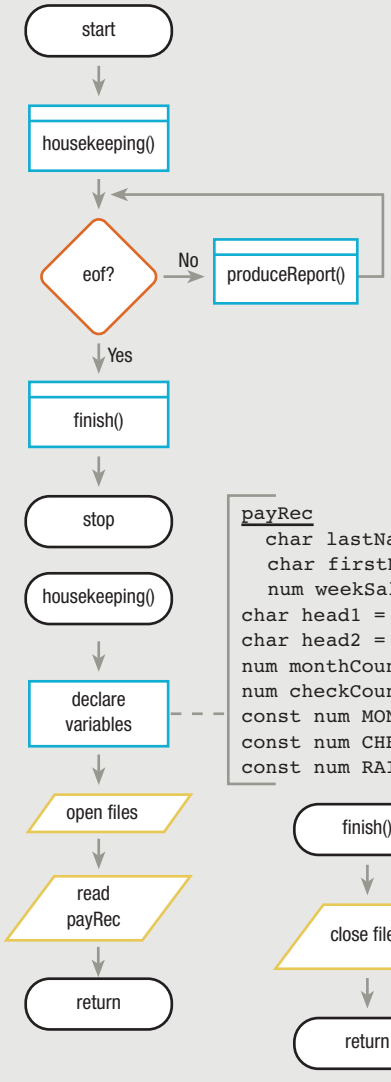
FIGURE 6-18: EMPLOYEE PAYROLL RECORD DATA FILE DESCRIPTION

```
File Name: EMPPAY
FIELD DESCRIPTION       DATA TYPE       COMMENTS
Employee Last Name      Character       12 characters
Employee First Name     Character       8 characters
Weekly salary at        Numeric         2 decimal places
   start of year
```

To produce the Projected Payroll report, you need to maintain two separate counters to control two separate loops. One counter will keep track of the month (1 through 12), and another will keep track of the pay period within the month (1 through 2). When nesting loops, you must maintain individual loop control variables—one for each loop—and alter each at the appropriate time.

Figure 6-19 shows the mainline, `housekeeping()`, and `finish()` logic for the program. These modules are standard. Besides the input file variables and the headers that print for each employee, the list of declared variables includes two counters. One, named `monthCounter`, keeps track of the month that is currently printing. The other, named `checkCounter`, keeps track of which check within the month is currently printing. Three additional declarations hold the number of months in a year (12), the number of checks in a month (2), and the rate of increase (0.0025). Declaring these constants is not required; the program could just use the numeric constants 12, 2, and 0.0025 within its statements, but providing those values with names serves two purposes. First, the program becomes more self-documenting—that is, it describes itself to the reader because the choice of variable names is clear. When other programmers read a program and encounter a number like 2, they might wonder about the meaning. Instead, if the value is named `CHECKS_IN_A_MONTH`, the meaning of the value is much clearer. Second, after the program is in production, the company might choose to change one of the values—for example, by going to an 11-month year, producing more or fewer paychecks in a month, or changing the raise rate. In those cases, the person who modifies the program would not have to search for appropriate spots to make those changes, but would simply redefine the values assigned to the appropriate named constants.

FIGURE 6-19: MAINLINE LOGIC, `housekeeping()`, AND `finish()` MODULES FOR PROJECTED PAYROLL REPORT PROGRAM




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

housekeeping()
     declare variables
     open files
     read payRec
return

finish()
     close files
return
```

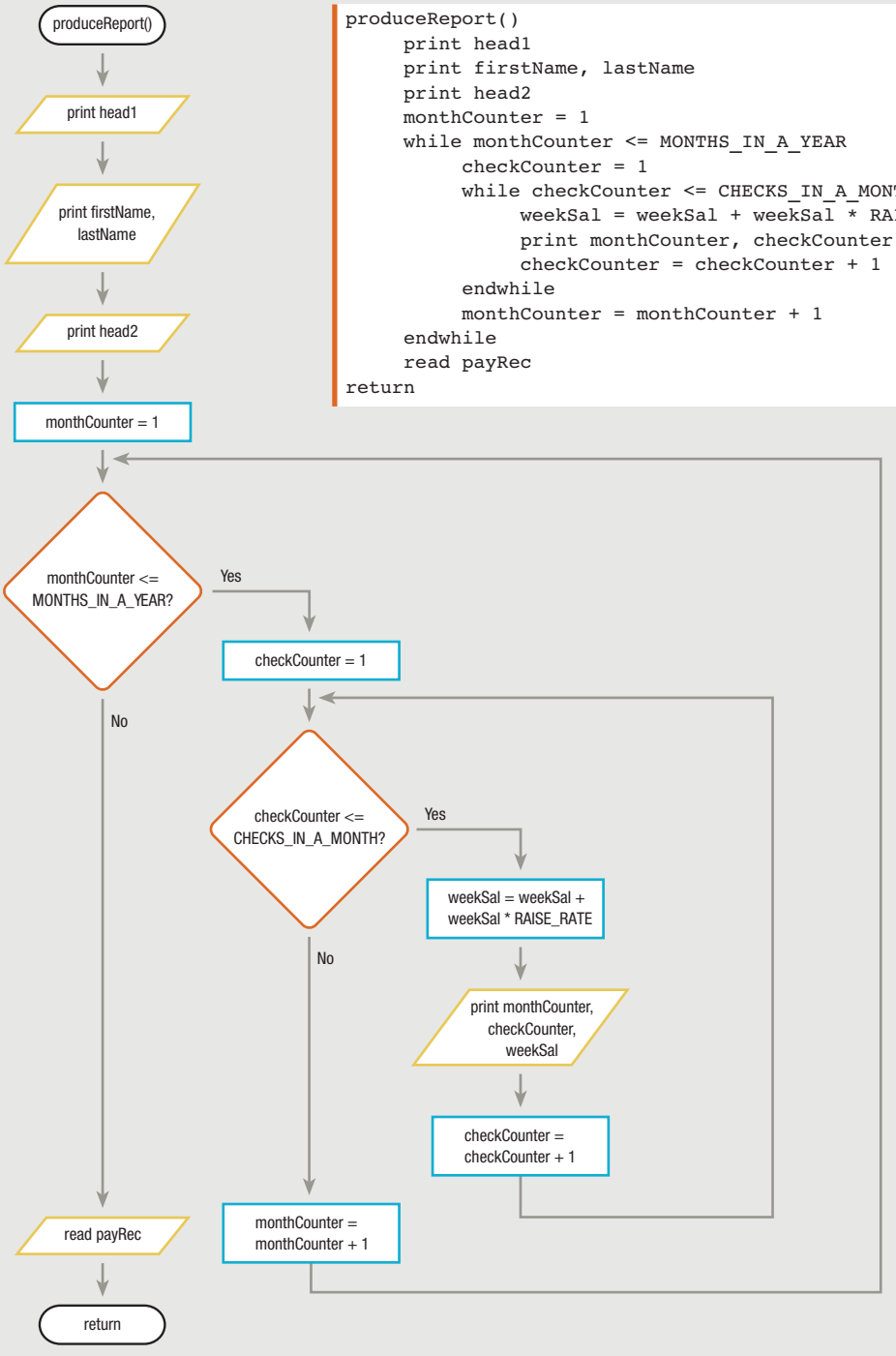
```
payRec
  char lastName
  char firstName
  num weekSal
char head1 = "Projected Payroll for"
char head2 = "Month Check Amount"
num monthCounter
num checkCounter
const num MONTH_IN_A_YEAR = 12
const num CHECKS_IN_A_MONTH = 2
const num RAISE-RATE = 0.0025
```

TIP In Chapter 1 you learned that by convention, many programmers use all uppercase letters when naming constants.

At the end of the `housekeeping()` module in Figure 6-19, the first employee record is read into main memory. Figure 6-20 shows how the record is processed in the `produceReport()` module. The program proceeds as follows:

1. The first heading prints, followed by the employee name and the column headings.
2. The `monthCounter` variable is set to 1; `monthCounter` is the loop control variable for the outer loop, and this step provides it with its initial value.
3. The `monthCounter` variable is compared to the number of months in a year, and because the comparison evaluates as true, the outer loop is entered. Within this loop, the `checkCounter` variable is used as a loop control variable for an inner loop.
4. The `checkCounter` variable is initialized to 1, and then compared to the number of checks in a month. Because this comparison evaluates as true, the inner loop is entered.
5. Within this inner loop, the employee's weekly salary is increased by one-fourth of one percent (the old salary plus 0.0025 of the old salary).
6. The month number (currently 1), check number (also currently 1), and newly calculated salary are printed.
7. The check number is increased (to 2), and the inner loop reaches its end; this causes the logical control to return to the top of the inner loop, where the `while` condition is tested again. Because the check number (2) is still less than or equal to the number of checks in a month, the inner loop is entered again.
8. The pay amount increases, and the month (still 1), check number (2), and new salary are printed.
9. Then, the check number becomes 3. Now, when the loop condition is tested for the third time, the check number is no longer less than or equal to the number of checks in a month, so the inner loop ends.
10. As the last step in the outer loop, `monthCounter` becomes 2.
11. After `monthCounter` increases to 2, control returns to the entry point of the outer loop.
12. The `while` condition is tested, and because 2 is not greater than the number of months in a year, the outer loop is entered for a second time.
13. The `checkCounter` variable is reset to 1 so that it will correctly count two checks for this month.
14. Because the newly reset `checkCounter` is not more than the number of checks in a month, the salary is increased, and the amount prints for month 2, check 1.
15. The `checkCounter` variable increases to 2 and another value is printed for month 2, check 2 before the inner loop ends and `monthCounter` is increased to 3.
16. Then, month 3, check 1 prints, followed by month 3, check 2. The inner loop is evaluated again. The `checkCounter` value is 3, so the evaluation result is false.
17. The `produceReport()` module continues printing two check amounts for each of 12 months before the outer loop is finished, when `monthCounter` eventually exceeds 12. Only then is the next employee record read into memory, and control leaves the `produceReport()` module and returns to the mainline logic, where the end of file is tested. If a new record exists, control returns to the `produceReport()` module for the new employee, for whom headings are printed, and `monthCounter` is set to 1 to start the set of 24 calculations for this employee.

FIGURE 6-20: THE `produceReport()` MODULE FOR THE PROJECTED PAYROLL REPORT PROGRAM




```
produceReport()
     print head1
     print firstName, lastName
     print head2
     monthCounter = 1
     while monthCounter <= MONTHS_IN_A_YEAR
          checkCounter = 1
          while checkCounter <= CHECKS_IN_A_MONTH
               weekSal = weekSal + weekSal * RAISE_RATE
               print monthCounter, checkCounter, weekSal
               checkCounter = checkCounter + 1
          endwhile
          monthCounter = monthCounter + 1
     endwhile
     read payRec
return
```

TIP If you have trouble seeing that the flowchart in Figure 6-20 is structured, consider moving the `checkCounter` loop and its three resulting actions to its own module. Then you should see that the `monthCounter` loop contains a sequence of three steps and that the middle step is a loop.

There is no limit to the number of loop-nesting levels a program can contain. For instance, suppose that in the projected payroll example, the company wanted to provide a slight raise each hour or each day of each pay period in each month for each of several years. No matter how many levels deep the nesting goes, each loop must still contain a loop control variable that is initialized, tested, and altered.

USING A LOOP TO ACCUMULATE TOTALS

Business reports often include totals. The supervisor requesting a list of employees who participate in the company dental plan is often as much interested in *how many* such employees there are as in *who* they are. When you receive your telephone bill at the end of the month, you are usually more interested in the total than in the charges for the individual calls. Some business reports list no individual detail records, just totals or other overall statistics such as averages. Such reports are called **summary reports**. Many business reports list both the details of individual records and totals at the end.

For example, a real estate broker might maintain a file of company real estate listings. Each record in the file contains the street address and the asking price of a property for sale. The broker wants a listing of all the properties for sale; she also wants a total value for all the company's listings. A typical report appears in Figure 6-21.

FIGURE 6-21: TYPICAL REAL ESTATE REPORT

PROPERTIES FOR SALE

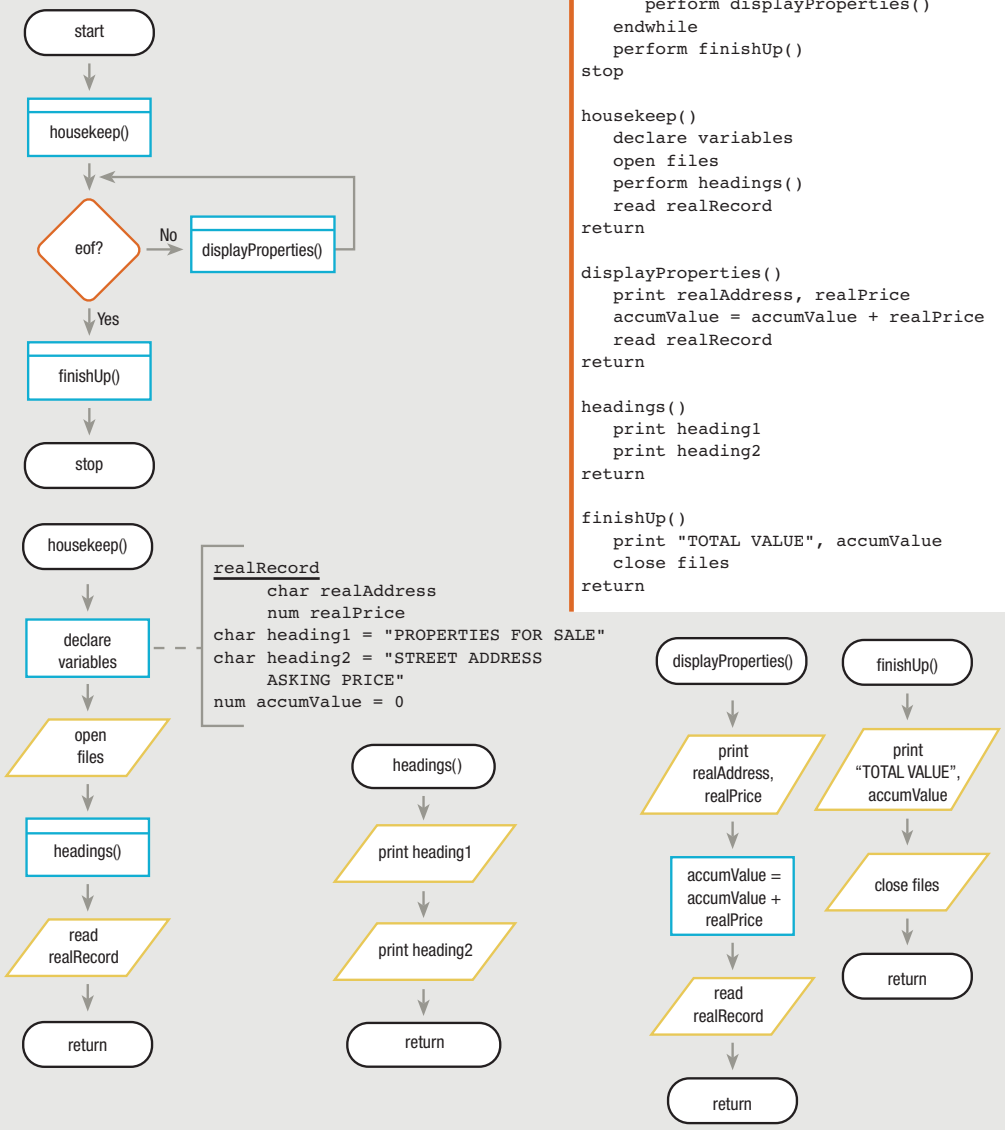
STREET ADDRESS	ASKING PRICE
12 Carpenter Road	218,000
312 Howard Street	119,900
416 Mockingbird Lane	349,900
58 Flowerwood Path	249,900
5914 Wisteria Lane	499,999
TOTAL VALUE	1,437,699

When you read a real estate listing record, besides printing it you must add its value to an accumulator. An **accumulator** is a variable that you use to gather, or accumulate, values. An accumulator is very similar to a counter. The difference lies in the value that you add to the variable; usually, you add just 1 to a counter, whereas you add some other value to an accumulator. If the real estate broker wants to know how many listings the company holds, you count them. When she wants to know total real estate value, you accumulate it.

In order to accumulate total real estate prices, you declare a numeric variable at the beginning of the program, as shown in the `housekeep()` module in Figure 6-22. You must initialize the accumulator, `accumValue`, to 0. In

Chapter 4, you learned that when using most programming languages, declared variables do not automatically assume any particular value; the unknown value is called garbage. When you read the first real estate record, you will add its value to the accumulator. If the accumulator contains garbage, the addition will not work. Some programming languages issue an error message if you don't initialize a variable you use for accumulating; others let you accumulate, but the results are worthless because you start with garbage.

FIGURE 6-22: THE REAL ESTATE PROGRAM



If you name the input record fields `realAddress` and `realPrice`, then the `displayProperties()` module of the real estate listing program can be written as shown in Figure 6-22. For each real estate record, you print it and add its value to the accumulator `accumValue`. Then you can read the next record.

After the program reaches the end of the file, the accumulator will hold the grand total of all the real estate values. When you reach the end of the file, the `finishUp()` module executes, and it is within the `finishUp()` module that you print the accumulated value, `accumValue`. After printing the total, you can close both the input and the output files and return to the mainline logic, where the program ends.

New programmers often want to reset the `accumValue` to 0 after printing it. Although you *can* take this step without harming the execution of the program, it does not serve any useful purpose. You cannot set `accumValue` to 0 in anticipation of having it ready for the next program, or even for the next time you execute this program. Program variables exist only for the life of the program, and even if a future program happens to contain a variable named `accumValue`, the variable will not necessarily occupy the same memory location as this one. Even if you run the program a second time, the variables might occupy physical memory locations different from those they occupied during the first run. At the beginning of the program, it is the programmer's responsibility to initialize all variables that must start with a specific value. There is no benefit to changing a variable's value when it will never be used again during the current execution of the program.

TIP It is especially important to avoid changing the value of a variable unnecessarily when the change occurs within a loop. One extra, unnecessary statement in a loop that executes hundreds of thousands of times can significantly slow a program's performance speed.

CHAPTER SUMMARY

- When you use a loop within a computer program, you can write one set of instructions that operates on multiple, separate sets of data.
- Three steps must occur in every loop: You must initialize a loop control variable, compare the variable to some value that controls whether the loop continues or stops, and alter the variable that controls the loop.
- A counter is a numeric variable you use to count the number of times an event has occurred. You can count occurrences by incrementing or decrementing a variable.
- You can use a variable sentinel value to control a loop.
- Sometimes it is convenient to reduce, or decrement, a loop control variable on every cycle through a loop.
- Mistakes that programmers often make with loops include neglecting to initialize the loop control variable and neglecting to alter the loop control variable. Other mistakes include using the wrong comparison with the loop control variable, including statements inside the loop that belong outside the loop, and initializing a variable that does not require initialization.
- Most computer languages support a **`for`** statement that you can use with definite loops when you know how many times a loop will repeat. The **`for`** statement uses a loop control variable that it automatically initializes, evaluates, and increments.
- When you want to ensure that a loop's body executes at least one time, you can use a **`do while`** loop or a **`do until`** loop, in which the loop control variable is evaluated after the loop body executes.
- All structured loops share these characteristics: The loop-controlling question provides either entry to or exit from the repeating structure, and the loop-controlling question provides the *only* entry to or exit from the repeating structure.
- When you must use loops within loops, you are using nested loops. When you create nested loops, you must maintain two individual loop control variables and alter each at the appropriate time.
- Business reports often include totals. Summary reports list no detail records—only totals. An accumulator is a variable that you use to gather or accumulate values.

KEY TERMS

A **loop** is a structure that repeats actions while some condition continues.

A **main loop** is a basic set of instructions that is repeated for every record.

A **loop control variable** is a variable that determines whether a loop will continue.

A **sentinel value** is a limit or ending value.

A **loop body** is the set of statements that executes within a loop.

A **counter** is any numeric variable you use to count the number of times an event has occurred.

Adding to a variable (often, adding one) is called **incrementing** the variable.

Decreasing a variable (often by one) is called **decrementing** the variable.

A loop that never stops executing is called an **infinite loop**.

An **indeterminate**, or **indefinite**, **loop** is one for which you cannot predetermine the number of executions.

A loop for which you definitely know the repetition factor is a **definite loop**.

A **`while` statement** can be used to code any loop.

A **`for` statement** frequently is used to code a definite loop. Most often, it contains a loop control variable that it initializes, evaluates, and increments.

Nesting loops are loops within loops.

When one loop appears inside another, the loop that contains the other loop is called the **outer loop**, and the loop that is contained is called the **inner loop**.

A **summary report** lists only totals and other statistics, without individual detail records.

An **accumulator** is a variable that you use to gather, or accumulate, values.

REVIEW QUESTIONS

1. **The structure that allows you to write one set of instructions that operates on multiple, separate sets of data is the _____.**
   a. sequence
   b. selection
   c. loop
   d. case

2. **Which of the following is not a step that must occur in every loop?**
   a. Initialize a loop control variable.
   b. Compare the loop control value to a sentinel.
   c. Set the loop control value equal to a sentinel.
   d. Alter the loop control variable.

3. The statements executed within a loop are known collectively as the _____.

a. sentinels
b. loop controls
c. sequences
d. loop body

4. A counter keeps track of _____.

a. the number of times an event has occurred
b. the number of modules in a program
c. the number of loop structures within a program
d. a total that prints at the end of a summary report

5. Adding 1 to a variable is also called _____.

a. digesting
b. incrementing
c. decrementing
d. resetting

6. In the following pseudocode, what is printed?

```
a = 1
b = 2
c = 5
while a < c
     a = a + 1
     b = b + c
endwhile
print a, b, c
```

a. 1 2 5
b. 5 22 5
c. 5 6 5
d. 6 22 9

7. In the following pseudocode, what is printed?

```
d = 4
e = 6
f = 7
while d > f
     d = d + 1
     e = e - 1
endwhile
print d, e, f
```

a. 7 3 7
b. 8 2 8
c. 4 6 7
d. 5 5 7

8. When you decrement a variable, most frequently you _____.

a. set it to 0
b. reduce it by one-tenth
c. subtract 1 from it
d. remove it from a program

9. In the following pseudocode, what is printed?

```
g = 4
h = 6
while g < h
     g = g + 1
endwhile
print g, h
```

a. nothing
b. 4 6
c. 5 6
d. 6 6

10. Most programmers use a `for` statement _____.

a. for every loop they write
b. as a compact version of the `while` statement
c. when they do not know the exact number of times a loop will repeat
d. when a loop will not repeat

11. Unlike a `while` loop, you use a `do until` loop when _____.

a. you can predict the exact number of loop repetitions
b. the loop body might never execute
c. the loop body must execute exactly one time
d. the loop body must execute at least one time

12. Which of the following is a characteristic shared by all loops—`while`, `do while`, and `do until` loops?

a. They all have one entry and one exit.
b. They all have a body that executes at least once.
c. They all compare a loop control variable at the top of the loop.
d. All of these are true.

13. A comparison with a loop control variable provides _____.

a. the only entry to a `while` loop
b. the only exit from a `do until` loop
c. both of the above
d. none of the above

14. When two loops are nested, the loop that is contained by the other is the _____ loop.

a. inner
b. outer
c. unstructured
d. captive

15. In the following pseudocode, how many times is "Hello" printed?

```
j = 2
k = 5
m = 6
n = 9
while j < k
     while m < n
          print "Hello"
          m = m + 1
     endwhile
     j = j + 1
endwhile
```

a. zero
b. three
c. six
d. nine

16. In the following pseudocode, how many times is "Hello" printed?

```
j = 2
k = 5
n = 9
while j < k
     m = 6
     while m < n
          print "Hello"
          m = m + 1
     endwhile
     j = j + 1
endwhile
```

a. zero
b. three
c. six
d. nine

17. In the following pseudocode, how many times is "Hello" printed?

```
p = 2
q = 4
while p < q
     print "Hello"
     r = 1
     while r < q
          print "Hello"
          r = r + 1
     endwhile
     p = p + 1
endwhile
```

a. zero
b. four
c. six
d. eight

18. A report that lists no details about individual records, but totals only, is a(n) _____ report.

a. accumulator
b. final
c. summary
d. detailless

19. Typically, the value added to a counter variable is _____.

a. 0
b. 1
c. 10
d. 100

20. Typically, the value added to an accumulator variable is _____.

a. 0
b. 1
c. at least 1000
d. Any value might be added to an accumulator variable.

FIND THE BUGS

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

1. **This method is supposed to print every fifth year starting with 2005; that is, 2005, 2010, 2015, and so on, for 30 years.**

```
printEveryFifthYear()
   const num YEAR = 2005
   num factor = 5
   const num END_YEAR = 2035
   while year > END_YEAR
      print year
      year = year + 1
   endwhile
return
```

2. **A standard mortgage is paid monthly over 30 years. This method is intended to print 360 payment coupons for a new borrower. Each coupon lists the month number, year number, and a friendly reminder.**

```
printCoupons()
   const num MONTHS = 12
   const num YEARS = 30
   num monthCounter
   num yearCounter
   while yearCounter <= YEARS
      while monthCounter <= 12
         print month, year, "Remember to send your payment by the 10th"
         yearCounter = yearCounter + 1
      endwhile
   endwhile
return
```

3. **This application is intended to print estimated monthly payment amounts for customers of the EZ Credit Loan Company. The application reads customer records, each containing an account number, name and address, requested original loan amount, term in months, and annual interest rate. The interest rate per month is calculated by dividing the annual interest rate by 12. The customer's total payback amount is calculated by charging the monthly interest rate on the original balance every month for the term of the loan. The customer's monthly payment is then calculated by dividing the total payback amount by the number of months in the loan. The application produces a notice containing the customer's name, address, and estimated monthly payment amount.**

```
start
   perform getReady()
   while not eof
      perform produceEstimate()
   perform ending()
stop
startUp()
   declare variables
   custRecord
      num acctNumber
      char name
      char address
      num originalLoanAmount
      num termInMonths
      num annualRate
   const num MONTHS_IN_YEAR = 12
   const num totalPayback
   num monthlyRate
   num count
   open files
   read custRecord
return

produceEstimate()
   count = 1
   monthlyRate = annualRate / monthsInYear
   while count = termInMonths
      totalPayback = totalPayback + monthlyRate * originalLoanAmount
      count = count + 1
   endwhile
   monthlyPayment = totalPayback / MONTHS_IN_YEAR
   print "Loan Payment Estimate for:"
   print name
   print address
   print "$", monthPayment
return

ending()
   close files
return
```

EXERCISES

1. **Design the logic for a module that would print every number from 1 through 10.**
   a. Draw the flowchart.
   b. Design the pseudocode.

2. **Design the logic for a module that would print every number from 1 through 10 along with its square and cube.**
   a. Draw the flowchart.
   b. Design the pseudocode.

3. **Design a program that reads credit card account records and prints payoff schedules for customers. Input records contain an account number, customer name, and balance due. For each customer, print the account number and name; then print the customer's projected balance each month for the next 10 months. Assume that there is no finance charge on this account, that the customer makes no new purchases, and that the customer pays off the balance with equal monthly payments, which are 10 percent of the original bill.**
   a. Design the output for this program; create either sample output or a print chart.
   b. Design the hierarchy chart for this program.
   c. Design the flowchart for this program.
   d. Write pseudocode for this program.

4. **Design a program that reads credit card account records and prints payoff schedules for customers. Input records contain an account number, customer name, and balance due. For each customer, print the account number and name; then print the customer's payment amount and new balance each month until the card is paid off. Assume that when the balance reaches $10 or less, the customer can pay off the account. At the beginning of every month, 1.5 percent interest is added to the balance, and then the customer makes a payment equal to 5 percent of the current balance. Assume the customer makes no new purchases.**
   a. Design the output for this program; create either sample output or a print chart.
   b. Design the hierarchy chart for this program.
   c. Design the flowchart for this program.
   d. Write pseudocode for this program.

5. **Assume you have a bank account that compounds interest on a yearly basis. In other words, if you deposit $100 for two years at 4 percent interest, at the end of one year you will have $104. At the end of two years, you will have the $104 plus 4 percent of that, or $108.16. Create the logic for a program that would (1) read in records containing a deposit amount, a term in years, and an interest rate, and (2) for each record, print the running total balance for each year of the term.**
   a. Design the output for this program; create either sample output or a print chart.
   b. Design the hierarchy chart for this program.
   c. Design the flowchart for this program.
   d. Write pseudocode for this program.

6. A school maintains class records in the following format:

```
CLASS FILE DESCRIPTION
File name: CLASS
FIELD DESCRIPTION     DATA TYPE       EXAMPLE
Class Code            Character       CIS111
Section No.           Numeric         101
Teacher Name          Character       Gable
Enrollment            Numeric         24
Room                  Character       A213
```

There is one record for each class section offered in the college. Design the program that would print as many stickers as a class needs to provide one for each enrolled student, plus one for the teacher. Each sticker would leave a blank for the student's (or teacher's) name, like this:

The border is preprinted, but you must design the program to print all the text you see on the sticker. (You do not need to worry about the differing font sizes of the sticker text. You do not need to design a print chart or sample output—the image of the sticker serves as a print chart.)

a. Design the hierarchy chart for this program.
b. Design the flowchart for this program.
c. Write pseudocode for this program.

7. A mail-order company often sends multiple packages per order. For each customer order, print enough mailing labels to use on each of the separate boxes that will be mailed. The mailing labels contain the customer's complete name and address, along with a box number in the form "Box 9 of 9". For example, an order that requires three boxes produces three labels: Box 1 of 3, Box 2 of 3, and Box 3 of 3. The file description is as follows:

```
SHIPPING FILE DESCRIPTION
File name: ORDERS
FIELD DESCRIPTION     DATA TYPE     EXAMPLE
Title                 Character     Ms
First Name            Character     Kathy
Last Name             Character     Lewis
Street                Character     847 Pine
```

```
City                Character     Aurora
State               Character     IL
Boxes               Numeric       3
Balance Due         Numeric       129.95
```

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

8. A secondhand store is having a seven-day sale during which the price of any unsold item drops 10 percent each day. The inventory file includes an item number, description, and original price on day one. For example, an item that costs $10.00 on the first day costs 10 percent less, or $9.00, on the second day. On the third day, the same item is 10 percent less than $9.00, or $8.10. Produce a report that shows the price of the item on each day, one through seven.

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

9. The state of Florida maintains a census file in which each record contains the name of a county, the current population, and a number representing the rate at which the population is increasing per year. The governor wants a report listing each county and the number of years it will take for the population of the county to double, assuming the present rate of growth remains constant.

```
CENSUS FILE DESCRIPTION
File name: CENSUS
FIELD DESCRIPTION        DATA TYPE          EXAMPLE
County Name              Character          Dade
Current Population       Numeric            525000
Rate of Growth           Numeric            0.07
```

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

10. A Human Resources Department wants a report that shows its employees the benefits of saving for retirement. Produce a report that shows 12 predicted retirement account values for each employee—the values if the employee saves 5, 10, or 15 percent of his or her annual salary for 10, 20, 30, or 40 years. The department maintains a file in which each record contains the name of an employee and the employee's current annual salary. Assume that savings grow at a rate of 8 percent per year.

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

11. Randy's Recreational Vehicles pays its salespeople once every three months. Salespeople receive one-quarter of their annual base salary plus 7 percent of all sales made in the last three-month period. Randy creates an input file with four records for each salesperson. The first of the four records contains the salesperson's name and annual base salary, while each of the three records that follow contains the name of a month and the monthly sales figure. For example, the first eight records in the file might contain the following data:

Kimball	**20000**
April	**30000**
May	**40000**
June	**60000**
Johnson	**15000**
April	**65000**
May	**78000**
June	**135500**

Because the two types of records contain data in the same format—a character field followed by a numeric field—you can define one input record format containing two variables that you use with either type of record. Design the logic for the program that reads a salesperson's record, and if not at `eof`, reads the next three records in a loop, accumulating sales and computing commissions. For each salesperson, print the quarterly base salary, the three commission amounts, and the total salary, which is the quarterly base plus the three commission amounts.

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

12. Mr. Furly owns 20 apartment buildings. Each building contains 15 units that he rents for $800 per month each. Design the logic for the program that would print 12 payment coupons for each of the 15 apartments in each of the 20 buildings. Each coupon should contain the building number (1 through 20), the apartment number (1 through 15), the month (1 through 12), and the amount of rent due.

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

13. **Mr. Furly owns 20 apartment buildings. Each building contains 15 units that he rents. The usual monthly rent for apartments numbered 1 through 9 in each building is $700; the monthly rent is $850 for apartments numbered 10 through 15. The usual rent is due every month except July and December; in those months Mr. Furly gives his renters a 50 percent credit, so they owe only half the usual amount. Design the logic for the program that would print 12 payment coupons for each of the 15 apartments in each of the 20 buildings. Each coupon should contain the building number (1 through 20), the apartment number (1 through 15), the month (1 through 12), and the amount of rent due.**
    a. Design the output for this program; create either sample output or a print chart.
    b. Design the hierarchy chart for this program.
    c. Design the flowchart for this program.
    d. Write pseudocode for this program.

DETECTIVE WORK

1. **What company's address is at One Infinite Loop, Cupertino, California?**
2. **What are fractals? How do they use loops? Find some examples of fractal art on the Web.**

UP FOR DISCUSSION

1. **If programs could only make decisions or loops, but not both, which structure would you prefer to retain?**
2. **Suppose you wrote a program that you suspect is in an infinite loop because it just keeps running for several minutes with no output and without ending. What would you add to your program to help you discover the origin of the problem?**
3. **Suppose you know that every employee in your organization has a seven-digit ID number used for logging on to the computer system to retrieve sensitive information about their own customers. A loop would be useful to guess every combination of seven digits in an ID. Are there any circumstances in which you should try to guess another employee's ID number?**