9 ADVANCED ARRAY MANIPULATION

After studying Chapter 9, you should be able to:

□ Describe the need for sorting data

□ Swap two values in computer memory

□ Use a bubble sort

□ Use an insertion sort

□ Use a selection sort

□ Use indexed files

□ Use a linked list

□ Use multidimensional arrays

UNDERSTANDING THE NEED FOR SORTING RECORDS

When you store data records, they exist in some sort of order; that is, one record is first, another second, and so on. When records are in **sequential order**, they are arranged one after another on the basis of the value in some field. Examples of records in sequential order include employee records stored in numeric order by Social Security number or department number, or in alphabetic order by last name or department name. Even if the records are stored in a random order—for example, the order in which a data-entry clerk felt like entering them—they still are in *some* order, although probably not the order desired for processing or viewing. When this is the case, the data records need to be **sorted**, or placed in order, based on the contents of one or more fields. When you sort data, you can sort either in **ascending order**, arranging records from lowest to highest value within a field, or **descending order**, arranging records from highest to lowest value. Here are some examples of occasions when you would need to sort records:

- A college stores students' records in ascending order by student ID number, but the registrar wants to view the data in descending order by credit hours earned so he can contact students who are close to graduation.
- A department store maintains customer records in ascending order by customer number, but at the end of a billing period, the credit manager wants to contact customers whose balances are 90 or more days overdue. The manager wants to list these overdue customers in descending order by the amount owed, so the customers maintaining the biggest debt can be contacted first.
- A sales manager keeps records for her salespeople in alphabetical order by last name, but needs to list the annual sales figure for each salesperson so she can determine the median annual sale amount. The **median** value in a list is the value of the middle item when the values are listed in order; it is not the same as the arithmetic average, or **mean**.

TIP

To help you understand the difference between median and mean, consider the following five values: 0, 7, 10, 11, 12. The median value is the middle position's value (when the values are listed in numerical order), or 10. The mean, however, is the sum (40) divided by the number of values (5), which evaluates to 8. The median is used as a statistic in many cases because it represents a more typical case—half the values are below it and half are above it. Unlike the median, the mean is skewed by a few very high or low values.

TIP

Sorting is usually reserved for a relatively small number of data items. If thousands of customer records are stored, and they frequently need to be accessed in order based on different fields (alphabetical order by customer name one day, zip code order the next), the records would probably not be sorted at all, but would be indexed or linked. You learn about indexing and linking later in this chapter.

When computers sort data, they always use numeric values when making comparisons between values. This is clear when you sort records by fields such as a numeric customer ID or balance due. However, even alphabetic sorts are numeric, because everything that is stored in a computer is stored as a number using a series of 0s and 1s. In every popular computer coding scheme, "B" is numerically one greater than "A", and "y" is numerically one less than "z". Unfortunately, whether "A" is represented by a number that is greater or smaller than the number representing "a" depends on your system. Therefore, to obtain the most useful and accurate list of alphabetically sorted records, either the data-entry personnel should be consistent in the use of capitalization, or the programmer should convert all the data to consistent capitalization.

TIP □□□□

Because "A" is always less than "B", alphabetic sorts are always considered ascending sorts. The most popular coding schemes include ASCII, Unicode, and EBCDIC. Each is a code in which a number represents every computer character. Appendix B contains additional information about these codes.

TIP □□□□

It's possible that as a professional programmer, you will never have to write a program that sorts records, because organizations can purchase prewritten, or "canned," sorting programs. Additionally, many popular language compilers come with built-in methods that can sort data for you. However, it is beneficial to understand the sorting process so that you can write a special-purpose sort when needed. Understanding sorting also improves your array-manipulating skills.

UNDERSTANDING HOW TO SWAP TWO VALUES

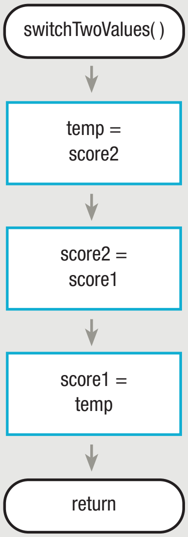
Many sorting techniques have been developed. A concept that is central to most sorting techniques involves **swapping** two values. When you swap the values stored in two variables, you reverse their positions; you set the first variable equal to the value of the second, and the second variable equal to the value of the first. However, there is a trick to reversing any two values. Assume you have declared two variables as follows:

```
num score1 = 90
num score2 = 85
```

You want to swap the values so that `score1` is 85 and `score2` is 90. If you first assign `score1` to `score2` using a statement such as `score2 = score1`, both `score1` and `score2` hold 90 and the value 85 is lost. Similarly, if you first assign `score2` to `score1` using a statement such as `score1 = score2`, both variables hold 85 and the value 90 is lost.

The solution to swapping the values lies in creating a temporary variable to hold one of the scores; then, you can accomplish the swap as shown in Figure 9-1. First, the value in `score2`, 85, is assigned to a temporary holding variable, named `temp`. Then, the `score1` value, 90, is assigned to `score2`. At this point, both `score1` and `score2` hold 90. Then, the 85 in `temp` is assigned to `score1`. Therefore, after the swap process, `score1` holds 85 and `score2` holds 90.

FIGURE 9-1: A MODULE THAT SWAPS TWO VALUES



```
switchTwoValues()
     temp = score2
     score2 = score1
     score1 = temp
return
```

TIP In Figure 9-1, you can accomplish identical results by assigning `score1` to `temp`, assigning `score2` to `score1`, and finally assigning `temp` to `score2`.

USING A BUBBLE SORT

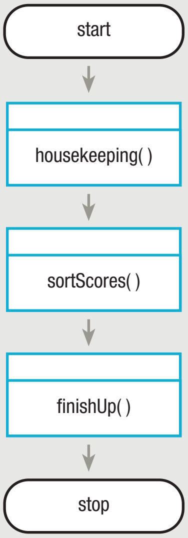
One of the simplest sorting techniques to understand is a bubble sort. You can use a bubble sort to arrange records in either ascending or descending order. In a **bubble sort**, items in a list are compared with each other in pairs, and when an item is out of order, it swaps values with the item below it. With an ascending bubble sort, after each adjacent pair of items in a list has been compared once, the largest item in the list will have "sunk" to the bottom. After many passes through the list, the smallest items rise to the top like bubbles in a carbonated drink.

TIP A bubble sort is sometimes called a **sinking sort**.

Assume that five student test scores are stored in a file and you want to sort them in ascending order for printing. To begin, you can define three modules in the mainline logic, as shown in Figure 9-2: `housekeeping()`, `sortScores()`, and `finishUp()`.

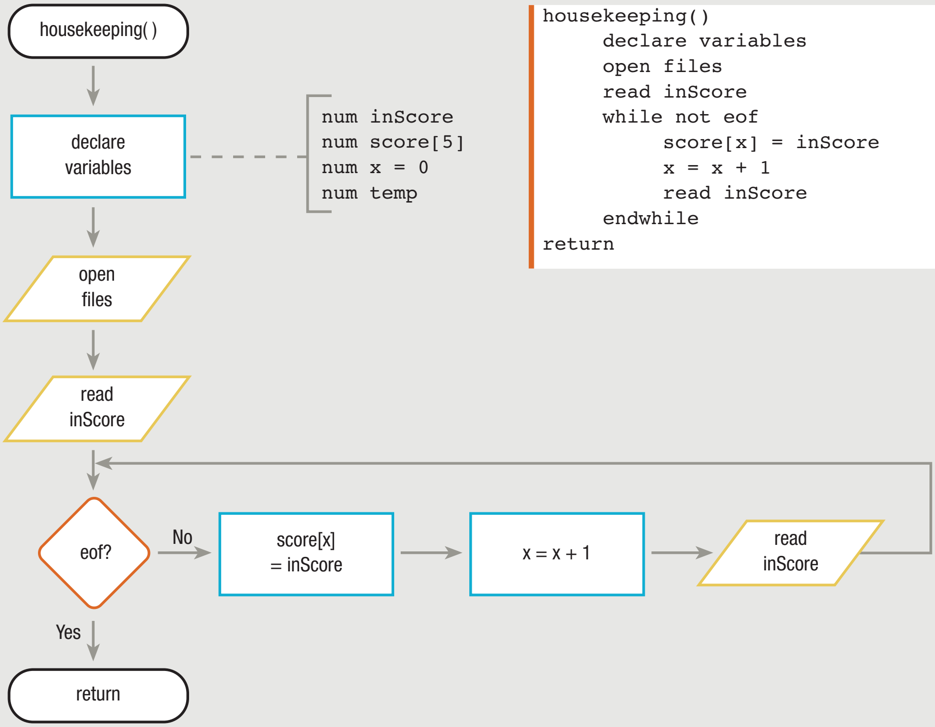
The `housekeeping()` module of this program defines a variable name for each individual score in the input file `(inScore)` and sets up an array of five elements `(score)` in which to store the five scores. The entire file is then read into memory, one score at a time, and each score is stored in one element of the array. See Figure 9-3.

FIGURE 9-2: MAINLINE LOGIC FOR THE SCORE-SORTING PROGRAM



```
start
     perform housekeeping()
     perform sortScores()
     perform finishUp()
stop
```

FIGURE 9-3: THE `housekeeping()` MODULE FOR THE SCORE-SORTING PROGRAM



```
housekeeping()
     declare variables
     open files
     read inScore
     while not eof
          score[x] = inScore
          x = x + 1
          read inScore
     endwhile
return
```

When the program logic leaves the `housekeeping()` module and enters the `sortScores()` module, five scores have been placed in the array. For example, assume they are:

```
score[0] = 90
score[1] = 85
score[2] = 65
score[3] = 95
score[4] = 75
```

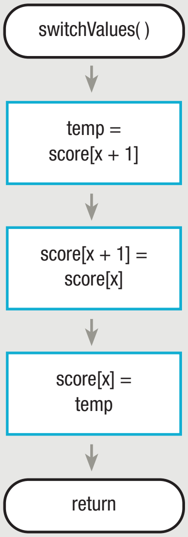
To begin sorting this list of scores, you compare the first two scores. If they are out of order, you reverse their positions, or swap their values. That is, if `score[0]` is more than `score[1]`, then `score[0]` assumes the value 85 and `score[1]` takes on the value 90. After this swap, the scores are in slightly better order than they were originally.

You could reverse the values of `score[0]` and `score[1]` using the following code:

```
switchValues()
     temp = score[1]
     score[1] = score[0]
     score[0] = temp
return
```

However, this code segment's usefulness is limited because it switches only the first two elements of the `score` array. If you use hard values such as 0 and 1 as subscripts, then you must write additional statements to swap the values in positions 1 and 2, 2 and 3, and 3 and 4. A more universal `switchValues()` module is shown in Figure 9-4. This module switches *any* two adjacent elements in the `score` array when the variable `x` represents the position of the first of the two elements, and the value `x + 1` represents the subsequent position.

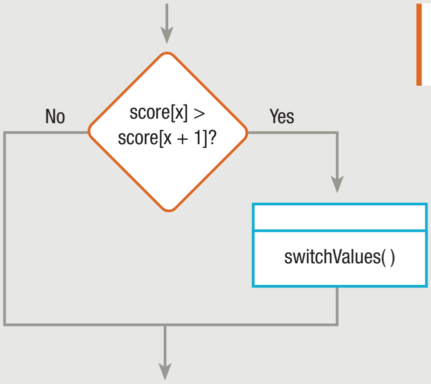
FIGURE 9-4: THE `switchValues()` MODULE THAT SWAPS ANY TWO ADJACENT VALUES IN AN ARRAY



```
switchValues()
     temp = score[x + 1]
     score[x + 1] = score[x]
     score[x] = temp
return
```

For an ascending sort, you need to perform the `switchValues()` module whenever any given element `x` of the `score` array has a value greater than the next element, `x + 1`, of the `score` array. For any `x`, if the xth element is not greater than the element at position `x + 1`, the switch should not take place. For example, when `score[x]` is 90 and `score[x + 1]` is 85, a swap should occur. On the other hand, when `score[x]` is 65 and `score[x + 1]` is 95, then no swap should occur. See Figure 9-5.

FIGURE 9-5: DECISION SHOWING WHEN TO CALL `switchValues()` MODULE



```
if score[x] > score[x + 1] then
     perform switchValues()
endif
```

TIP

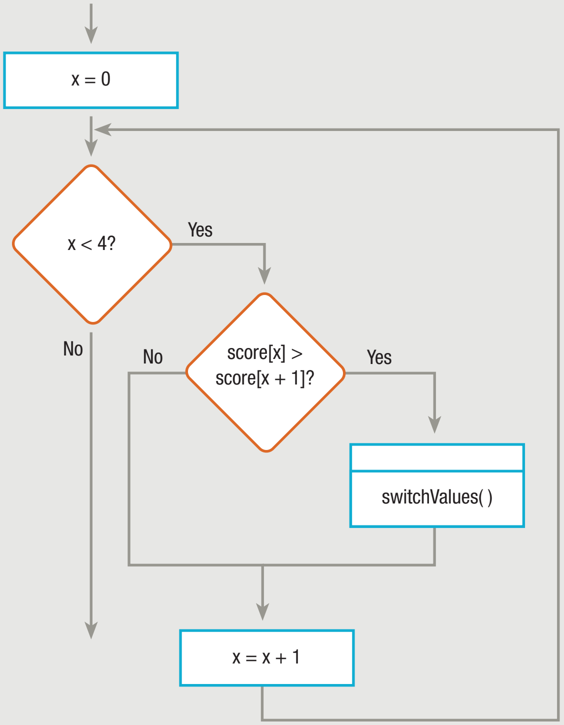
For a descending sort, in which you want to end up with the highest value first, write the decision so that you perform the switch when `score[x]` is *less than* `score[x + 1]`.

TIP

In the sort, you could use either greater than (>) or greater than or equal to (>=) to compare adjacent values. Using the greater than comparison to determine when to switch values in the sort is more efficient than using greater than or equal to, because if two compared values are equal, there is no need to swap them.

You must execute the decision `score[x] > score[x + 1]?` four times—when `x` is 0, 1, 2, and 3. You should not attempt to make the decision when `x` is 4, because then you would compare `score[4]` to `score[4 + 1]`, and there is no valid position for `score[5]` in the array. (Remember that the valid subscripts in a five-element array are the values 0 through 4.) Therefore, Figure 9-6 shows the correct loop, which compares the first two array elements, swapping them if they are out of order, increases the subscript, and continues to test array element values and make appropriate swaps while the array subscript, `x`, is less than 4.

FIGURE 9-6: LOOP THAT COMPARES ENTIRE LIST OF FIVE SCORES, MAKING NECESSARY SWAPS



```
x = 0
while x < 4
     if score[x] > score[x + 1] then
          perform switchValues()
     endif
     x = x + 1
endwhile
```

If you have these original scores:

```
score[0] = 90
score[1] = 85
score[2] = 65
score[3] = 95
score[4] = 75
```

then the logic proceeds like this:

1. Set `x` to 0.
2. The value of `x` is less than 4, so enter the loop.
3. Compare `score[x]`, 90, to `score[x + 1]`, 85. The two scores are out of order, so they are switched.

The list is now:

```
score[0] = 85
score[1] = 90
score[2] = 65
score[3] = 95
score[4] = 75
```

4. After the swap, add 1 to `x` so `x` is 1.
5. Return to the top of the loop. The value of `x` is less than 4, so enter the loop a second time.
6. Compare `score[x]`, 90, to `score[x + 1]`, 65. These two values are out of order, so swap them.

Now the result is:

```
score[0] = 85
score[1] = 65
score[2] = 90
score[3] = 95
score[4] = 75
```

7. Add 1 to `x`, so `x` is now 2.
8. Return to the top of the loop. The value of `x` is less than 4, so enter the loop.
9. Compare `score[x]`, 90, to `score[x + 1]`, 95. These values are in order, so no switch is made.
10. Add 1 to `x`, making it 3.
11. Return to the top of the loop. The value of `x` is less than 4, so enter the loop.
12. Compare `score[x]`, 95, to `score[x + 1]`, 75. These two values are out of order, so switch them.

Now the list is as follows:

```
score[0] = 85
score[1] = 65
score[2] = 90
score[3] = 75
score[4] = 95
```

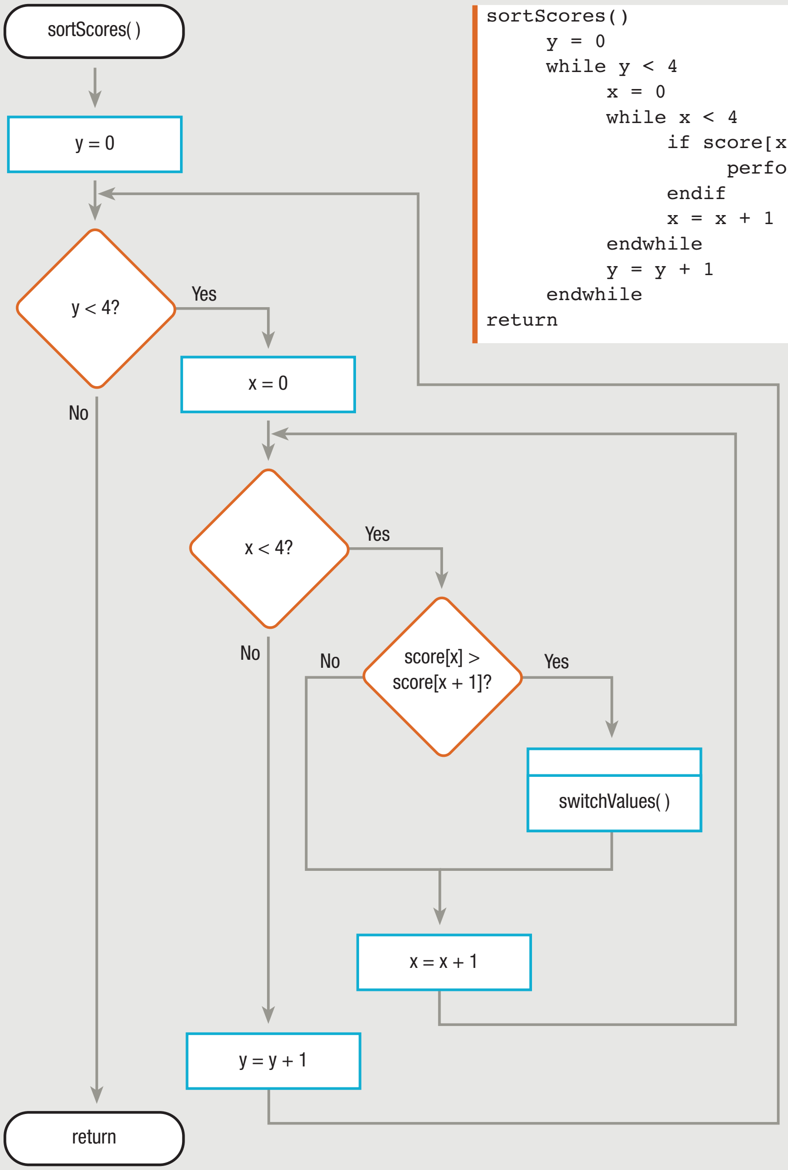
13. Add 1 to `x`, making it 4.
14. Return to the top of the loop. The value of `x` is 4, so do not enter the loop again.

When `x` reaches 4, every element in the list has been compared with the one adjacent to it. The highest score, a 95, has "sunk" to the bottom of the list. However, the scores still are not in order. They are in slightly better ascending order than they were to begin with, because the largest value is at the bottom of the list, but they are still out of order. You need to repeat the entire procedure illustrated in Figure 9-6 so that 85 and 65 (the current `score[0]` and `score[1]` values) can switch places, and 90 and 75 (the current `score[2]` and `score[3]` values) can switch places. Then, the scores will be 65, 85, 75, 90, and 95. You will have to perform the procedure to go through the list yet again to swap the 85 and 75.

As a matter of fact, if the scores had started out in the worst possible order (95, 90, 85, 75, 65), the process shown in Figure 9-6 would have to take place four times. In other words, you would have to pass through the list of values four times, making appropriate swaps, before the numbers would appear in perfect ascending order. You need to place the loop in Figure 9-6 within another loop that executes four times.

Figure 9-7 shows the complete logic for the `sortScores()` module. The `sortScores()` module uses a loop control variable named `y` to cycle through the list of scores four times. The `y` variable is added to the variable list declared in `housekeeping()`. With an array of five elements, it takes four comparisons to work through the array once, comparing each pair, and it takes four sets of those comparisons to ensure that every element in the entire array is in sorted order.

FIGURE 9-7: THE `sortScores()` MODULE



```
sortScores()
     y = 0
     while y < 4
          x = 0
          while x < 4
               if score[x] > score[x + 1] then
                    perform switchValues()
               endif
               x = x + 1
          endwhile
          y = y + 1
     endwhile
return
```

When you sort the elements in an array this way, you use nested loops—an inner loop within an outer loop. The general rule is that, whatever the number of elements in the array, the greatest number of pair comparisons you need to make during each loop is *one less* than the number of elements in the array. You use an inner loop to make the pair comparisons. In addition, the number of times you need to process the list of values is *one less* than the number of elements in the array. You use an outer loop to control the number of times you walk through the list. As an example, if you want to sort a 10-element array, you make nine pair comparisons on each of nine rotations through the loop, executing a total of 81 score comparison statements.

TIP □ □ □ □

In many cases, you do not want to sort a single data item such as a score. Instead, you might want to sort data records that contain fields such as ID number, name, and score, placing the records in score order. The sorting procedure remains basically the same, but you need to store entire records in an array. Then, you make your comparisons based on a single field, but you make your swaps using entire records.

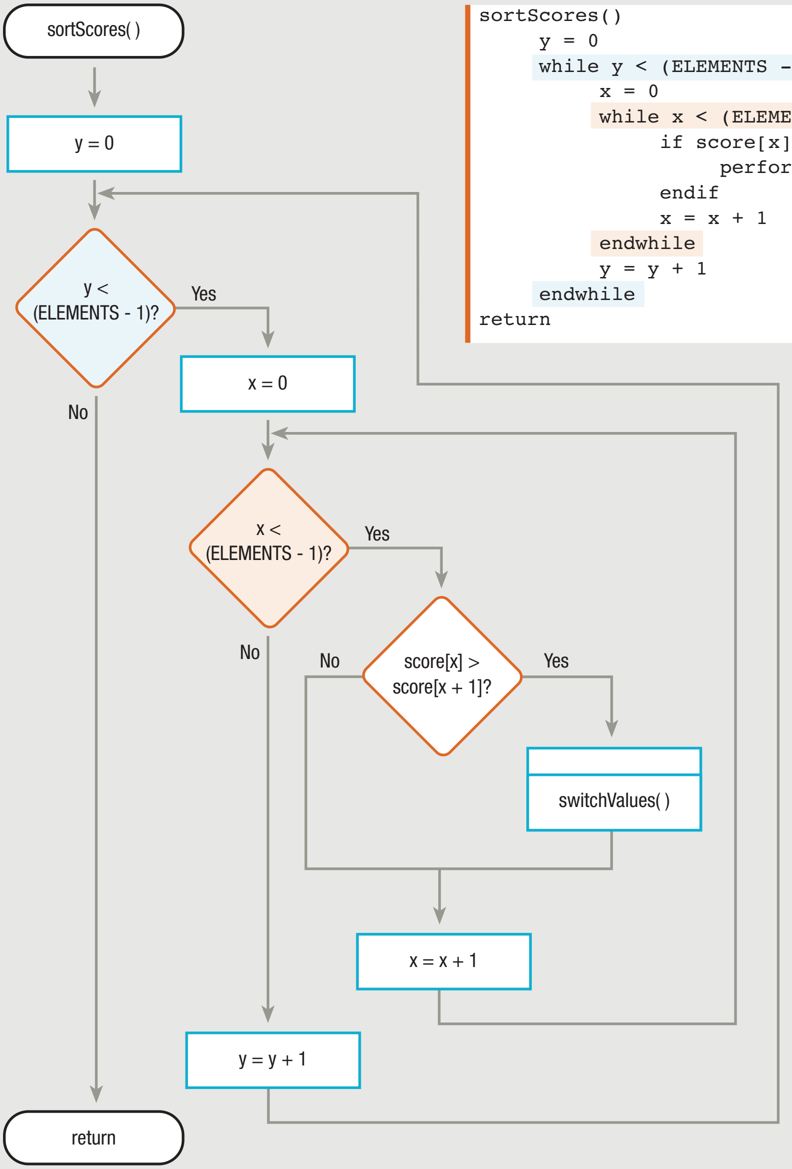
REFINING THE BUBBLE SORT BY USING A CONSTANT FOR THE ARRAY SIZE

Keep in mind that when performing a bubble sort, you need to perform one fewer pair comparison than you have elements. You also pass through the list of elements one fewer time than you have elements. In Figure 9-7, you sorted a five-element loop, so you performed the inner loop while `x` was less than 4 and the outer loop while `y` was less than 4. You can add a refinement that makes the sorting logic easier to understand. When performing a bubble sort on an array, you compare two separate loop control variables with a value that equals the number of elements in the list. If the number of elements in the array is stored in a constant named `ELEMENTS`, the general logic for a bubble sort is shown in Figure 9-8.

To use the logic shown in Figure 9-8, you must declare `ELEMENTS` along with any other variables and constants in the `housekeeping()` module. There you can set the value of `ELEMENTS` to 5, because you know there are five elements in the array to be sorted. Besides being useful for sorting, the `ELEMENTS` constant is also useful in any module that prints the scores, sums them, or performs any other activity with the list. For example, Figure 9-9 shows an entire program that uses a bubble sort. Not only does the `sortScores()` module use `ELEMENTS` to control the number of passes through the array to perform the sort, but the `finishUp()` module in Figure 9-9 also uses the `ELEMENTS` constant to control the print loop. One advantage to using a named constant instead of an unnamed, literal constant (such as 5) in your program is that if you modify the program array to accommodate more or fewer scores in the future, you can simply change the value in the named constant once where it is defined. Then, you do not need to alter every instance of a literal constant number throughout the program; the named location automatically holds the correct value in each place in the program where it is used.

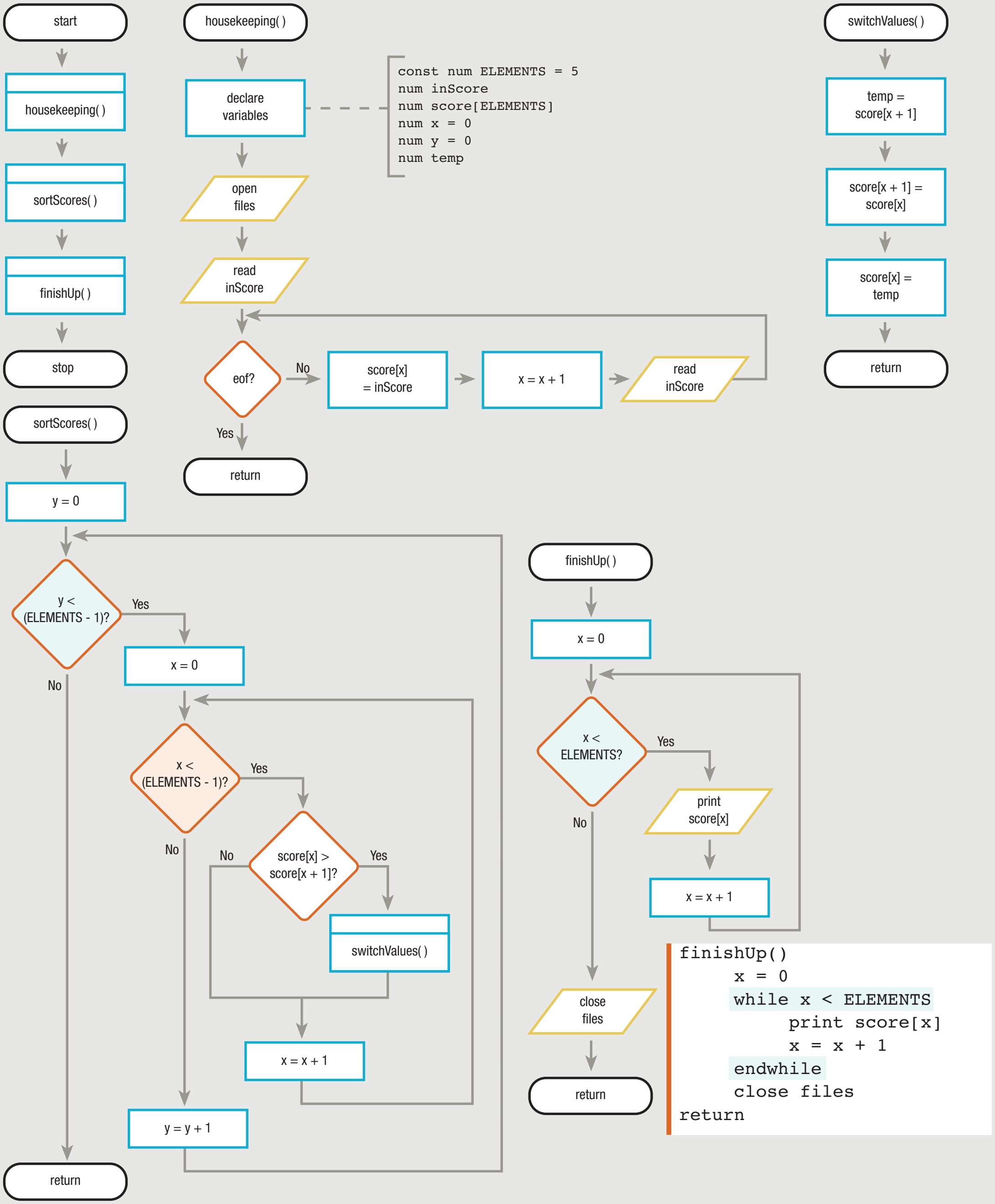
Figure 9-9 shows pseudocode for the `finishUp()` module only; pseudocode for the other modules has been shown in previous figures in this chapter.

FIGURE 9-8: GENERIC BUBBLE SORT MODULE USING A NAMED CONSTANT FOR NUMBER OF ELEMENTS



```
sortScores()
     y = 0
     while y < (ELEMENTS - 1)
          x = 0
          while x < (ELEMENTS - 1)
               if score[x] > score[x + 1] then
                    perform switchValues()
               endif
               x = x + 1
          endwhile
          y = y + 1
     endwhile
return
```

FIGURE 9-9: A COMPLETE SCORE-SORTING PROGRAM THAT PRINTS THE SORTED SCORES



```
const num ELEMENTS = 5
num inScore
num score[ELEMENTS]
num x = 0
num y = 0
num temp
```

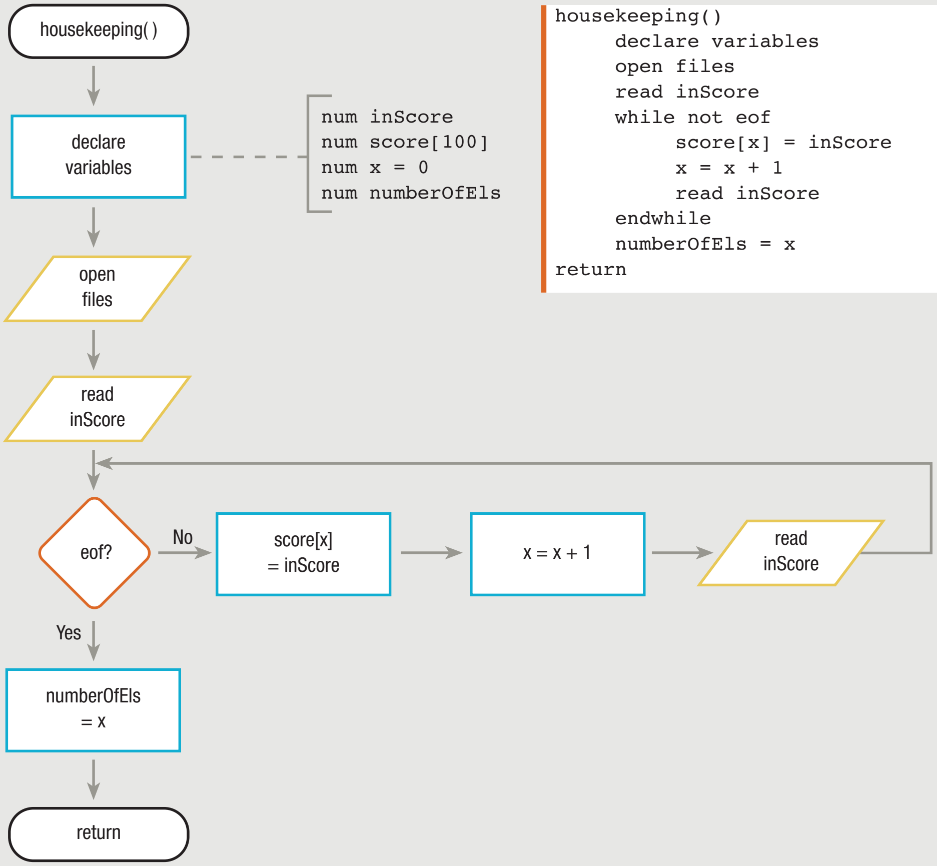
```
finishUp()
     x = 0
     while x < ELEMENTS
          print score[x]
          x = x + 1
     endwhile
     close files
return
```

SORTING A LIST OF VARIABLE SIZE

In the score-sorting program in Figure 9-9, an `ELEMENTS` constant was initialized to the number of elements to be sorted near the start of the program—within the `housekeeping()` module. Sometimes, you don't want to create a constant such as `ELEMENTS` at the start of the program. You might not know how many array elements will hold valid values—for example, sometimes when you run the program, the input file contains only three or four scores to sort, and sometimes it contains 20. In other words, what if the size of the list to be sorted varies? Rather than initializing a constant to a fixed value, you can count the input scores, and then give a variable the value of the number of array elements to use after you know how many scores exist.

To keep track of the number of elements stored in an array, you can create a `housekeeping()` module such as the one shown in Figure 9-10. When you read each `inScore` during `housekeeping()`, you increase `x` by 1 in order to place each new score into a successive element of the `score` array. In this example, the `score` array is created to hold 100 elements, a number larger than you anticipate you will need. The variable `x` is initialized to 0. After you read one `inScore` value and place it in the first element of the array, `x` is increased to 1. After a second score is read and placed in `score[1]`, `x` is increased to 2, and so on. After you reach `eof`, `x` holds the number of elements that have been placed in the array, so you can set a variable named `numberOfEls` to the value of `x`. With this approach, it doesn't matter if there are not enough `inScore` values to fill the array. You simply make one fewer pair comparison than the number of the value held in `numberOfEls`. Using this technique, you avoid always making a larger fixed number of pair comparisons. For example, if there are 35 scores in the input file, `numberOfEls` will be set to 35 in the `housekeeping()` module, and when the program sorts, it will use 35 as a cutoff point for the number of pair comparisons to make. The sorting program will never make pair comparisons on array elements 36 through 100—those elements will just "sit there," never being involved in a comparison or swap.

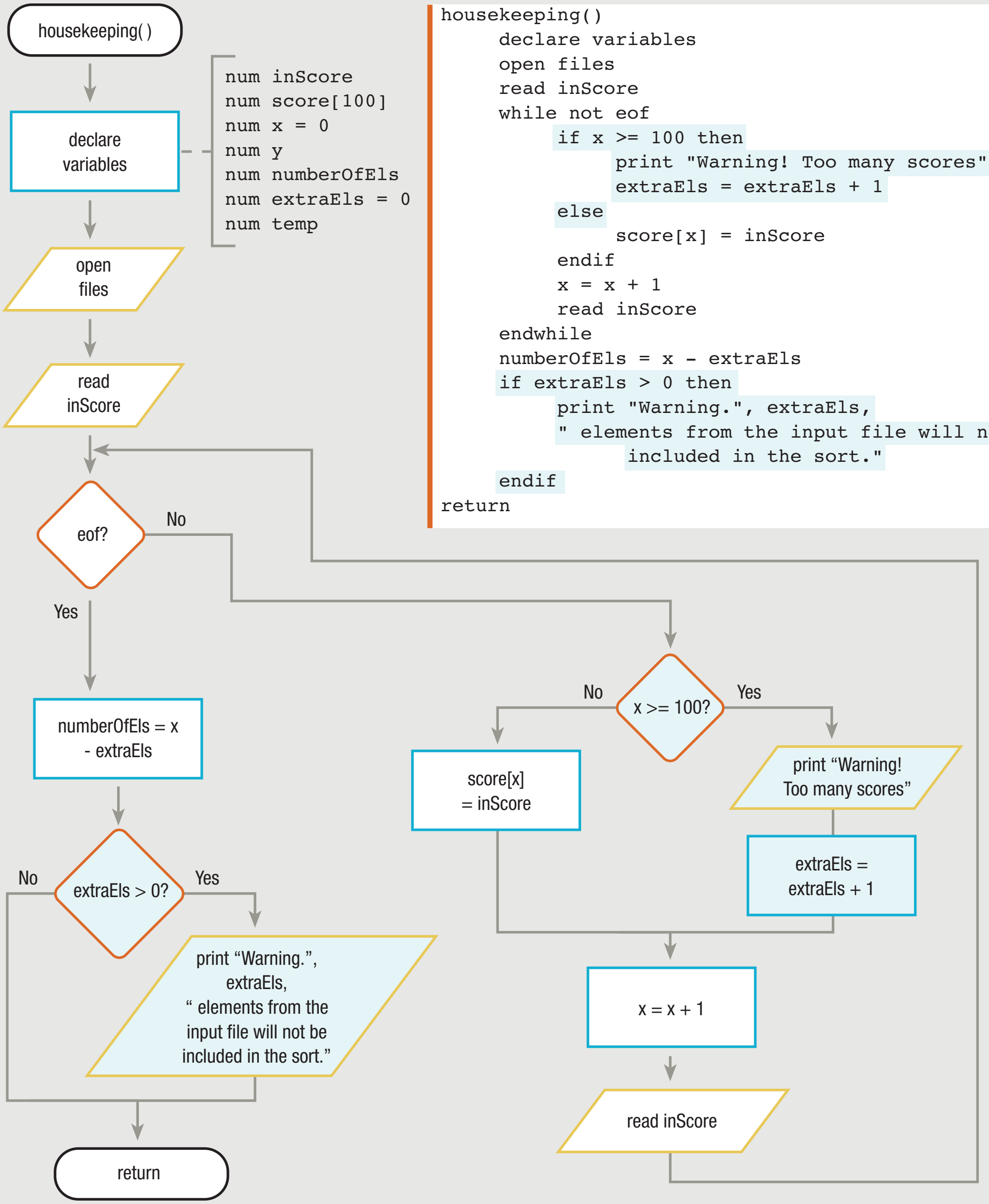
FIGURE 9-10: THE `housekeeping()` MODULE FOR A SCORE-SORTING PROGRAM THAT ACCOMMODATES A VARIABLE-SIZE INPUT FILE



```
housekeeping()
     declare variables
     open files
     read inScore
     while not eof
          score[x] = inScore
          x = x + 1
          read inScore
     endwhile
     numberOfEls = x
return
```

When you count the input records and use the **`numberOfEls`** variable, it does not matter if there are not enough scores to fill the array. However, it does matter if there are more scores than the array can hold. Every array must have a finite size, and it is an error to try to store data past the end of the array. When you don't know how many elements will be stored in an array, you must overestimate the number of elements you declare. If the number of scores in the **`score`** array can be 100 or fewer, then you can declare the **`score`** array to have a size of 100, and you can use 100 elements or fewer. Figure 9-11 shows the pseudocode that provides one possibility for an additional improvement to the **`housekeeping()`** module in Figure 9-10. If you use the logic in Figure 9-11, you read **`inScore`** values until **`eof`**, but if the array subscript **`x`** equals or exceeds 100, you display a warning message and do not attempt to store any additional **`inScore`** values in the **`score`** array. When a program uses the **`housekeeping()`** logic shown in Figure 9-11, after **`x`** becomes 100, only a warning message is displayed—no new elements are added to the array. To provide additional information to the user, extra elements are counted when they exist, and a message is displayed so the user understands exactly how many unsorted elements exist in the input file.

FIGURE 9-11: FLOWCHART AND PSEUDOCODE FOR `housekeeping()` THAT PREVENTS OVEREXTENDING THE ARRAY



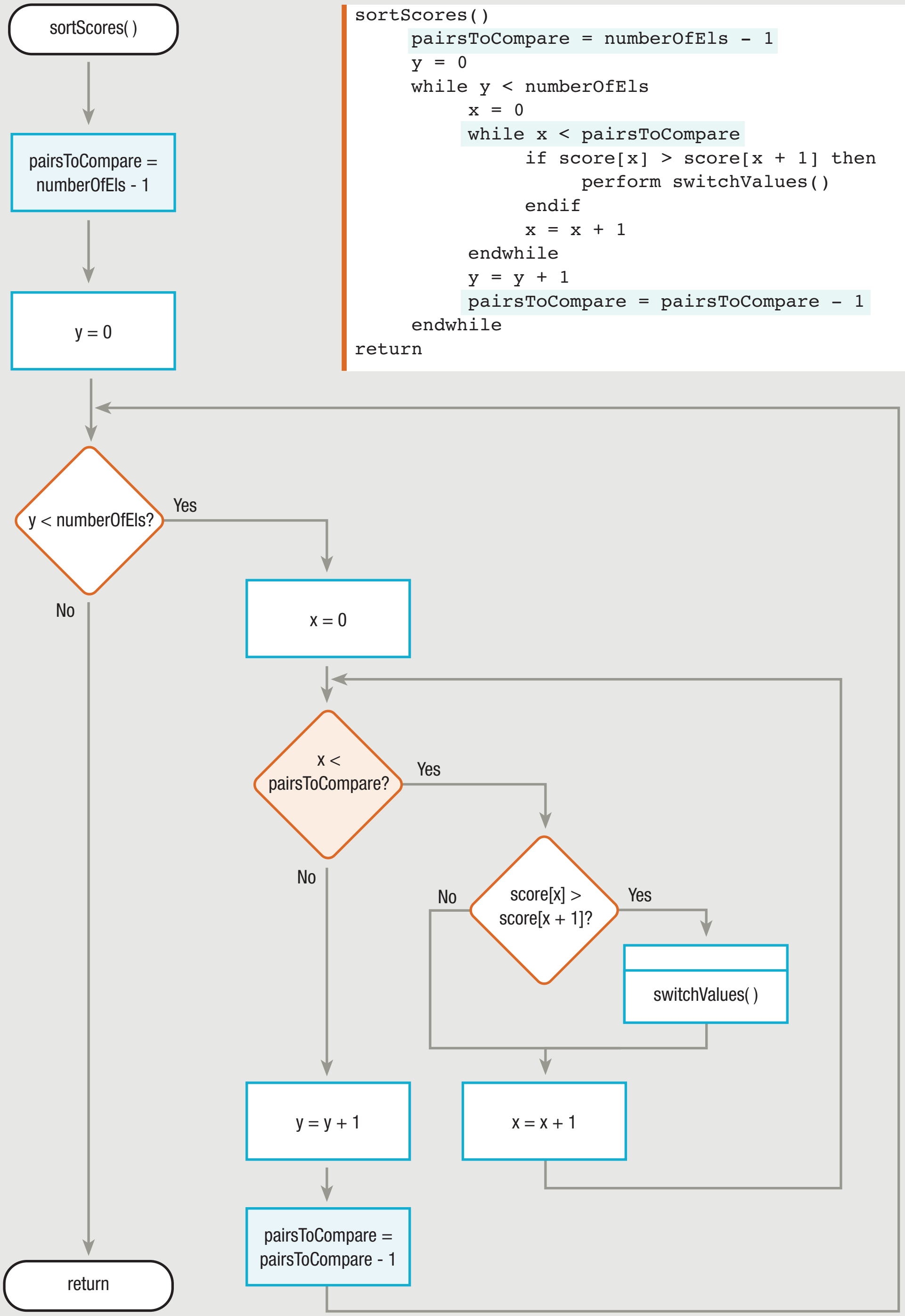
```
housekeeping()
     declare variables
     open files
     read inScore
     while not eof
          if x >= 100 then
               print "Warning! Too many scores"
               extraEls = extraEls + 1
          else
               score[x] = inScore
          endif
          x = x + 1
          read inScore
     endwhile
     numberOfEls = x - extraEls
     if extraEls > 0 then
          print "Warning.", extraEls,
          " elements from the input file will not be
                included in the sort."
     endif
return
```

REFINING THE BUBBLE SORT BY REDUCING UNNECESSARY COMPARISONS

You can make additional improvements to the bubble sort created in the previous sections. As illustrated in Figure 9-8, when performing the sorting module for a bubble sort, you pass through a list, making comparisons and swapping values if two values are out of order. If you are performing an ascending sort, then after you have made one pass through the list, the largest value is guaranteed to be in its correct final position at the bottom of the list. Similarly, the second-largest element is guaranteed to be in its correct second-to-last position after the second pass through the list, and so on. If you continue to compare every element pair in the list on every pass through the list, you are comparing elements that are already guaranteed to be in their final correct position.

On each pass through the array, you can afford to stop your pair comparisons one element sooner. In other words, after the first pass through the list, there is no longer a need to check the bottom element; after the second pass, there is no need to check the two bottom elements. You can avoid comparing these already-in-place values by creating a new variable, `pairsToCompare`, and setting it equal to the value of `numberOfEls – 1`. On the first pass through the list, every pair of elements is compared, so `pairsToCompare` *should* equal `numberOfEls – 1`. In other words, with five array elements to sort, there are four pairs to compare. For each subsequent pass through the list, `pairsToCompare` should be reduced by 1, because after the first pass there's no need to check the bottom element anymore. See Figure 9-12 to examine the use of the `pairsToCompare` variable.

FIGURE 9-12: FLOWCHART AND PSEUDOCODE FOR `sortScores()` MODULE USING `pairsToCompare` VARIABLE



```
sortScores()
    pairsToCompare = numberOfEls - 1
    y = 0
    while y < numberOfEls
        x = 0
        while x < pairsToCompare
            if score[x] > score[x + 1] then
                perform switchValues()
            endif
            x = x + 1
        endwhile
        y = y + 1
        pairsToCompare = pairsToCompare - 1
    endwhile
return
```

REFINING THE BUBBLE SORT BY ELIMINATING UNNECESSARY PASSES

A final improvement that could be made to the bubble sort module in Figure 9-12 is one that reduces the number of passes through the array. If array elements are so badly out of order that they are in reverse order, then it takes many passes through the list to place it in order; it takes one fewer pass than the value in `numberOfEls` to complete all the comparisons and swaps needed to get the list in order. However, when the array elements are in order or nearly in order to start, all the elements might be correctly arranged after only a few passes through the list. All subsequent passes result in no swaps. For example, assume the original scores are as follows:

```
score[0] = 65
score[1] = 75
score[2] = 85
score[3] = 90
score[4] = 95
```

The bubble sort module in Figure 9-12 would pass through the array list four times, making four sets of pair comparisons. It would always find that each `score[x]` is *not* greater than the corresponding `score[x + 1]`, so no switches would ever be made. The scores would end up in the proper order, but they *were* in the proper order in the first place; therefore, a lot of time would be wasted.

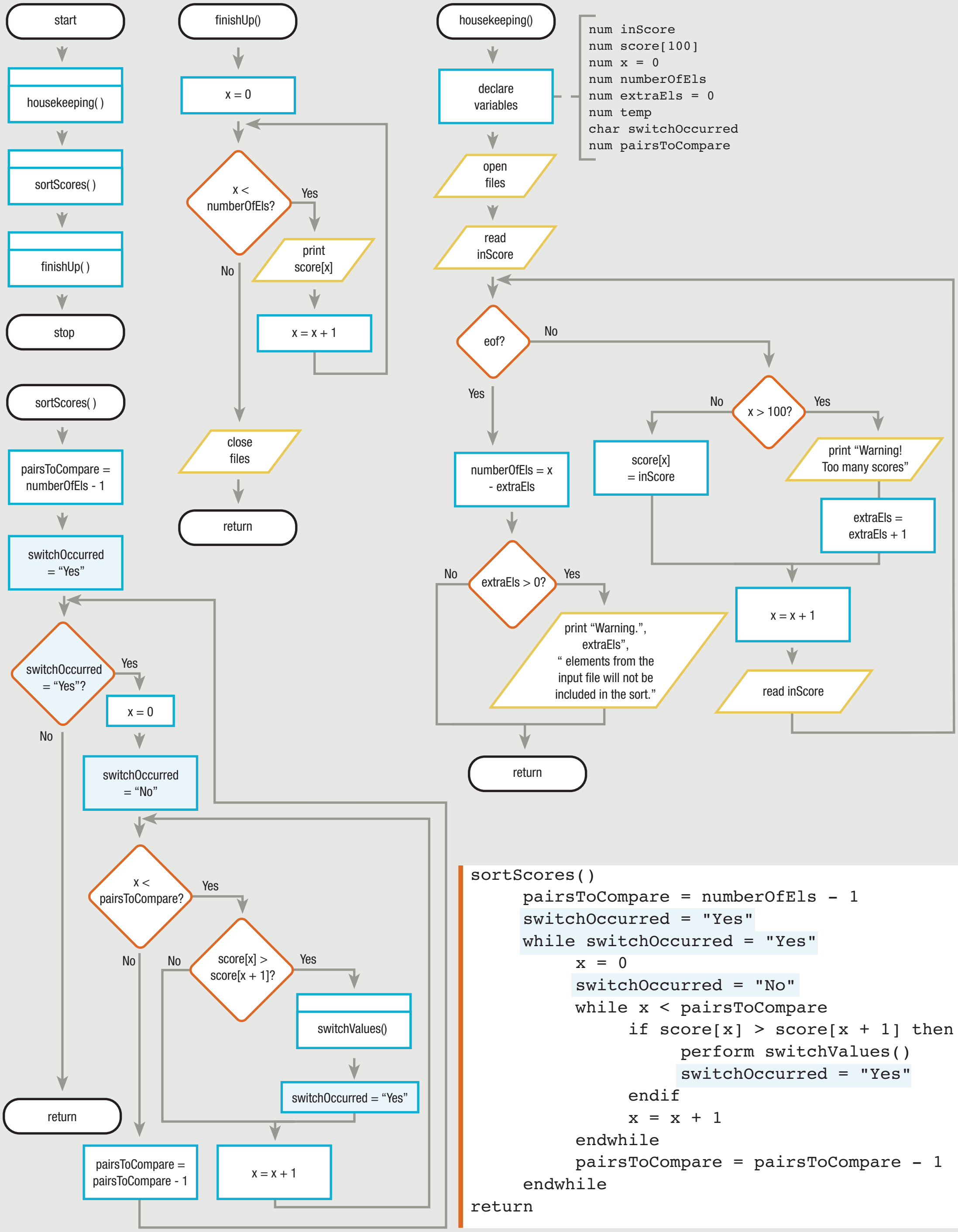
A possible remedy is to add a flag variable that you set to a "continue" value on any pass through the list in which any pair of elements is swapped (even if just one pair), and that holds a different "finished" value when no swaps are made—that is, all elements in the list are already in the correct order. For example, you can create a variable named `switchOccurred` and set it to "No" at the start of each pass through the list. You can change its value to "Yes" each time the `switchValues()` module is performed (that is, each time a switch is necessary).

If you ever "make it through" the entire list of pairs without making a switch, the `switchOccurred` flag will *not* have been set to "Yes", meaning that no switch has occurred and that the array elements must already be in the correct order. This *might* be on the first or second pass through the array list, or it might not be until a much later pass. If the array elements are already in the correct order at any point, there is no need to make more passes through the list. You can stop making passes through the list when `switchOccurred` is "No" after a complete trip through the array.

Figure 9-13 illustrates a module that sorts scores and uses a `switchOccurred` flag. At the beginning of the `sortScores()` module, initialize `switchOccurred` to "Yes" before entering the comparison loop the first time. Then, immediately set `switchOccurred` to "No". When a switch occurs—that is, when the `switchValues()` module executes—set `switchOccurred` to "Yes".

Figure 9-13 shows pseudocode for the `sortScores()` module only; pseudocode for the other modules has been shown in previous figures in this chapter.

FIGURE 9-13: BUBBLE SORT WITH `switchOccurred` FLAG



```
sortScores()
     pairsToCompare = numberOfEls - 1
     switchOccurred = "Yes"
     while switchOccurred = "Yes"
          x = 0
          switchOccurred = "No"
          while x < pairsToCompare
               if score[x] > score[x + 1] then
                    perform switchValues()
                    switchOccurred = "Yes"
               endif
               x = x + 1
          endwhile
          pairsToCompare = pairsToCompare - 1
     endwhile
return
```

TIP

With the addition of the flag variable in Figure 9-13, you no longer need the variable `y`, which was keeping track of the number of passes through the list. Instead, you just keep going through the list until you can make a complete pass without any switches. For a list that starts in perfect order, you go through the loop only once. For a list that starts in the *worst* possible order, you will make a switch with every pair each time through the loop until `pairsToCompare` has been reduced to 0. In this case, on the last pass through the loop, `x` is set to 1, `switchOccurred` is set to "No", `x` is no longer less than or equal to `pairsToCompare`, and the loop is exited.

USING AN INSERTION SORT

The bubble sort works well and is relatively easy for novice array users to understand and manipulate, but even with all the improvements you added to the original bubble sort in previous sections, it is actually one of the least efficient sorting methods available. An insertion sort provides an alternate method for sorting data, and it usually requires fewer comparison operations.

TIP

Although a sort (such as the bubble sort) might be inefficient, it is easy to understand. When programming, you frequently weigh the advantages of using simple solutions against writing more complicated ones that perform more efficiently.

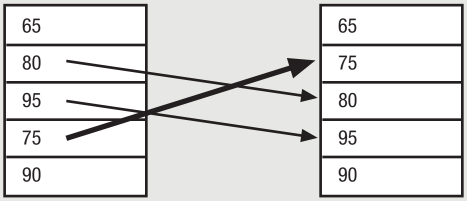
As with the bubble sort, when using an **insertion sort**, you also look at each pair of elements in an array. When you find an element that is smaller than the one before it (for an ascending sort), this element is "out of order." As soon as you locate such an element, search the array backward from that point to see where an element smaller than the out-of-order element is located. At that point, you open a new position for the out-of-order element by moving each subsequent element down one position. Then, you insert the out-of-order element into the newly opened position.

For example, consider these scores:

```
score[0] = 65
score[1] = 80
score[2] = 95
score[3] = 75
score[4] = 90
```

If you want to rearrange the scores in ascending order using an insertion sort, you begin by comparing `score[0]` and `score[1]`, which are 65 and 80, respectively. You determine that they are in order, and leave them alone. Then, you compare `score[1]` and `score[2]`, which are 80 and 95, and leave them alone. When you compare `score[2]`, 95, and `score[3]`, 75, you determine that the 75 is "out of order." Next, you look backward from the `score[3]` of 75. The value of `score[2]` is not smaller than `score[3]`, nor is `score[1]`; however, because `score[0]` is smaller than `score[3]`, `score[3]` should follow `score[0]`. So you store `score[3]` in a temporary variable, then move `score[1]` and `score[2]` "down" the list to higher subscripted positions. You move `score[2]`, 95, to the `score[3]` position. Then, you move `score[1]`, 80, to the `score[2]` position. Finally, you assign the value of the temporary variable, 75, to the `score[1]` position. Figure 9-14 diagrams the movements as 75 moves up to the second position and 80 and 95 move down.

FIGURE 9-14: MOVEMENT OF THE VALUE "75" TO A "BETTER" ARRAY POSITION IN AN INSERTION SORT



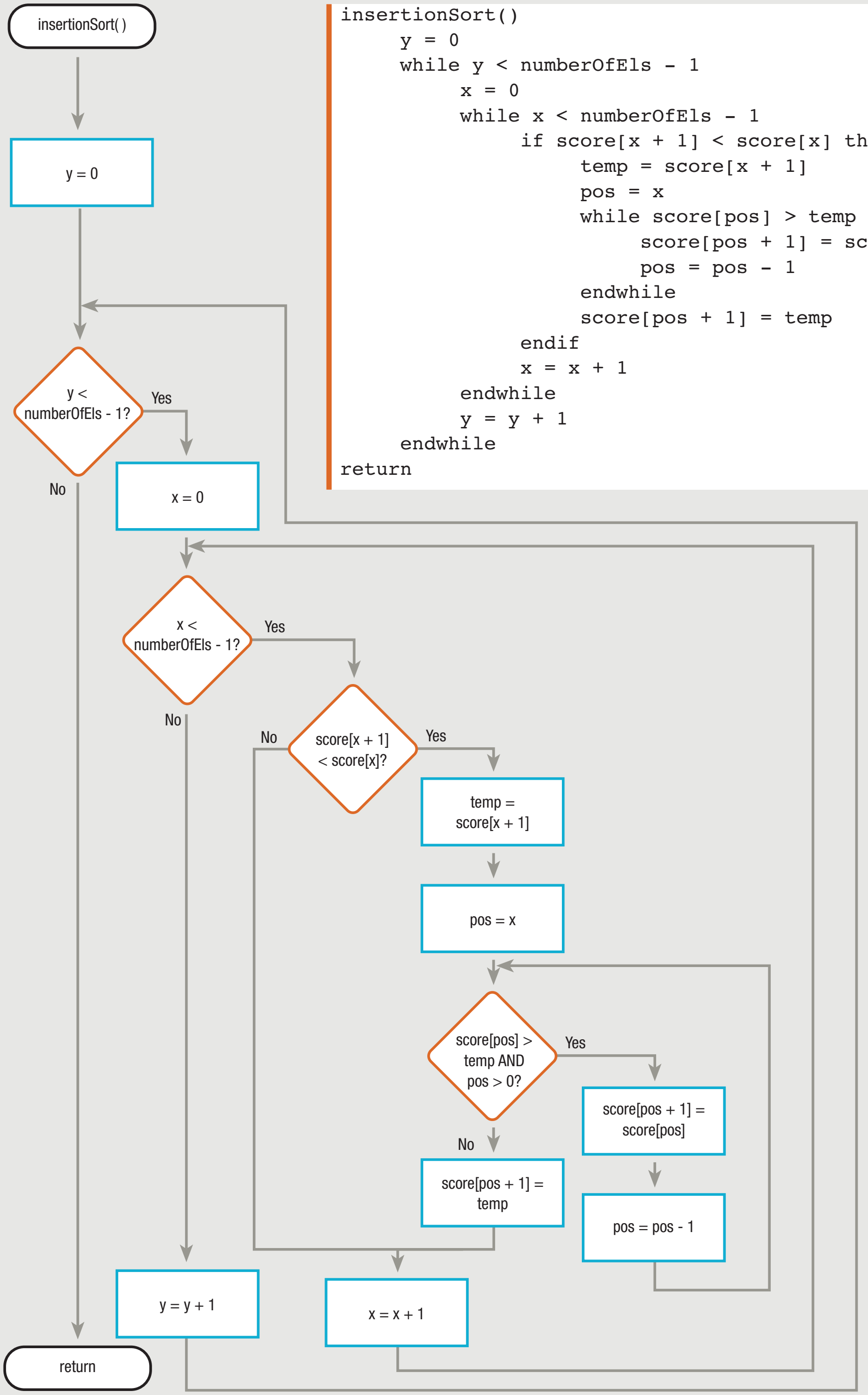
After the sort finds the first element that was out of order and inserts it in a “better” location, the results are:

```
score[0] = 65
score[1] = 75
score[2] = 80
score[3] = 95
score[4] = 90
```

You then continue down the list, comparing each pair of variables. A complete insertion sort module is shown in Figure 9-15.

The logic for the insertion sort is slightly more complicated than that for the bubble sort, but the insertion sort is more efficient because, for the average out-of-order list, it takes fewer “switches” to put the list in order.

FIGURE 9-15: SAMPLE INSERTION SORT MODULE



```
insertionSort()
     y = 0
     while y < numberOfEls - 1
          x = 0
          while x < numberOfEls - 1
               if score[x + 1] < score[x] then
                    temp = score[x + 1]
                    pos = x
                    while score[pos] > temp AND pos > 0
                         score[pos + 1] = score[pos]
                         pos = pos - 1
                    endwhile
                    score[pos + 1] = temp
               endif
               x = x + 1
          endwhile
          y = y + 1
     endwhile
return
```

USING A SELECTION SORT

A selection sort provides another sorting option. In an ascending **selection sort**, the first element in the array is assumed to be the smallest. Its value is stored in a variable—for example, `smallest`—and its position in the array, 0, is stored in another variable—for example, `position`. Then, every subsequent element in the array is tested. If one with a smaller value than `smallest` is found, `smallest` is set to the new value, and `position` is set to that element's position. After the entire array has been searched, `smallest` holds the smallest value and `position` holds its position.

The element originally in `position[0]` is then switched with the `smallest` value, so at the end of the first pass through the array, the lowest value ends up in the first position, and the value that was in the first position is where the smallest value used to be.

For example, assume you have the following list of scores:

```
score[0] = 95
score[1] = 80
score[2] = 75
score[3] = 65
score[4] = 90
```

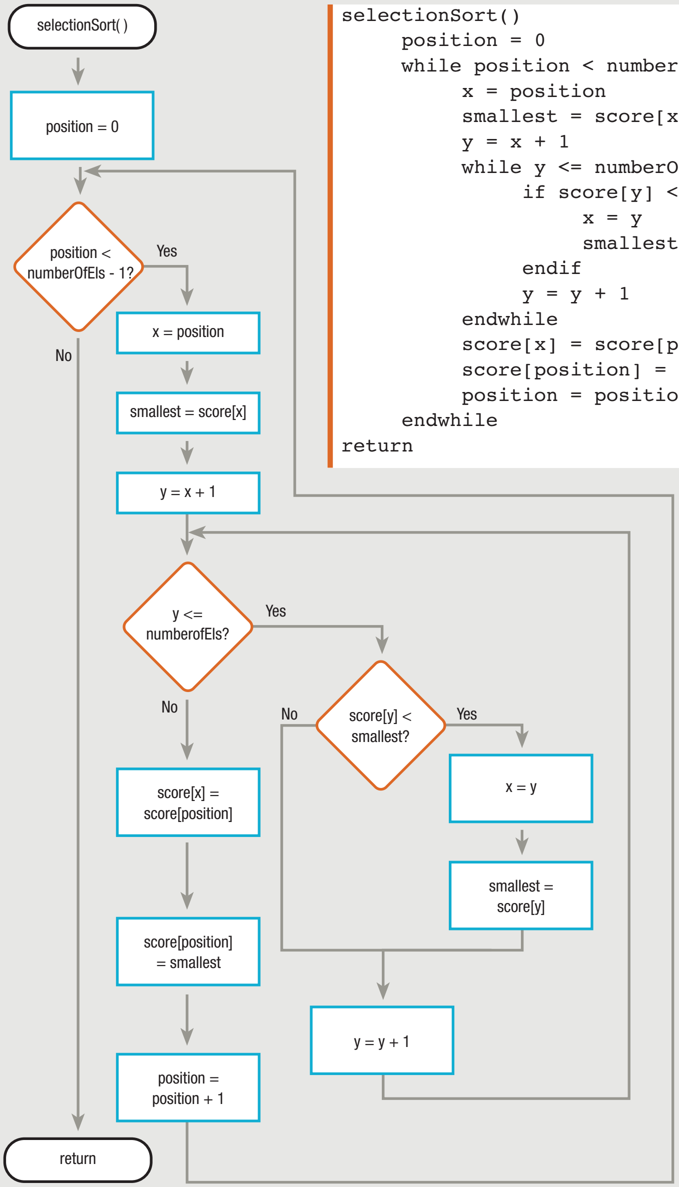
First, you place 95 in `smallest`. Then check `score[1]`; it's less than 95, so place 1 in `position` and 80 in `smallest`. Then test `score[2]`; it's smaller than `smallest`, so place 2 in `position` and 75 in `smallest`. Then test `score[3]`; because it is smaller than `smallest`, place 3 in `position` and 65 in `smallest`. Finally, check `score[4]`; it *isn't* smaller than `smallest`.

So at the end of the first pass through the list, `position` is 3 and `smallest` is 65. You move the value 95 to `score[position]`, or `score[3]`, and the value of `smallest`, 65, to `score[0]`. The list becomes:

```
score[0] = 65
score[1] = 80
score[2] = 75
score[3] = 95
score[4] = 90
```

Now that the smallest value is in the first position, you repeat the whole procedure starting with the second array element, `score[1]`. After you have passed through the list `numberOfEls - 1` times, all elements will be in the correct order. Walk through the logic shown in Figure 9-16.

FIGURE 9-16: SAMPLE SELECTION SORT MODULE



```
selectionSort()
     position = 0
     while position < numberOfEls - 1
          x = position
          smallest = score[x]
          y = x + 1
          while y <= numberOfEls
               if score[y] < smallest then
                    x = y
                    smallest = score[y]
               endif
               y = y + 1
          endwhile
          score[x] = score[position]
          score[position] = smallest
          position = position + 1
     endwhile
return
```

Like the insertion sort, the selection sort almost always requires fewer switches than the bubble sort, but the variables might be a little harder to keep track of, because the logic is a little more complex. Thoroughly understanding at least one of these sorting techniques provides you with a valuable tool for arranging data and increases your understanding of the capabilities of arrays.

USING INDEXED FILES

Sorting a list of five scores does not require significant computer resources. However, many data files contain thousands of records, and each record might contain dozens of data fields. Sorting large numbers of data records requires considerable time and computer memory. When a large data file needs to be processed in ascending or descending order based on some field, it is usually more efficient to store and access records based on their logical order than to sort and access them in their physical order. When records are stored, they are stored in some physical order. For example, if you write the names of 10 friends, each one on an index card, the stack of cards has a **physical order**—that is, a "real" order. You can arrange the cards alphabetically by the friends' last names, chronologically by age of the friendship, or randomly by throwing the cards in the air and picking them up as you find them. Whichever way you do it, the records still follow each other in *some* order. In addition to their current physical order, you can think of the cards as having a **logical order**; that is, a virtual order, based on any criterion you choose—from the tallest friend to the shortest, from the one who lives farthest away to the closest, and so on. Sorting the cards in a new physical order can take a lot of time; using the cards in their logical order without physically rearranging them is often more efficient.

A common method of accessing records in logical order is to use an index. Using an index involves identifying a key field for each record. A record's **key field** is the field whose contents make the record unique among all records in a file. For example, multiple employees can have the same last name, first name, salary, or street address, but each employee possesses a unique Social Security number, so a Social Security number field might make a good key field for a personnel file. (Because of security issues, a company-assigned employee ID number might make a better key field.) Similarly, a product number makes a good key field on an inventory file.

When you **index** records, you store a list of key fields paired with the storage address for the corresponding data record. When you use an index, you can store records on a **random-access storage device**, such as a disk, from which records can be accessed in any logical order. Each record can be placed in any physical location on the disk, and you can use the index as you would use the index in the back of a book. If you pick up a 600-page American history text because you need some facts about Betsy Ross, you do not want to start on page one and work your way through the text. Instead, you turn to the index, discover that Betsy Ross is discussed on page 418, and go directly to that page.

As pages in a book have numbers, computer memory and storage locations have **addresses**. In Chapter 1, you learned that every variable has a numeric address in computer memory; likewise, every data record on a disk has a numeric address where it is stored. You can store records in any physical order on the disk, but the index can find the records in order based on their addresses. For example, you might store a list of employees on a disk in the order in which they were hired. However, you often need to process the employees in Social Security number order. When adding a new employee to such a file, you can physically place the employee anywhere there is room available on the disk. Her Social Security number is inserted in proper order in the index, along with the physical address where her record is located.

TIP □ □ □ □ You do not need to determine a record's exact physical address in order to use it. A computer's operating system takes care of locating available storage for your records.

You can picture an index based on Social Security numbers by looking at Table 9-1.

TABLE 9-1: SAMPLE INDEX

Social Security number	Location
111-22-3456	6400
222-44-7654	4800
333-55-1234	2400
444-88-9812	5200

When you want to access the data for employee 333-55-1234, you tell your computer to look through the Social Security numbers in the index, find a match, and then proceed to the memory location specified. Similarly, when you want to process records in order based on Social Security number, you tell your system to retrieve records at the locations in the index in sequence. Thus, even though employee 111-22-3456 may have been hired last and the record is stored at the highest physical address on the disk, if the employee record has the lowest Social Security number, it will be accessed first during any ordered processing.

When a record is removed from an indexed file, it does not have to be physically removed. Its reference can simply be deleted from the index, and then it will not be part of any further processing.

USING LINKED LISTS

Another way to access records in a desired order, even though they might not be physically stored in that order, is to create a linked list. In its simplest form, creating a **linked list** involves creating one extra field in every record of stored data. This extra field holds the physical address of the next logical record. For example, a record that holds a customer's ID, name, and phone number might contain the fields:

```
custId
custName
custPhoneNum
custNextCustAddress
```

Every time you use a record, you access the next record based on the address held in the `custNextCustAddress` field.

Every time you add a new record to a linked list, you search through the list for the correct logical location for the new record. For example, assume that customer records are stored at the addresses shown in Table 9-2 and that they are linked in customer ID order. Notice that the addresses are not shown in sequential order. The records are shown in their logical order, with each one's `custNextCustAddress` field holding the address of the record shown in the following line.

TABLE 9-2: LINKED CUSTOMER LIST

Address of record	custId	custName	custPhoneNum	custNextCustAddress
0000	111	Baker	234-5676	7200
7200	222	Vincent	456-2345	4400
4400	333	Silvers	543-0912	6000
6000	444	Donovan	329-8744	eof

You can see from Table 9-2 that each customer's record contains a `custNextCustAddress` field that stores the address of the next customer who follows in customer ID number order (and not necessarily in address order). For any individual customer, the next logical customer's address might be physically distant. Each customer record, besides containing data about that customer, contains a `custNextCustAddress` field that associates the customer with the next customer who follows in `custId` value order.

Examine the file shown in Table 9-2, and suppose a new customer with number 245 and the name Newberg is acquired. Also suppose the computer operating system finds an available storage location for Newberg's data at address 8400. In this case, the procedure to add Newberg to the list is:

1. Create a variable named `currentAddress` to hold the address of the record in the list you are currently examining. Store the address of the first record in the list, 0000, in this variable.
2. Compare the new customer Newberg's ID, 245, with the current (first) record's ID, 111 (in other words, the ID at address 0000). The value 245 is higher than 111, so you save the first customer's address (the address you are currently examining), 0000, in a variable you can name `saveAddress`. The `saveAddress` variable always holds the address you just finished examining. The first customer's record contains a link to the address of the next logical customer—7200. Store the 7200 in the `currentAddress` variable.
3. Examine the second customer record, the one that physically exists at the address 7200, which is currently held in the `currentAddress` variable.
4. Compare Newberg's ID, 245, with the ID stored in the record at `currentAddress`, 222. The value 245 is higher, so save the current address, 7200, in `saveAddress` and store its `custNextCustAddress` address field, 4400, in the `currentAddress` variable.
5. Compare Newberg's ID, 245, with 333, which is the ID at `currentAddress` (4400). Up to this point, 245 had been higher than each ID tested, but this time the value 245 is lower, so that means customer 245 should logically precede customer 333. Set the `custNextCustAddress` field in Newberg's record (customer 245) to 4400, which is the address of customer 333 and the address currently stored in `currentAddress`. This means that in any future processing, Newberg's record will logically be followed by the record containing 333. Also set the `custNextCustAddress` field of the record located at `saveAddress` (7200, Vincent, customer 222, the customer who logically preceded Newberg) to the new customer Newberg's address, 8400. The updated list appears in Table 9-3.

TABLE 9-3: UPDATED CUSTOMER LIST

Address of record	custId	custName	custPhoneNum	custNextCustAddress
0000	111	Baker	234-5676	7200
7200	222	Vincent	456-2345	8400
8400	245	Newberg	222-9876	4400
4400	333	Silvers	543-0912	6000
6000	444	Donovan	329-8744	eof

As with indexing, when removing records from a linked list, the records do not need to be physically deleted from the medium on which they are stored. If you need to remove customer 333 from the preceding list, all you need to do is change Newberg's `custNextCustAddress` field to the value in Silvers' `custNextCustAddress` field, which is Donovan's address: 6000. In other words, the value of 6000 is obtained not by knowing who Newberg should point to, but by knowing who Silvers used to point to. When Newberg's record points to Donovan, Silvers' record is then bypassed during any further processing that uses the links to travel from one record to the next.

More sophisticated linked lists store *two* additional fields with each record. One field stores the address of the next record, and the other field stores the address of the *previous* record so that the list can be accessed either forward or backward.

USING MULTIDIMENSIONAL ARRAYS

An array that represents a single list of values is a **single-dimensional array** or **one-dimensional array**. For example, an array that holds five rent figures that apply to five floors of a building can be displayed in a single column, as in Figure 9-17.

FIGURE 9-17: A SINGLE-DIMENSIONAL `rent` ARRAY

```
rent[0] = 350
rent[1] = 400
rent[2] = 475
rent[3] = 600
rent[4] = 1000
```

TIP □□□□ You used the single-dimensional `rent` array in Chapter 8.

The location of any `rent` value in Figure 9-17 depends on only a single variable—the floor of the building. Sometimes, however, locating a value in an array depends on more than one variable. If you must represent values in a table or grid that contains rows and columns instead of a single list, then you might want to use a **multidimensional array**—specifically in this case, a **two-dimensional array**.

Assume that the floor is not the only factor determining rent in your building, but that another variable, `numberOfBedrooms`, also needs to be taken into account. The rent schedule might be the one shown in Table 9-4.

TABLE 9-4: RENT SCHEDULE BASED ON FLOOR AND NUMBER OF BEDROOMS

Floor	Studio apartment	1-bedroom apartment	2-bedroom apartment
0	350	390	435
1	400	440	480
2	475	530	575
3	600	650	700
4	1000	1075	1150

Each element in a two-dimensional array requires two subscripts to reference it—one subscript to determine the row and a second to determine the column. Thus, the 15 separate `rent` values for a two-dimensional array based on the rent table in Table 9-4 would be those shown in Figure 9-18.

FIGURE 9-18: TWO-DIMENSIONAL `rent` ARRAY VALUES BASED ON FLOOR AND NUMBER OF BEDROOMS

```
rent[0][0] = 350
rent[0][1] = 390
rent[0][2] = 435
rent[1][0] = 400
rent[1][1] = 440
rent[1][2] = 480
.
.
.
rent[4][2] = 1150
```

Suppose you want to read records that store a floor and number of bedrooms in an apartment, and print the appropriate rent for that apartment. If you store tenant records that contain two fields named `floor` and `numberOfBedrooms`, then the correct rent can be printed with the statement: `print rent[floor][numberOfBedrooms]`. The first subscript represents the array row; the second subscript represents the array column.

TIP □□□□

Some languages access two-dimensional array elements with commas separating the subscript values; for example, the first-floor, two-bedroom rate might be written `rent[1,2]`. In every language, you provide a subscript for the row first and for the column second.

TIP Just as within a one-dimensional array, each element in a multidimensional array must be the same data type.

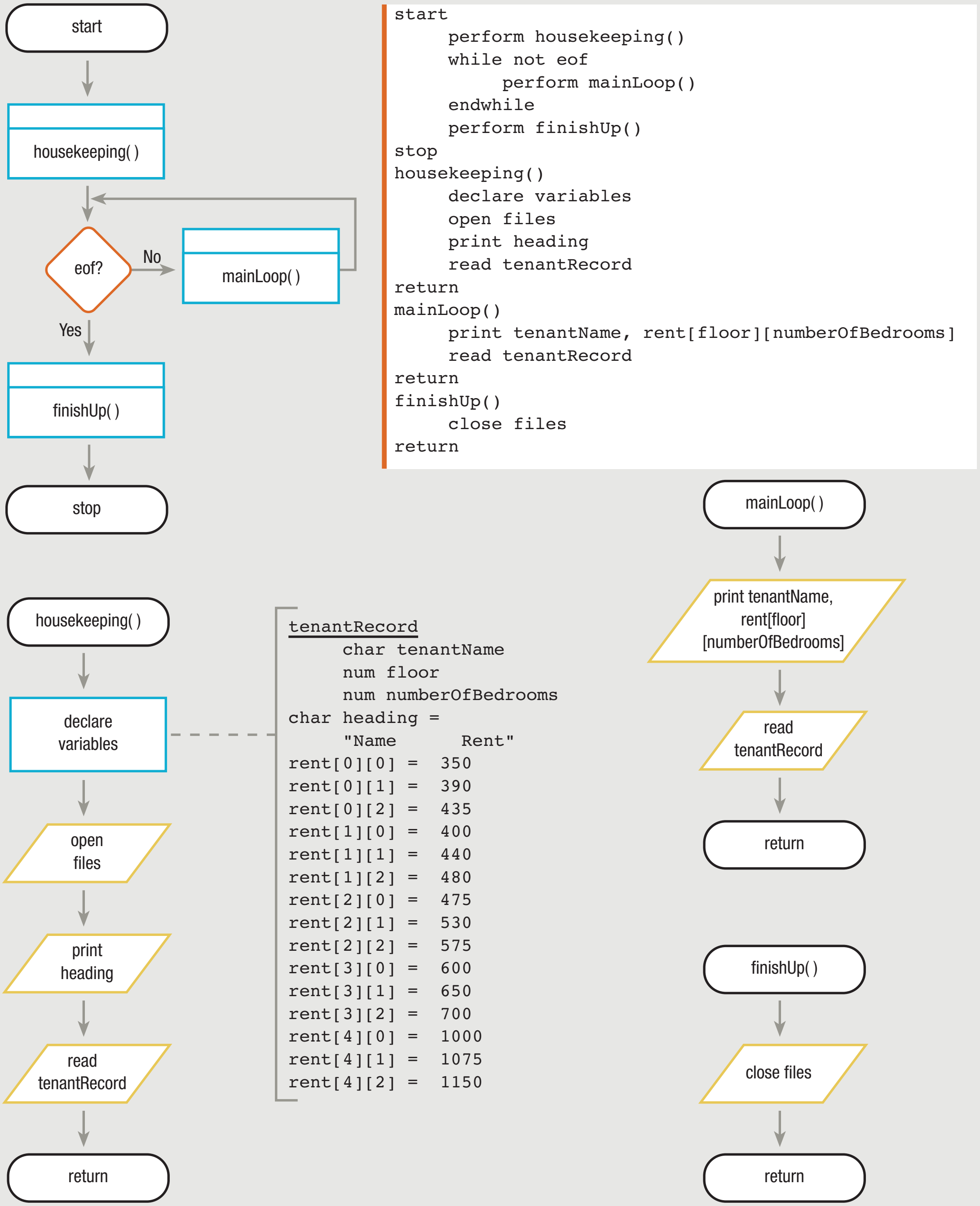
Two-dimensional arrays are never actually *required* in order to achieve a useful program. The same 15 categories of rent information could be stored in three separate single-dimensional arrays of five elements each. Of course, don't forget that even one-dimensional arrays are never *required* for you to be able to solve a problem. You could also declare 15 separate rent variables and make 15 separate decisions to determine the rent.

Figure 9-19 shows an entire program that produces a report that determines rent amounts for tenant records stored in a file. Notice that although significant setup is required to provide all the values for the rents, the `mainLoop()` module is extremely brief and easy to follow.

Some languages allow multidimensional arrays containing three levels, or **three-dimensional arrays**, in which you access array values using three subscripts. For example, rent might not only be determined by the two factors `floor` and `numberOfBedrooms`. There might also be 12 different buildings. The third dimension of a three-dimensional array to hold all these different rents would be a variable such as `buildingNumber`.

Some languages allow even more dimensions. It's usually hard for people to keep track of more than three dimensions, but if five variables determine rent—for example, floor number, number of bedrooms, building number, city number, and state number—you might want to try using a five-dimensional array.

FIGURE 9-19: RENT-DETERMINING PROGRAM



```
start
     perform housekeeping()
     while not eof
          perform mainLoop()
     endwhile
     perform finishUp()
stop
housekeeping()
     declare variables
     open files
     print heading
     read tenantRecord
return
mainLoop()
     print tenantName, rent[floor][numberOfBedrooms]
     read tenantRecord
return
finishUp()
     close files
return
```

```
tenantRecord
     char tenantName
     num floor
     num numberOfBedrooms
char heading =
     "Name       Rent"
rent[0][0] =  350
rent[0][1] =  390
rent[0][2] =  435
rent[1][0] =  400
rent[1][1] =  440
rent[1][2] =  480
rent[2][0] =  475
rent[2][1] =  530
rent[2][2] =  575
rent[3][0] =  600
rent[3][1] =  650
rent[3][2] =  700
rent[4][0] =  1000
rent[4][1] =  1075
rent[4][2] =  1150
```

CHAPTER SUMMARY

□ When the sequential order of data records is not the order desired for processing or viewing, the data needs to be sorted in ascending or descending order based on the contents of one or more fields.

□ You can swap two values by creating a temporary variable to hold one of the values. Then, you can assign the second value to the temporary variable, assign the first value to the second, and assign the temporary value to the first variable.

□ In a bubble sort, items in a list are compared in pairs, and when an item is out of order, it swaps with the item below it. With an ascending bubble sort, after each adjacent pair of items in a list has been compared once, the largest item in the list will have "sunk" to the bottom.

□ When performing a bubble sort on an array, you compare two separate loop control variables with a value that equals the number of elements in the list. An advantage to using a variable instead of a constant to hold the number of elements is that if you modify the program array to accommodate more or fewer elements in the future, you can simply change the value in the variable once, where it is defined.

□ On each pass through an array that is being sorted using a bubble sort, you can afford to stop your pair comparisons one element sooner than the time before.

□ To avoid making unnecessary passes through a list while performing a bubble sort, you can add a flag that you test on every pass through the list, to determine when all elements are already in the correct order.

□ When using an insertion sort, you look at each pair of elements in an array. When you find an element that is out of order, search the array backward from that point, find an element smaller than the out-of-order element, move each subsequent element down one position, and insert the out-of-order element into the list at the newly opened position.

□ In an ascending selection sort, the first element in the array is assumed to be the smallest. Its value and position are stored. Then, every subsequent element in the array is tested, and if one has a smaller value, the new value and position are stored. After searching the entire array, you switch the original first value with the smallest value. Then you repeat the process with each subsequent list value.

□ You can use an index to access data records in a logical order that differs from their physical order. Using an index involves identifying a key field for each record.

□ Creating a linked list involves creating an extra field within every record, to hold the physical address of the next logical record.

□ You use a multidimensional array whenever locating a value in an array depends on more than one variable.

KEY TERMS

When records are in **sequential order**, they are arranged one after another on the basis of the value in some field.

Sorted records are in order based on the contents of one or more fields.

Records in **ascending order** are arranged from lowest to highest, based on a value within a field.

Records in **descending order** are arranged from highest to lowest, based on a value within a field.

The **median** value in a list is the value in the middle position when the values are sorted.

The **mean** value in a list is the arithmetic average.

Swapping two values is the process of setting the first variable equal to the value of the second, and the second variable equal to the value of the first.

A **bubble sort** is a sort in which you arrange records in either ascending or descending order by comparing items in a list in pairs; when an item is out of order, it swaps values with the item below it.

A **sinking sort** is another name for a bubble sort.

When using an **insertion sort**, you look at each pair of elements in an array. For example, for an ascending insertion sort, when you find an element that is smaller than the one before it, you search the array backward from that point to see where an element smaller than the out-of-order element is located. At that point, you open a new position for the out-of-order element by moving each subsequent element down one position. Then, you insert the out-of-order element into the newly opened position.

In an ascending **selection sort**, you search for the smallest list value, and then swap it with the value in the first position. You then repeat the process with each subsequent list position.

A list's **physical order** is the order in which it is actually stored.

A list's **logical order** is the order in which you use it, even though it is not necessarily physically stored in that order.

A record's **key field** is the field whose contents make the record unique among all records in a file.

When you **index** records, you store a list of key fields paired with the storage address for the corresponding data record.

A **random-access storage device**, such as a disk, is one from which records can be accessed in any order.

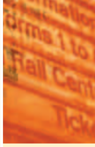
Computer memory and storage locations have **addresses**.

Creating a **linked list** involves creating one extra field in every record of stored data. This extra field holds the physical address of the next logical record.

An array that represents a single list of values is a **single-dimensional array** or **one-dimensional array**.

An array that represents a table or grid containing rows and columns is a **multidimensional array**—for example, a **two-dimensional array**.

Some languages allow **three-dimensional arrays**, in which you access values using three subscripts.



REVIEW QUESTIONS

1. **Employee records stored in order from highest-paid to lowest-paid have been sorted in ________ order.**

   a. ascending
   b. descending
   c. staggered
   d. recursive

2. **Student records stored in alphabetical order by last name have been sorted in ________ order.**

   a. ascending
   b. descending
   c. staggered
   d. recursive

3. **In the series of numbers 7, 5, 5, 5, 3, 2, and 1, what is the mean?**

   a. 3
   b. 4
   c. 5
   d. 6

4. **When computers sort data, they always ________.**

   a. place items in ascending order
   b. use a bubble sort
   c. begin the process by locating the position of the lowest value
   d. use numeric values when making comparisons

5. **Which of the following code segments correctly swaps the values of variables named `x` and `y`?**

   a.
   ```
   x = y
   y = temp
   x = temp
   ```
   b.
   ```
   x = y
   temp = x
   y = temp
   ```
   c.
   ```
   temp = x
   x = y
   y = temp
   ```
   d.
   ```
   temp = x
   y = x
   x = temp
   ```

6. **Which type of sort compares list items in pairs, swapping any two adjacent values that are out of order?**
   a. bubble sort
   b. selection sort
   c. insertion sort
   d. indexed sort

7. **Which type of sort compares pairs of values, looking for an out-of-order element, then searches the array backward from that point to see where an element smaller than the out-of-order element is located?**
   a. bubble sort
   b. selection sort
   c. insertion sort
   d. indexed sort

8. **Which type of sort tests each value in a list, looking for the smallest, then switches the element in the first list position with the smallest value?**
   a. bubble sort
   b. selection sort
   c. insertion sort
   d. indexed sort

9. **To sort a list of eight values using a bubble sort, the greatest number of times you would have to pass through the list making comparisons is ________.**
   a. six
   b. seven
   c. eight
   d. nine

10. **To sort a list of eight values using a bubble sort, the greatest number of pair comparisons you would have to make before the sort is complete is ________.**
   a. seven
   b. eight
   c. 49
   d. 64

11. **When you do not know how many items need to be sorted in a program, you create an array that has ________.**
   a. at least one element less than the number you predict you will need
   b. at least as many elements as the number you predict you will need
   c. variable-sized elements
   d. a variable number of elements

12. **In a bubble sort, on each pass through the list that must be sorted, you can stop making pair comparisons ______.**

    a. one comparison sooner
    b. two comparisons sooner
    c. one comparison later
    d. two comparisons later

13. **When performing a bubble sort on a list of 10 values, you can stop making passes through the list of values as soon as ______ on a single pass through the list.**

    a. no more than 10 swaps are made
    b. no more than nine swaps are made
    c. exactly one swap is made
    d. no swaps are made

14. **Student records are stored in ID number order, but accessed by grade point average for a report. Grade point average order is a(n) ______ order.**

    a. imaginary
    b. physical
    c. logical
    d. illogical

15. **With a linked list, every record ______.**

    a. is stored in sequential order
    b. contains a field that holds the address of another record
    c. contains a code that indicates the record's position in an imaginary list
    d. is stored in a physical location that corresponds to a key field

16. **Data stored in a table that can be accessed using row and column numbers is stored as a ______ array.**

    a. single-dimensional
    b. two-dimensional
    c. three-dimensional
    d. nondimensional

17. The Funland Amusement Park charges entrance fees as shown in the following table. The table is stored as an array named `price` in a program that determines ticket price based on two factors—number of tickets purchased and month of the year. A clerk enters the `month` (5 through 9 for May through September), from which 5 is subtracted, so the month value becomes 0 through 4. A clerk also enters the number of `tickets` being purchased; if the number is over 6, it is forced to be 6. One is subtracted from the number of people, so the value is 0 through 5.

	Adjusted month number				
People in party	0	1	2	3	4
0	29.00	34.00	36.00	36.00	29.00
1	28.00	32.00	34.00	34.00	28.00
2	26.00	30.00	32.00	32.00	26.00
3	24.00	26.00	27.00	28.00	25.00
4	23.00	25.00	26.00	27.00	23.00
5	20.00	23.00	24.00	25.00	21.00

What is the price of a ticket for any party purchasing tickets?

a. `price[tickets][month]`
b. `price[month][tickets]`
c. `month[tickets][price]`
d. `tickets[price][month]`

18. Using the same table as in Question 17, where is the ticket price stored for a party of four purchasing tickets in September?

a. `price[4][9]`
b. `price[3][4]`
c. `price[4][3]`
d. `price[9][4]`

19. In a four-dimensional array, you would need to use ________ subscript(s) to access a single item.

a. one
b. two
c. three
d. four

20. In a two-dimensional array, the second subscript needed to access an item refers to the ________.

a. row
b. column
c. page
d. record

FIND THE BUGS

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

1. **This application reads a file containing employee data, including salaries, for 1,000 employees. The salaries are sorted so the median salary in the organization can be displayed.**

```
start
     perform housekeeping()
     perform sortSalaries()
     perform finishUp()
stop

housekeeping()
     declare variables
        inRec
          char name
          num pay
          num x = 0
          num y = 0
          const num SIZE = 1000
          num salary[SIZE]
          num temp
          num midNum
     open files
     read inRec
     while not eof
          salary[SIZE] = pay
          x = x + 1
          read inRec
     endwhile
return

sortSalaries()
     y = 0
     while y > SIZE - 1
          x = 0
          while x < SIZE
               if salary[x] > salary[x] then
                    perform switchValues()
               endif
               x = x + 1
          endwhile
          y = y + 1
     endwhile
return
```

```
switchValues()
     temp = salary[x + 1]
     salary[x] = salary[x]
     salary[x] = temp
return

finishUp()
     midNum = SIZE / 2
     print "Median salary is ", salary[midNum]
     close files
return
```

2. **This application reads student typing test records. The records contain the student's ID number and name, the number of errors on the test, and the number of words typed per minute. Grades are assigned based on the following table:**

	Errors		
Speed	0	1	2 or more
0–30	C	D	F
31–50	C	C	F
51–80	B	C	D
81–100	A	B	C
101 and up	A	A	B

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

housekeeping()
     declare variables
          stuRecord
            num id
            char name
            num errors
            num speed
          num speedArray[0] = 0
          num speedArray[1] = 31
          num speedArray[2] = 51
          num speedArray[3] = 51
          num speedArray[4] = 101
          num x
          num speedCategory
          num grade[0][0] = "C"
          num grade[0][1] = "C"
          num grade[0][2] = "C"
          num grade[1][0] = "C"
          num grade[1][1] = "C"
          num grade[1][2] = "C"
          num grade[2][0] = "C"
          num grade[2][1] = "C"
          num grade[2][2] = "A"
          num grade[3][0] = "A"
          num grade[3][1] = "A"
          num grade[3][2] = "A"
          num grade[4][0] = "A"
          num grade[4][1] = "A"
          num grade[4][2] = "A"
     open files
     read stuRecord
return

mainLoop()
     if errors > 2 then
         errors = 2
     endif
     x = 4
     while x >= 0
       if speed = speedArray[speed]
          speedCategory = x
          x = 0
        endif
        x = x + 1
     endwhile
     print id, name, grade[speedCategory][errors]
     read stuRecord
return

finishUp()
     close files
return
```

EXERCISES

1. Professor Zak allows students to drop the two lowest scores on the ten 100-point quizzes she gives during the semester. Develop the logic for a program that reads student records that contain ID number, last name, first name, and 10 quiz scores. The output lists student ID, name, and total points for the eight highest-scoring quizzes.

2. The Hinner College Foundation holds an annual fundraiser for which the foundation director maintains records. Each record contains a donor name and contribution amount. Assume that there are never more than 300 donors. Develop the logic for a program that sorts the donation amounts in descending order. The output lists the highest five donation amounts.

3. A greeting-card store maintains customer records with data fields for first name, last name, address, and annual purchases in dollars. At the end of the year, the store manager invites the 100 customers with the highest annual purchases to an exclusive sale event. Develop the flowchart or pseudocode that sorts up to 1,000 customer records by annual purchase amount and prints the names and addresses for the top 100 customers.

4. The village of Ringwood has taken a special census. Every census record contains a household ID number, number of occupants, and income. Ringwood has exactly 75 households. Village statisticians are interested in the median household size and the median household income. Develop the logic for a program that determines these figures. (Remember, a list must be sorted before you can determine the median value.)

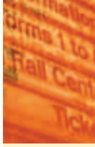
5. The village of Marengo has taken a special census and collected records that each contain a household ID number, number of occupants, and income. The exact number of household records has not yet been determined, but you know that there are fewer than 1,000 households in Marengo. Develop the logic for a program that determines the median household size and the median household income.

6. Create the flowchart or pseudocode that reads a file of 10 employee salaries and prints them from lowest to highest. Use an insertion sort.

7. Create the flowchart or pseudocode that reads a file of 10 employee salaries and prints them from highest to lowest. Use a selection sort.

8. The MidAmerica Bus Company charges fares to passengers based on the number of travel zones they cross. Additionally, discounts are provided for multiple passengers traveling together. Ticket fares are shown in the following table:

	Zones crossed			
Passengers	0	1	2	3
1	7.50	10.00	12.00	12.75
2	14.00	18.50	22.00	23.00
3	20.00	21.00	32.00	33.00
4	25.00	27.50	36.00	37.00



Develop the logic for a program that reads records containing number of passengers and zones crossed. The output is the ticket charge.

9. **In golf, par represents a standard number of strokes a player will need to complete a hole. Instead of using an absolute score, players can compare their scores on a hole to the par figure and determine whether they are above or below par. Families can play nine holes of miniature golf at the Family Fun Miniature Golf Park. So that family members can compete fairly, the course provides a different par for each hole, based on the player's age. The par figures are shown in the following table:**

	Holes								
Age	1	2	3	4	5	6	7	8	9
4 and under	8	8	9	7	5	7	8	5	8
5–7	7	7	8	6	5	6	7	5	6
8–11	6	5	6	5	4	5	5	4	5
12–15	5	4	4	4	3	4	3	3	4
16 and over	4	3	3	3	2	3	2	3	3

a. Develop the logic for a program that reads records containing a player's name, age, and nine-hole score. For each player, print a page that contains the player's name and score on each of the nine holes, with one of the phrases "Over par", "Par", or "Under par" next to each score.

b. Modify the program in Exercise 9a so that, at the end of each golfer's report, the golfer's total score is displayed. Include the figure indicating how many strokes over or under par the player is for the entire course.

10. **Parker's Consulting Services pays its employees an hourly rate based on two criteria—number of years of service and last year's performance rating, which is a whole number, 0 through 5. Employee records contain ID number, last and first names, year hired, and performance score. The salary schedule follows:**

	Performance score					
Years of service	0	1	2	3	4	5
0	8.50	9.00	9.75	10.30	12.00	13.00
1	9.50	10.25	10.95	11.30	13.50	15.25
2	10.50	11.00	12.00	13.00	15.00	17.60
3	11.50	12.25	14.00	14.25	15.70	18.90
4 or more	12.50	13.75	15.25	15.50	17.00	20.00

In addition to the pay rates shown in the table, an employee with more than 10 years of service receives an extra 5 percent per hour for each year over 10. Develop the logic for a program that prints each employee's ID number, name, and correct hourly salary for the current year.

11. The Roadmaster Driving School allows students to sign up for any number of driving lessons. The school allows up to four attempts to pass the driver's license test; if all the attempts are unsuccessful, then the student's tuition is returned. The school maintains an archive containing student records for those who have successfully passed the licensing test over the last 10 years. Each record contains a student ID number, name, number of driving lessons completed, and the number of the attempt on which the student passed the licensing test. The records are stored in alphabetical order by student name. The school administration is interested in examining the correlation between the number of lessons taken and the number of attempts required to pass the test. Develop the logic for a program that would produce a table for the school. Each row represents the number of lessons taken: 0–9, 10–19, 20–29, and 30 or more. Each column represents the number of test attempts in order to pass—1 through 4.

12. The Stevens College Testing Center creates a record each time a student takes a placement test. Students can take a test in any of 12 subject areas: English, Math, Biology, Chemistry, History, Sociology, Psychology, Art, Music, Spanish, German, or Computer Science. Each record contains the date the test was taken, the student's ID number, the test subject area, and a percent score on the test. Records are maintained in the order they are entered as the tests are taken. The college wants a report that lists each of the 12 tests along with a count of the number of students who have received scores in each of the following categories: at least 90 percent, 80 through 89 percent, 70 through 79 percent, and below 70 percent. Develop the logic that produces the report.

DETECTIVE WORK

1. This chapter discussed the idea of using an employee's Social Security number as a key field. Is a Social Security number unique?

2. This chapter examines the bubble, insertion, and selection sorting algorithms. What other named sort processes can you find?

UP FOR DISCUSSION

1. Now that you are becoming comfortable with arrays, you can see that programming is a complex subject. Should all literate people understand how to program? If so, how much programming should they understand?

2. What are language standards? At this point in your study of programming, what do they mean to you?