Although the value of a variable may change during execution of a program, in all our programs so far, a single value has been associated with each variable name at any given time. In this chapter, we will discuss the concept of an array—a collection of variables of the same type and referenced by the same name. We will discuss one-dimensional arrays (lists) at length and focus briefly on two-dimensional arrays (tables). You will learn how to set up and use arrays to accomplish various tasks.

After reading this chapter, you will be able to do the following:

- Declare and use one-dimensional arrays [Section 6.1]
- Manipulate parallel arrays [Section 6.1]
- Use the serial search technique to search an array for a specified element [Section 6.2]
- Use the bubble sort technique to sort an array into a specified order [Section 6.2]
- Use the binary search procedure for locating an item in an array [Section 6.3]
- Use the selection sort procedure for sorting an array [Section 6.3]
- Represent character strings as arrays [Section 6.4]
- Declare and use two-dimensional arrays [Section 6.5]
Chapter 6 • Arrays: Lists and Tables

In the Everyday World

Arrays

Chances are that you frequently use lists in your daily life. The following is a common example:

Shopping List
1. Milk
2. Bread
3. Eggs
4. Butter

Or if you’ve ever done a home improvement project, you might have developed a list such as the following:

Tools Required List
1. Hammer
2. Saw
3. Screwdriver

If you write individual items on separate random pieces of paper, you might get them confused, and end up going to the grocery store for a saw. By presenting a convenient method of organizing and separating your data, a list prevents this from happening.

Sometimes a single list isn’t powerful enough for a given purpose. In this case, we use a table—a collection of related lists—such as the following:

Phone Book
A. Name List  B. Address List  C. Phone List
1. Ellen Cole  1. 341 Totem Dr.  1. 212-555-2368
2. Kim Lee    2. 96 Elm Dr.   2. 212-555-0982
3. Jose Rios  3. 1412 Main St. 3. 212-555-1212

Notice that this table contains three separate (vertical) lists that are all related horizontally. For example, to call someone you would scan down the Name List for the appropriate name, then look across that line to the Phone List to find the corresponding phone number. Could anything be more convenient? Thanks to a little organization and structure, you can easily find what you’re looking for, even if the individual lists are hundreds of items long.

Everyone from farmers (weather tables) to accountants (ledgers) to sports fans (player statistics) uses lists and tables. Here are a couple of computer-related examples: If you use a spreadsheet program, you’re using an electronic table; and if you write a computer program with a lot of data that “goes together,” you might organize it as a list or table called an array.
One-Dimensional Arrays

A one-dimensional array is a list of related data of the same type (for example, integers or strings) referred to by a single variable name with an index number to identify each item. In this section, we will discuss how to set up and manipulate these arrays, and we will present several advantages of their use.

Array Basics

Since an array stores many data values under the same variable name, we must have some way to refer to the individual elements contained within it. Each element is an item in the array and has its own value. To indicate a particular element programming languages follow the array name by an index number enclosed in parentheses or brackets. For example, if the name of an array is Month and the index number is 3, the corresponding array element might be indicated by Month[3].

The name of the array is similar to the name of a variable. For example, we might have an array called Scores that contains the final exam scores for a certain class with 25 students. Therefore, the array would have 25 scores. Each score is an element and the array must indicate which particular element the program refers to at any time. This is done by using the index number. The first element of an array is referred to with the index number 0. An array with 25 elements will have index numbers that range from 0 to 24. This is a significant fact to keep in mind, especially when you manipulate arrays with loops.

An individual element of an array is referred to by writing the array name followed by its index number surrounded by brackets. For example, since the first element of an array has index number 0, the first element of the array Scores would be referred to as Scores[0], and in that same array the second element is Scores[1] and the third is Scores[2]. We read these as ”Scores sub 0”, ”Scores sub 1”, and ”Scores sub 2”. Here, 0, 1, and 2 are called the subscripts—or index numbers—of the array elements.

An array element such as Scores[2] is treated by the program as a single (or simple) variable, and may be used in input, assignment, and output statements in the usual way. Thus, to display the value of the third student’s final exam score, we use the following statement:

```
Write Scores[2]
```

Example 6.1 shows how to enter elements in an array.

Example 6.1 Entering Elements in an Array

If we wanted to input the final exam scores of a class of 25 students using an array named Scores which has 25 elements, we could use a loop as follows:

```
For (K = 0; K < 25; K++)
  Write "Enter score: 
  Input Scores[K]
End For
```

Appendix D covers instructions on how to create and use arrays in RAPTOR.
What Happened?

- On the first pass through the loop, the input prompt is displayed ("Enter score: ") and the Input statement pauses execution, allowing the user to enter the first test score. Because \( K = 0 \), this value is then assigned to the first element of the array Scores, which is \( \text{Scores}[0] \).
- On the second pass through the loop, \( K = 1 \) and the next value input is assigned to \( \text{Scores}[1] \), the second element of Scores.
- On the third pass through the loop, the value input is assigned to \( \text{Scores}[2] \).
- Because this loop began with \( K = 0 \), we only need to go to \( K = 24 \) to complete all 25 passes through the loop and load all 25 values.

After 25 passes through this little loop all the scores have been entered. This is certainly a more efficient way to write code than to type 25 Write and Input statements for 25 different variables! We will see that using arrays not only makes it much easier to load large amounts of data, but also makes manipulating that data easier and more efficient.

Declaring Arrays

Arrays are stored in a computer’s memory in a sequence of consecutive storage locations. As you know, when you declare a variable the computer sets aside a small amount of memory to store the value of that variable. The data type of the variable, the name of the variable, and the value of the variable are one little package and that package is stored in a small amount of memory. When you create an array you tell the computer what the data type is and how many elements the array will contain. Then a space in the computer’s memory is set aside for that array. If an array has five elements, the computer will set aside space with five equal parts, one part for each element of the array. Therefore, each element of the array has the same sized space in computer memory, the same array name, and the same data type. And all the elements are stored in consecutive locations in memory. The index (or subscript) number of the element sets it apart from the other members in the array; the subscript identifies the name of that specific element.

Declaring Arrays in C++ and Visual Basic

The computer must know, prior to the first use of an array, how many storage locations to set aside (or allocate) for that array, so the array declaration statement must include the array name and the size of the memory space needed for that array. In program code, this is done by declaring the array in a statement at the beginning of the program or program module in which it is used. It is also possible to declare arrays without specifying the size and this is used in some cases where the programmer may want the size of the array to be open-ended. The declaration statement varies from language to language. The following shows how an array named Age, consisting of a maximum of six integer values, would be declared in two popular programming languages:

- In C++, the following statement:
  
  ```cpp
  int Age[6]
  ```
allocates six locations referred to as Age[0], Age[1], ..., Age[5].

- In Visual Basic, the following statement:
  ```vbnet
dim Age(6) as integer
  ```
allocates six locations referred to as Age(0), Age(1), ..., Age(5).

Notice that the arrays begin with the subscript 0, and therefore, end with a subscript that is one less than the number of elements in the array.

In this book, for an array of integers, we will use the following pseudocode:

```pseudocode
declare Age[6] as integer
to allocate six locations referred to as Age[0], Age[1], ..., Age[5]. In memory this array would occupy six consecutive storage locations, and assuming that the elements have been assigned the values 5, 10, 15, 20, 25, and 30, the array can be pictured as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
</tr>
</tbody>
</table>
```

It would have been possible to create six integer variables with the values 5, 10, 15, 20, 25, and 30. We could have named them Age5, Age10, Age15, and so forth. But there are many advantages associated with using arrays. Arrays make it easy to manipulate many variables at a time, using only a few lines of code. Example 6.2 shows the declaration and use of an array.

**Example 6.2 When It Rains, It Pours, So Use an Array**

This program uses arrays to find the average monthly rainfall in Sunshine City for the year 2011, after the user has entered the rainfall for each month. First the program computes the average of the monthly rainfall for a year and then displays a list of the months, identified by number (January is Month 1, February is Month 2, and so forth), each month’s rainfall, and the year’s average monthly rainfall. In the program we will show all the variable declarations needed, not just those for the arrays.

```vbnet
1 declare rain[12] as float
2 declare sum as float
3 declare average as float
4 declare k as integer
5 set sum = 0
6 for (k = 0; k < 12; k++)
   7 write "Enter rainfall for month " + (k + 1)
   8 input rain[k]
   9 set sum = sum + rain[k]
10 end for
11 set average = sum/12
12 for (k = 0; k < 12; k++)
   13 write "Rainfall for Month " + (k+1) + " is " + rain[k]
```
What Happened?

- Line 1 declares an array named Rain, which will consist of 12 elements of type Float.
- Lines 2 and 3 declare two floating point variables, Sum and Average. The initial value of Sum is set to 0 on line 5.
- Line 4 declares one integer variable, K. This variable will serve as a counter and will also identify each month when we write the display later.
- Lines 6–10 include the first For loop, where the user inputs the 12 rainfall figures (one for each month) into the array Rain. The loop also totals the value of the rainfall amounts for 12 months. At this point, note the following:
  - The variable K has been initialized to 0 and is set to count to 11. This will result in 12 passes through the loop, which is what we need to load values for 12 months in a year. Since we want to associate each element of the Rain array with its corresponding value of K, we begin the loop with K = 0.
  - However, the first month of the year is Month 1, which is why the Write statement on line 7 asks the user to input the value for Month (K+1). On the first pass, the Write statement asks for Month 1 and on the 12th pass, the Write statement asks for Month 12.
- Line 8 gets the input and stores the value. On the first pass, the user inputs a value for Month 1 since K + 1 = 1 and the value is stored in Rain[0] since K = 0. On the next pass, the user inputs a value for Month 2 since K + 1 is now 2, but the value is stored in the second element of the Rain array, which is Rain[K], or Rain[1]. On the last pass, the user enters a value for Month 12, which is stored in Rain[11].
- Line 11 computes the average value of monthly rainfall amounts for the whole year.
- At this point, the computer has stored rainfall amounts as 12 variables—but all the variables are elements of the array named Rain. The computer also knows the average of the yearly rainfall. All that is left is to tell us the results.
- The second For loop on lines 12–14 displays the month (K + 1) and the amount of rainfall for that month, which was stored in Rain[K] on line 8. This line is pretty important; when we say Write Rain[K], it’s the same as saying Write the value stored in element K of the Rain array.
- Line 15 writes the average of the rainfall for all the months.

If you need to be further convinced about the advantages of using arrays to manipulate a large amount of related data, try to rewrite the pseudocode shown in Example 6.2 without an array. Instead of an array named Rain, declare 12 variables (January, February, March, April, May, June, July, August, September, October,
November, and December) for the values of each month's rainfall amounts. Then rewrite the pseudocode to display each month, its rainfall amount, and the average for the year.

In Example 6.2, the variable we used as a counter to load the array and to identify the number of the month we were referring to at any time was $K$. Initially we set the value of $K$ to 0, which meant that there were certain conditions that had to be met. We could not refer to the Month as simply $K$. Instead we had to use $K + 1$ since there is no Month 0. We also were required to set the test condition so that the loop ends when $K = 11$ to ensure 12 passes. However, we could also have written this program segment with the initial value of $K = 1$. Rewrite this pseudocode with $K = 1$. What changes will you need to make so the program does the exact same thing as shown in Example 6.2? This is the first Self Check question of this section.

Before we move on we will use Example 6.2 to illustrate how the monthly rainfall pseudocode would look in two programming languages. The following program segments show how C++ and Java code is implemented to load the Rain array, display its contents, and calculate and display the average of all entries. For clarity, in both code samples the variable names have been kept exactly the same as the pseudocode.

### When It Rains ... with C++

The C++ code for Example 6.2 is as follows:

```c++
1  int main()
2  {
3    float Sum;
4    float Average;
5    float Rain[12];
6    int K;
7    Sum = 0;
8    Average= 0;
9    for (K = 0; K < 12; K++)
10    {
11      cout << "Enter rainfall for month " << (K + 1) << end1;
12      cin >> Rain[K];
13      Sum = Sum + Rain[K];
14    }
15    Average = Sum/12;
16    for (K=0; K < 12; K++)
17    {
18      cout << "Rainfall for month " << ( K + 1) << " is " << Rain[K] << end1;
19    }
20    cout << "The average monthly rainfall is " << Average << end1;
21    return 0;
22  }
```

---

ISBN 1-256-14455-X

When It Rains ... with Java
The Java code for Example 6.2 is as follows:

```java
public static void main(String args[])
{
    float Sum;
    float Average;
    float[] Rain = new float[12];
    int K;
    Scanner scanner = new Scanner(System.in);
    Sum = 0;
    Average = 0;
    for (K = 0; K < 12; K++)
    {
        System.out.println("Enter rainfall for month" + (K + 1));
        Rain[K] = scanner.nextFloat();
        Sum = (Sum + Rain[K]);
    }
    Average = Sum/12;
    for (K = 0; K < 12; K++)
    {
        System.out.println("Rainfall for month" + (K + 1) + " is "
            + Rain[K]);
    }
    System.out.println("The average monthly rainfall is "
        + Average);
}
```

There are a few noteworthy points:

Most of the syntax for the actual logic of the program is identical for C++ and Java. The main differences are in how the languages handle starting a program and how they retrieve and display data. In C++ the statement `int main()` begins the program while in Java, the program begins with `public static void main(String args[])`.

In C++ the `cout` and `cin` statements allow for input and output. In Java, an object, called a scanner object is created. It allows input to be placed into a buffer. The item in parentheses (in this case, `System.in`) tells the computer where the input is coming from (the keyboard in this case). For example:

```
System.out.println("Enter rainfall for month" + (K + 1));
```
tells the computer to output what is inside the parentheses. The use of `println` does the same thing as `endl` in C++—it tells the computer to move to the next line after executing this line of code. Then the line:

```
Rain[K] = scanner.nextFloat();
```
tells the computer to look at the scanner object, get the next floating point data that is in the buffer and store it in Rain[K].
Parallel Arrays

In programming we often use parallel arrays. These are arrays of the same size in which elements with the same subscript are related. For example, suppose we wanted to modify the program of Example 6.2 to find the average monthly rainfall and snowfall. If we store the snowfall figures for each month in an array named Snow, then Rain and Snow would be parallel arrays. For each K, Rain[K] and Snow[K] would refer to the same month so they are related data items. Example 6.3 illustrates this idea further.

Example 6.3 You’ll Be Sold on Parallel Arrays

This program segment inputs the names of salespersons and their total sales for the month into two parallel arrays (Names and Sales) and determines which salesperson has the greatest sales (Max). Figure 6.1 shows the flowchart for this pseudocode. It’s helpful to walk through the flowchart to understand the logic used to solve this problem visually.

```plaintext
1  Declare Names[100] As String
2  Declare Sales[100] As Float
3  Set Max = 0
4  Set K = 0
5  Write "Enter a salesperson's name and monthly sales."
6  Write "Enter *, 0 when done."
7  Input Names[K]
8  Input Sales[K]
9  While Names[K] != "*"
10   If Sales[K] > Max Then
11     Set Index = K
12     Set Max = Sales[Index]
13   End If
14   Set K = K + 1
15   Write "Enter name and sales (enter *, 0 when done)."
16   Input Names[K]
17   Input Sales[K]
18  End While
19  Write "Maximum sales for the month: " + Max
20  Write "Salesperson: " + Names[Index]
```

What Happened?

- In this program segment we do not use a For loop to input the data because the number of salespersons may vary from run to run. Instead, we use a sentinel-controlled loop with an asterisk (*) and a 0 as the sentinels. The user indicates that there are no more salespeople to enter by entering the asterisk (*) for the name and 0 for the amount.
- Lines 1 and 2 declare two arrays. Names is an array of String elements and Sales is an array of Floats. You may wonder why these arrays have been declared to have 100 elements each, when the program is going to allow the user to enter as many salespeople as necessary. In some programming
Arrays: Lists and Tables

Figure 6.1 Flowchart for loading parallel arrays

Enter

Declare Names[100] As String
Declare Sales[100] As Float

Set Max = 0
Set K = 0

Write “Enter a salesperson’s name and monthly sales.”
Write “Enter *, 0 when done.”
Input Names[K], Sales[K]

Names[K] !="*"?

Yes

Sales[K] > Max?

Yes

Set Index = K
Set Max = Sales[Index]

No

Set K = K +1

Write “Enter a salesperson’s name and monthly sales.”
Write “Enter *, 0 when done.”
Input Names[K], Sales[K]

No

Write “Maximum sales for month” + Max
Write “Salesperson: “ + Names[Index]

Exit
languages you may have to specify the size of the array. It's better to declare an array for a larger number of elements than you plan to use. In this program segment the maximum number of salespeople that can be entered is 100, but we assume that is plenty. If there are only 38 salespeople, the rest of the elements of the array are simply unused.

- Let's look at what happens on lines 5–8 in more detail. Line 5 simply requests the input (a salesperson’s name and his/her monthly sales amount). Line 6 explains to the user how to finish entering data. Lines 7 and 8 are of some special interest. When a user enters a name and an amount, the first value entered is stored in the *Names* array and the second value entered is stored in the *Sales* array. This is very important! The information must be entered in the correct order. Imagine that the fifth entry was for a saleswoman named Josephine Jones whose sales totaled $4,283.51. If the information was entered in the wrong order, *Names[4]* would store 4283.51. This would not be the correct name, but a computer doesn’t care what text is entered into a string so a string of numbers would be a legitimate entry. However, when the user entered “Jones” into the *Float* array, *Sales*, at best the program would simply halt or give an error message or, worse, it would crash.

- Line 9 begins the loop to do the work of this program. It continues until the asterisk is entered.

- Line 10 checks to see if the value of the current salesperson's monthly sales is greater than the maximum value to that point (*Max*). The variable *Max* holds the maximum value, so as all the entries are made, *Max* will continually change every time an entry is made with a larger sales amount.

- Line 11 sets the variable named *Index* equal to the value of *K*. *K* is just an integer. If the value in *Sales[K]* is greater than the *Max* value, we want the new *Max* value to be the value of whatever *Sales[K]* is at this point in the program. Let’s say for example, that *Sales[3]* is the high value at one point in the program. We need a way to associate that value with the salesperson who sold that amount. The variable *Index* keeps track of that. Regardless of what the value of *K* becomes as the program proceeds, until there is a higher value for *Sales[K]*, the person associated with *Names[Index]* (in this case, *Names[3]*) remains the high seller.

- In line 12, if the current salesperson has sold more than whatever value was in *Max*, the new *Max* is set equal to sales amount of the current salesperson. If the current salesperson has not sold more than the value of *Max*, nothing changes.

- Lines 14–17 increment the counter and get another set of data. The loop continues until the user ends it by entering the two sentinel values.

- Line 18 simply ends the *While* loop and lines 19 and 20 display the results.

In Example 6.3, what would happen if a salesperson had sold nothing for that month? The values entered would be the salesperson’s name for the *Name* array and 0 for the *Sales* array. Would this create a problem? No, because the test condition
tests only for the salesperson's name. As long as the salesperson who sold nothing was not named "**", the program would continue.

And what would happen if the user decided to end the input by entering "**" for the Name array but 23 instead of 0 for the Sales array? Would the program end? Yes, it would because the test condition on line 9 (Names[K] != "**") would no longer be true, the While loop would not execute, and the value 23 would not be compared to other values.

You may be wondering why we ask the user to enter a 0 for the sales amount if the program segment is only interested in the salesperson's name. When this program is coded and run, there are two inputs required for each entry. The user must enter something for the sales amount. We could just have easily have specified "Enter *, -8,983 when done" on lines 6 and 15, but we picked 0 since it makes sense that a normal monthly sales figure wouldn't be nothing. However, we have to specify some number since Sales is an array of numbers.

Some Advantages of Using Arrays

We close this section by describing some of the benefits of using arrays. As you have already seen, arrays can reduce the number of variable names needed in a program because we can use a single array instead of a collection of simple variables to store related data. Also arrays can help create more efficient programs. Once data are entered into an array, that data can be processed many times without having to be input again. Example 6.4 illustrates this point.

Example 6.4 Arrays Save You Time and Effort

Professor Merlin has asked you to help him. He has 100 students in his four classes but he is not sure that all of them took his last exam. He wants to average the grades for his last exam in four sections of his medieval literature course and then determine how many students scored above the average and how many scored below. Without arrays you would have to enter all the test scores, find their average, and then enter them again to determine how many exceed the average. But you know how to use arrays, so you won’t need to enter the input a second time. The following pseudocode does the job. We assume variables have been declared as given:

- Integer variables: Sum, Count1, Count2, and K
- Float variables: Score and Average

1 Declare Medieval[100] As Float
2 Set Sum = 0
3 Set Count1 = 0
4 Write "Enter a test score (or 999 to quit): "
5 Input Score
6 While Score != 999
7 Set Medieval[Count1] = Score
8 Set Count1 = Count1 + 1
9 Set Sum = Sum + Score
10 Write "Enter another score or 999 to quit: 
11 Input Score
12 End While
13 Set Average = Sum/Count1
14 Set Count2 = 0
15 Set K = 0
16 While K < Count1
17 If Medieval[K] > Average Then
18 Set Count2 = Count2 + 1
19 End If
20 Set K = K + 1
21 End While
22 Write "The average is:" + Average
23 Write "The number of scores above the average is: " + Count2
24 Write "The number of scores below the average is: " + (Count1 – Count2)

What Happened?

In the While loop, which inputs the scores, the Count1 variable serves as a subscript for the array Medieval and also counts the number of scores input. Since we don’t know exactly how many students took the exam, we must use a sentinel-controlled loop here. However, when it’s time to determine the number of items above the average (Count2), we know the number of items that have been input. A second While loop is used (lines 15–21) with a limit value of one less than Count1 to determine the number of scores above and below the average.

Let’s think about this limit value for a moment. The value of Count1 on line 12, at the end of the While loop, is equal to the number of scores entered. For example, since Count1 is initialized to 0 on line 3, each time Professor Merlin enters a score for a student, Count1 has the value of 1 less than the number of students up to that time. This is done so that the value of the first score will be stored in Medieval[0]. However, Count1 is incremented on the line following the score input (line 8). Therefore, when all the scores have been entered and the While loop is exited, Count1 will have the same value as the number of scores that have been entered. For example, if the Professor has 23 students in a class, he will store those students’ scores in Medieval[0] through Medieval[22] but, on exiting the While loop, Count1 will equal 23.

The While loop that begins on line 16 needs to compare each score (each element of the Medieval array) with the value of Average. Therefore, if there are 23 scores entered in Medieval[0] through Medieval[22], the loop must make 23 passes. The counter for this loop, K, begins at 0 and, since it increments by 1 for each pass, by the time K has reached 22, it will have completed 23 passes. This is why we set the limit condition of this loop on line 16 to K < Count1.

Another benefit of using arrays is that they help create programs that can be used with greater generality. If we use 30 simple variables to hold the values for 30 test scores, we can only have 30 test scores. If five students register late for a class and
we need variables for 35 students, we have to declare five more variables. However, because we do not have to use all the elements that were allocated to an array when it was declared, arrays give us more flexibility, as shown in Example 6.5. We could initialize an array for scores with space for 50 elements, even if we only use 30, or later in the semester, use 35. The unused spaces in the computer’s memory are set aside for a value, if one is entered.

**Example 6.5 Arrays Make Programming Easy and Concise**

Professor Merlin received a list of all his students from the head of his department. The names are listed in alphabetical order from A to Z. But Professor Merlin is not a forward-thinking professor. He likes his student class lists to appear in reverse alphabetical order from Z to A. He asks you to write a program to allow him to input a list of names and display them in reverse alphabetical order. This is easy to do using arrays, even if the number of names to be input is unknown at the time the program is written.

```
1 Declare Names[100] As String
2 Set Count = 0
3 Write "Enter a name. (Enter * to quit.)"
4 Input TempName
5 While TempName != "*"
6   Set Names[Count] = TempName
7   Set Count = Count + 1
8   Write "Enter a name. (Enter * to quit.)"
9   Input TempName
10 End While
11 Set K = Count - 1
12 While K >= 0
13   Write Names[K]
14   Set K = K - 1
15 End While
```

**What Happened?**

This program segment inputs the list of names into the array Names and then displays the elements of that array in reverse order by “stepping backward” through a While loop whose control variable (K) is also the array subscript. The purpose of the variable TempName is to hold the string entered by the user temporarily. If that string is really a name and not the sentinel value "*", the first While loop is entered and the string is assigned to the next array element.

Note that when the first While loop ends, the value of Count is one greater than the highest subscript of the Names array. This is why we begin the second While loop with the new counter, K, set equal to the value of Count – 1.
Self Check for Section 6.1

Self Check Questions 6.1 and 6.2 refer to Example 6.2.

6.1 Rewrite the pseudocode to load the array, Rain[], with 12 rainfall amounts using K = 1 as the initial value of K.

6.2 Rewrite the pseudocode of Example 6.2 while loops instead of For loops.

In Self Check Questions 6.3 and 6.4, what is displayed when code corresponding to the given pseudocode is executed?

    Declare K As Integer
    Set K = 1
    While K <= 3
        Write A[2 * K]
        Set K = K + 1
    End While

6.4 In this exercise, Letter is an array of characters and the characters that have been input are F, R, O, D, O.
    Declare Letter[5] As Character
    Declare J As Integer
    For (J = 0; J <= 4; J++)
        Input Letter[J]
    End For
    For (J = 0; J <= 4; J + 2)
        Write Letter[J]
    End For

6.5 The following program segment is supposed to find the average of the numbers input. It contains one error. Find and correct it.
    Declare Avg[10] As Integer
    Declare Sum, K As Integer
    Set Sum = 0
    For (K = 0; K <= 9; K++)
        Input X[K]
        Set Sum = Sum + X[K]
    End For
    Set Average = Sum/10

6.6 Write a program segment that inputs 20 numbers in increasing order and displays them in reverse order.

6.7 State two advantages of using arrays instead of a collection of simple (unsubscripted) variables.
6.2 Searching and Sorting Arrays

When we write programs we often find that we need to search a one-dimensional array (or list) to locate a given item or to sort it in a specified order. Consequently, there are many algorithms available to perform each of these tasks. In this section, we will present simple techniques for searching and sorting an array. In Section 6.3, we will present several other search and sort techniques.

The Serial Search Technique

Suppose you have arrived at the airport to meet your best friend who is flying in from her overseas vacation. You know her flight number but not the arrival time or gate, so you consult the posted flight information, which is given in the following tabular form:

<table>
<thead>
<tr>
<th>Flight</th>
<th>Origin</th>
<th>Time</th>
<th>Gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>Kansas City</td>
<td>4:15 p.m.</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>St. Louis</td>
<td>5:05 p.m.</td>
<td>4</td>
</tr>
<tr>
<td>96</td>
<td>London</td>
<td>5:23 p.m.</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>Dubuque</td>
<td>5:30 p.m.</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

To find the arrival time and gate of your friend’s flight, you scan the leftmost column until you locate her flight number and then you move across that row to read the corresponding time and gate in the last two columns.

In computer lingo, you have performed a table lookup. In data processing terminology, the item you were seeking (your friend’s flight number) is called the search key. The search key is a particular item in the list of all such items (all the flight numbers); we call this list the table keys. In general, the data in the table are called the table values. The way in which you looked for a desired flight number, checking the numbers in the order listed, makes this a serial search.

Basic Steps in a Serial Search

1. Load the table: this is where we input the data in the table, often from a file (see Chapter 8), into parallel arrays, one for each column of the table.
2. Search the array of table keys: here we compare the search key to the elements of this array, one by one, until a match occurs or the end of the array is reached.
3. Display the search key and corresponding table values: or if the search key is not found in the array, display a message to this effect.

Use a Flag to Indicate a Successful Search

In Step 2 of the search procedure described above, we loop through the array of table keys to see if any element is identical to the given search key. There can be
two results: the search will be successful—the item we are seeking (the search key) will match one of the table keys—or the search will fail—no match will occur.

When we exit the loop and carry out Step 3 of the procedure, we must know which of these two possibilities has occurred so that the appropriate message can be displayed. An elegant way to indicate whether or not the search was successful is to use a variable known as a flag.

A flag is a variable that takes on one of two values, typically 0 and 1. It's used to indicate whether or not a certain action has taken place. Usually a value of 0 indicates that the action has not occurred; a value of 1 indicates that it has occurred. In a serial search we set the flag equal to 0 before we begin the search. If a match occurs within the search loop we change the flag's value to 1. This means that if the search loop never finds the item we are looking for, the flag's value will never change and it will be 0 at the end. But if the search loop finds what we are looking for, the flag is changed to 1. Thus, when we exit the loop the value of the flag can be used to verify the success or failure of the search and to display the proper message.

### Pseudocode for a Serial Search

Consider a list of table keys contained in an array named KeyData. The array has \( N \) elements. We want to find an item called Key, which is one of the elements in the array. The flowchart shown in Figure 6.2 pictures the logic used in this serial search. In this flowchart, the variable Found is the flag for the search and Index is the array subscript.

The pseudocode for performing this serial search follows:

```plaintext
//Set the subscript (Index) of the current table key equal to 0
Set Index = 0
//Set a flag (Found) equal to 0
Set Found = 0
While (Found == 0) AND (Index < N)
  If KeyData[Index] == Key Then
    Set Found = 1
  End If
  //Increment Index by 1
  Set Index = Index + 1
End While
If Found == 1 Then
  //Display the array element with subscript of Index-1
  Write KeyData[Index - 1]
Else
  //Display a message that the search was unsuccessful
  Write "The item you are searching for was not found."
End If
```

In this pseudocode the variable Index is used within the loop to hold the current array subscript. The variable Found is a flag that indicates whether or not the search has been successful. Note that we could, alternatively, have used a For loop with the counter, Index, running from 0 to \( N - 1 \). When we exit from the While
loop, we check the value of Found. A value of 1 indicates that the search was successful and that the item, Key, has been found among the table keys. The value of Index – 1 tells us the location of Key in KeyData because Index – 1 is the subscript that identifies the element in the array. If the value of Found is 0 when we exit the loop, we know that the search item, Key, was not found and we display this fact. Example 6.6 provides a particular instance of the serial search technique.

**Example 6.6 A Serial Search with Parallel Arrays**

This program segment displays the test score for a particular student when the student’s identification number is input by the user. It searches an array called IDNumbers, which consists of identification numbers for the ID number input (IDKey). Then it does the following:

- Displays the corresponding student name and test score contained in two parallel arrays named Names and Scores, if the number IDKey is found in the IDNumbers array.

![Flowchart for a serial search](image-url)
Displays an appropriate message if the IDKey is not found in the IDNumbers array.

We assume that the arrays IDNumbers, Names, and Scores have already been declared and loaded with the necessary data. We also assume that the number of elements in each of these parallel arrays is N and that the variables IDKey, Index, and Found have been declared as Integer variables.

```
1  Write "Enter a student ID number: "
2  Input IDKey
3  Set Index = 0
4  Set Found = 0
5  While (Found == 0) AND (Index < N)
6    If IDNumbers[Index] == IDKey Then
7      Set Found = 1
8    End If
9    Set Index = Index + 1
10  End While
11  If Found == 0 Then
12    Write "Student ID not found"
13  Else
14    Write "ID Number: " + IDKey
15    Write "Student name: " + Names[Index - 1]
16    Write "Test score: " + Scores[Index - 1]
17  End If
```

What Happened?

- Lines 1 and 2 of this program segment prompt for and input the ID number for the student we are seeking. Notice that the ID number of the student we seek is a plain variable (IDKey).
- The variable (Index), which is initialized on line 3, is used as a counter in the While loop and also as the index (or subscript) of the arrays, as we search through the arrays.
- The variable (Found), which is initialized on line 4, is used as our flag. If we find the ID number of the student we want in the While loop, Found is set to 1 (on line 7) so we know what message to display later in the program.
- Notice that line 4 sets the value of Found to 0 while the statement on line 5 compares the value of Found to 0. This is an example of how both the assignment and the comparison operators are used in a program.

Lines 5–10 contain the While loop that does several things as follows:

- The conditions under which the loop will continue are set up here. Line 5 contains a compound condition with an AND operator. You will remember that this means both parts of the condition must be true for the loop to continue. This line says that as long as the IDKey has not been found (Found will still contain the value of 0) AND the Index is less than the number of elements in the ID numbers array (Index < N), continue doing the loop. As soon as one of those conditions is not true, the loop ends. This allows the loop to end as soon as a match is found.
The If-Then clause on lines 6–8 checks to see if the ID number of the student in the array whose subscript is Index matches the IDKey. If that is the case, the flag (Found) is set to 1. There is no Else clause here because if there is no match, nothing needs to be done.

Line 9 increments the counter (Index), which counts the number of times through the loop to ensure that we only loop through as many times as there are students in the array and sets the correct subscript for the comparison, which begins at the next pass through the loop.

Line 10 ends the While loop. At this point there are two possibilities. In one case, the loop has gone around N times so all ID numbers in the IDNumbers array have been checked and no match has been found. In this case, Found still is 0. The only other possibility is that a match has been discovered and Found has the value of 1. The value of Index is one higher than it was when the match was found.

Lines 11–17 display the appropriate message depending on whether or not a match has been found.

The Bubble Sort Technique

When we sort data we arrange it in some prescribed order. For numbers, the “prescribed order” would normally be ascending (from smallest to largest) or descending (from largest to smallest). For names, the prescribed order is usually alphabetical.

As long as the number of items to be sorted is relatively small (say, fewer than 100), the bubble sort algorithm provides a reasonably quick and simple way to sort. To apply this technique, we make several sweeps (or passes) through the data, and on each pass we compare all adjacent pairs of data items and interchange the data in an adjacent pair if they are not already in the proper order. We continue making passes until no interchanges are necessary in an entire pass, which indicates that the data is sorted.

Swapping Values

If you brought a peanut butter sandwich for lunch and your friend brought a cheese sandwich, the two of you could swap lunches in one motion. You hold out your sandwich, your friend holds out his; he takes your sandwich and you take his. For even a fraction of a second, one of you holds two sandwiches and the other holds none. A computer cannot make this kind of exchange. In a computer each value is stored in its own location in memory. If you put a cheese sandwich in the location that previously held a peanut butter sandwich, the peanut butter sandwich will be replaced by the cheese sandwich and disappear into cyberspace. So before we discuss the bubble sort algorithm, first we must understand how the computer swaps the values of two items.

How would you change the contents of two different sized boxes if each box can only contain one item at a time? We start with Blue in Box1 and White in Box2 and want to end up with White in Box1 and Blue in Box2. But if we put Blue into Box2, we have lost the value of White. That’s the problem a computer programmer faces. So we create an empty temporary storage space to save the contents of Box1 while we change its value to White, as shown in Figure 6.3.
Example 6.7 shows the pseudocode for the box shuffle. We’ll name our variables `Box1` and `Box2` and assume that `Box1` initially contains the string “Blue” and `Box2` initially contains “White”. We will also use a variable named `Temp` which initially contains a blank space.

**Example 6.7 Swapping Boxes**

```
Set Temp = " "  /*comment: the Temp variable is initialized to contain just a blank space*/
Set Temp = Box1  /*comment: now the value "Blue" is in two places—it’s still in Box1 and also in Temp*/
Set Box1 = Box2  /*comment: now the value of Box1 is "White" and "White" is still in Box2 but we have held the value of "Blue" in Temp*/
Set Box2 = Temp  /*comment: the value of Box2 has been replaced by "Blue" and the swap is complete—Box1 and Box2 have exchanged values. The variable Temp still has "Blue" but we don’t care since Temp is not used for anything else.*/
```

The method used to interchange the values of two variables that was demonstrated in Figure 6.3 and in Example 6.7 is called a **swap routine**. It will be used many times from now on in this book and will become one of the most important tools.
you will use as you continue to write programs. If you spend a few extra minutes now to ensure that you understand how the swap routine works, you will be glad you did!

**Using the Bubble Sort Algorithm**

To illustrate the bubble sort, first we will do a simple example by hand. Figure 6.4 demonstrates how to sort five numbers in a data set so they are listed from smallest to largest. The numbers are initially stored as follows:

```
9  13  5  8  6
```

A computer can only do one thing at a time, so first it must compare two numbers and decide if they need to switch places. After it makes this decision, it can move on to the next pair. This is why, in this example, there are three passes and each pass has four steps. Figure 6.4 shows, for each pass, the data at the start of the pass on the left and the results of the four comparisons in the next four steps.

![Figure 6.4](image_url)

**Figure 6.4** The bubble sort

**First Pass**

```
| 9 | 13 | 5 | 8 | 6 |
```

**Second Pass**

```
| 9 | 5 | 13 | 8 | 6 |
```

**Third Pass**

```
| 5 | 9 | 8 | 6 | 13 |
```

**Fourth Pass**

```
| 5 | 6 | 8 | 9 | 13 |
```

No interchanges take place; the numbers are sorted.
columns. If an interchange takes place, the arrows cross, indicating which items have been swapped.

In the first pass (the top row of Figure 6.4) the first number (9) is compared with the next number (13) to see if the first is larger than the second. Since 9 is less than 13, no swap is made. Then the second number (13) is compared to the third number (5). In this case, because 5 is less than 13, the numbers are switched. Remember—the computer can only do one thing at a time. Each time you see a switch in Figure 6.4, a swap routine, using a temporary variable as a holding place, has been used. Further, a computer can’t think, as you might, “Well, 5 is also smaller than 9 so I should switch the 5 and 9.” It will do that on the next pass, but it can’t do it yet. After the 5 and 13 are swapped, the number in the third place (which is now 13) is compared to the number in the fourth place (8). Once again, since 8 is less than 13, they are swapped. Now 13 is in fourth place and 13 is compared to the number in the last place (6). Because 13 is larger than 6, they are swapped. One pass is complete.

On the second pass, we return to the first number. It is still a 9. But now the second number is a 5, so when these two are compared, the 5 moves to first place. Now the pass continues as before. By the end of the second pass you will see that the second-largest number has moved down to fourth place.

The bubble sort gets its name from the fact that the larger numbers “sink” to the bottom (the end) of the list and the smaller ones “bubble” to the top. After the first pass the largest number will be at the bottom of the list; after the second pass the second largest will be next to last; and so forth. In this example, only three passes are needed to sort the numbers. This is simply a result of the way the original list was given. If the list had been written differently, we might have needed up to four passes to sort. In general, to sort N items, it will take at most N – 1 passes through the list to sort them (and one additional pass to determine that they are sorted).

The bubble sort of an array named A of N numbers in ascending order is described in the following general pseudocode. Figure 6.5 shows a flowchart of the bubble sort logic.

```plaintext
While the array A is not sorted
    For (K = 0; K < N – 1; K++)
            Interchange A[K] and A[K + 1]
        End If
    End For
End While
```

This pseudocode is somewhat vague. We cannot just say “interchange two values” because as we have seen, we need to use the swap routine. In Figure 6.4, we saw that while the maximum number of passes needed to sort N numbers is N – 1, some data can be sorted in fewer passes. If we had a list of 100 numbers to sort and only 2 numbers were out of order, it would be inefficient to allow the loop to continue for 99 passes. Therefore, we should include some way to indicate that the data is
sorted and that the loop can end. We should refine the previous pseudocode as follows:

1. We use the swap routine to interchange array elements $A[K]$ and $A[K+1]$ by temporarily copying one of them to a temporary location and then swapping their values:
   
   Set $Temp = A[K]$
   Set $A[K+1] = Temp$

   Try it by executing these statements with $A[K] = 3$ and $A[K+1] = 5$.

2. To determine when the list is sorted, we borrow a technique from the serial search algorithm given earlier in this section and use a flag. In the bubble sort algorithm, a flag value of 0 indicates that during the latest pass, an interchange took place. Therefore, we initialize the flag to 0 and continue to reenter the while loop as long as its value remains 0. Once inside the
loop, we set the flag equal to 1 and change it back to 0 if an interchange takes place. If no interchange takes place (meaning the data is sorted), the flag remains 1 and the loop is exited.

Example 6.8 gives detailed pseudocode for a typical bubble sort.

**Example 6.8 Input Numbers in Any Order, See Them Sorted!**

This program segment implements the bubble sort procedure described above. It inputs numbers from the user, sorts them in descending order, and then displays the results. Imagine how convenient this would be for Professor Merlin. He could enter his students' test scores in any order, and in a few seconds, all the scores could be sorted. This would help him decide how much of a curve to give!

In this program segment, the numbers are input into an array named `TestScores`. Each test score entered is stored in a variable named `OneScore`. The variable, `Flag`, will indicate when enough passes have been made to sort the whole array, regardless of whether the array is full (with 100 elements) or partially full.

After you read the pseudocode and the explanation, pick some values for test scores of your own and walk through the pseudocode to verify that it works and that you understand what each line does.

```plaintext
1  Declare TestScores[100] As Float
2  Declare Count As Integer
3  Declare Flag As Integer
4  Declare K As Integer
5  Declare OneScore As Float
6  Declare Temp As Float
7  Write "Enter a test score; enter –9999 when done: "
8  Input OneScore
9  Set Count = 0
10  While OneScore != –9999
11    Set TestScores[Count] = OneScore
12    Set Count = Count + 1
13    Write "Enter a test score; enter –9999 when done: "
14    Input OneScore
15  End While(OneScore)
16  Set Flag = 0
17  While Flag == 0
18    Set Flag = 1
19    Set K = 0
20    While K <= (Count - 2)
21      If TestScores[K] < TestScores[K + 1] Then
22        Set Temp = TestScores[K]
23        Set TestScores[K] = TestScores[K + 1]
24        Set TestScores[K + 1] = Temp
25        Set Flag = 0
26      End If
27    Set K = K + 1
28  End While(K)
```
29  End While(Flag)
30  Write "Sorted list..."
31  Set K = 0
32  While K <= (Count – 1)
33    Write TestScores[K]
34    Set K = K + 1
35  End While(K)

What Happened?

- Lines 1–15 declare and load the array, TestScores, with up to 100 scores. The counter, Count, is initialized to 0. The first While loop (lines 10–15) accomplishes the input of all the scores and counts how many scores are entered by the user. The array, TestScores, was originally declared to have 100 elements, so this program segment can only sort up to 100 numbers. However, it will stop asking for numbers when the user enters the sentinel value of –9999, so Professor Merlin could use this program for a class of 20 or 50 or any number up to 100.

- At the end of this loop, the value of Count is the same as the number of scores entered into the array. In other words, if Professor Merlin has entered 45 test scores, Count also equals 45. Can you see why? On line 8 the first score is entered and on line 9 Count is set equal to 0. Then, within the loop (lines 10–15), Count is incremented after the value of OneScore is stored in TestScores[0]. Now there is one element in the array and Count equals 1. In effect, Count "catches up" with the number of elements in the TestScores array. However, later in the program we want Count to keep track of the subscripts of the elements in the array. For example, if there are 45 elements in the array, the subscripts on these elements are, as you know, 0 through 44. We must remember this as we continue with the program.

- Line 16 begins the interesting part. The Flag is set equal to 0. The next While loop (lines 17–28) implements the general bubble sort procedure given prior to this example.

Let’s walk through lines 17–28 with a few test values. Assume you have input the following four test scores: 82, 77, 98, and 85.

What Will Happen?

Line 17 sets up the condition for the outer loop (the While loop) of the sorting process. When Flag no longer is 0, this loop will end. Line 18 changes the value of Flag to 1. Line 20 begins the inner loop. It begins with a variable, K, previously set equal to 0. K will be incremented by 1 for each pass through this loop. The test condition says to continue the loop until K = (Count – 2). Remember that Count is equal to the number of elements in the array, but the indices of the array begin at 0 and go up through the value of Count – 1. The bubble sort compares one element in the array to the next element so the last comparison is between the next-to-last element and the last element. This is why we only need K to go up to the next-to-last element, Count – 2. In our example, we have four elements in the array so the loop will end after three passes when K = 2.
On the first pass, line 21 tests to see if \texttt{TestScores[0]} is less than \texttt{TestScores[1]}. In this example \texttt{TestScores[0]} = 82 and \texttt{TestScores[1]} = 77 so the first element is not less than the second and no swap is made. Now (line 27) \( K = 1 \).

On the second pass through this loop, we compare \texttt{TestScores[1]} (77) with \texttt{TestScores[2]} (98). Since 77 is less than 98, the swap is made (lines 22–24) and \texttt{Flag} is set to 0. At this point, the value of \texttt{Flag} does not come into play; it will be significant later on. Now \texttt{TestScores[1]} contains the value 98 and \texttt{TestScores[2]} contains the value 77.

Now \( K = 2 \) and the loop compares \texttt{TestScores[2]} (77) with \texttt{TestScores[3]} (85). Since 77 is less than 85, these values are swapped. \texttt{Flag} is still equal to 0. However, \( K \) is now 3, which is greater than the test condition so the loop ends. At this point the array contains the following values:

\begin{verbatim}
TestScores[0] = 82
TestScores[1] = 98
TestScores[2] = 85
TestScores[3] = 77
\end{verbatim}

Notice that the lowest value has “bubbled” down to last place after all the comparisons have been made.

Control now goes back to line 17. The value of \texttt{Flag} is checked, and since it’s still 0, the inner loop begins again. Line 18 sets \texttt{Flag} to 1 and lines 20–28 compare all the values again. In this iteration of the outer loop, the inner loop swaps \texttt{TestScores[0]} with \texttt{TestScores[1]} and then \texttt{TestScores[1]} with \texttt{TestScores[2]}. \texttt{Flag} is set back to 0. When \( K \) becomes greater than 2, the inner loop ends again and the array now looks as follows:

\begin{verbatim}
TestScores[0] = 98
TestScores[1] = 85
TestScores[2] = 82
TestScores[3] = 77
\end{verbatim}

Once again control goes back to line 17. Since \texttt{Flag} is 0, the outer loop begins again. \texttt{Flag} is now set to 1. This time no swaps will be made in the inner loop. This means that the \texttt{Flag} will never be set back to 0 because the \texttt{If-Then} statements on lines 21–26 are never executed.

Now when control goes back to line 17, the test condition is false. \texttt{Flag} is not equal to 0, so the outer loop ends and all the values have been sorted. Control passes to line 30 and then to the little \texttt{While} loop at the end, which displays all the sorted values.

\section*{Other Sorts of Sorting}

We can use the pseudocode presented in Example 6.8, with little or no modification, to perform several related sorting tasks as follows:

- Almost identical pseudocode can be used to sort names alphabetically. Of course in this case, the array \texttt{TestScores} must be declared to be an array of strings, which we would probably rename something like \texttt{Names}, and \texttt{OneScore} must be a string variable, which would probably be renamed \texttt{OneName}.  


• Virtually the same pseudocode can also be used to sort numbers in ascending order. The only modification needed here is to change the first line of the If statement to read as follows:

If Names[K] > Names[K+1] Then

Self Check for Section 6.2

6.8 What is the output if the following program segment is coded and run?

Declare Bird As String
Declare Cat As String
Declare Temp As String
Set Bird = "black"
Set Cat = "green"
Set Temp = Bird
Set Bird = Cat
Set Cat = Temp
Write "My bird has " + Bird + "feathers."
Write "My cat is " + Cat + "."

6.9 Given the following values for three Character variables, what will be stored in each variable after the swap routine in the pseudocode shown below has been coded and run given that:

X = "X"  Y = "Y"  Z = "Z"
Set Z = X
Set X = Y
Set Y = Z

6.10 Indicate whether each of the following statements is true or false:

a. T  F  The serial search requires that the list of table keys be arranged in order.

b. T  F  The serial search can only be used with a list of numbers.

6.11 Write a program that searches an array of 100 names, Client, for the name “Smith.” If the name is found, the program should display “FOUND”; if not, it should display “NOT FOUND.”

6.12 How many interchanges take place in sorting the numbers 3, 2, 1 in ascending order using a bubble sort?

6.13 Write a program that sorts an array of 100 names, Client, in alphabetical order using the bubble sort method.

6.3 More about Searching and Sorting

In the previous section, we described simple methods for searching and sorting arrays. However, for arrays that contain huge amounts of data, these methods are not very efficient. In this section, we will introduce more efficient searching and sorting techniques. Sorting and searching are extremely important concepts. They form the backbone of relational database management systems that are used in every aspect of commerce and government throughout the world today.
Binary Search

The **binary search** method is a good way to search a large amount of data for a particular item, which is called the search key. It is considerably more efficient than the serial search technique, which was discussed in Section 6.2. However, a binary search requires that the **table keys**, the array of data to be searched, is in numerical or alphabetical order.

To illustrate how the binary search method works, let's suppose you want to look up a certain word (the **target word**) in a dictionary. If you used a serial search to find it, starting with the first page and going through the dictionary word by word, it might take hours or even days. A more reasonable approach would be as follows:

1. Open the dictionary to the target word’s approximate location.
2. Check the target word against an entry on that page to determine if you have gone too far or not far enough.
3. Repeat Steps 1 and 2 until you have located the target word.

This example demonstrates the basic idea underlying the binary search procedure. To carry out a binary search, first we compare the **search key** (the **target**) to the table key midway through the given array. Because the array data is ordered, we can then determine in which **half** of the array the search key lies. We now compare the search key to the table key in the middle of this half, and in so doing are able to determine in which **quarter** of the array the search key is located. Then we look at the middle entry in this quarter, and so forth, continuing this process until we find the search key.

The following general pseudocode performs a binary search, assuming that the array named **Array** is sorted in ascending order. The variable **Key** represents the item sought and the array **Array** is a given array of a certain number of table keys.

Recall the following:

- The function $\text{Int}(X)$ produces the integer obtained by discarding the fractional part of $X$, if any.
- A program **flag** is a variable that indicates whether or not a specified action has taken place. Typically, a flag value of 1 indicates that the action has occurred; a value of 0 indicates that it has not occurred.

Example 6.9 demonstrates the general pseudocode for the binary search.

**Example 6.9  General Pseudocode for the Binary Search**

In the following pseudocode, the variables **Low** and **High** represent the smallest and largest array **indexes** in the part of the array currently under consideration. Recall that an array begins with 0 as the first index and the highest index number is actually one less than the number of elements in the array. If the highest index in this array is $N$, then the number of elements in the array is $N + 1$. Initially, we are searching the entire array, so $\text{Low} = 0$ and $\text{High} = N$. However, after the first attempt at locating **Key**, we are either searching the first half or the last half of the array, so either $\text{Low} = 0$ and $\text{High} = \text{Int}(N/2)$ or $\text{Low} = \text{Int}(N/2)$ and $\text{High} = N$. The variable **Index** represents the middle element of the part of the array under consideration. Thus, **Index** is initially $\text{Int}(N/2)$, and in general, it is the...
average of Low and High: \( \text{Index} = \text{Int}((\text{Low} + \text{High})/2) \). Note how this works mathematically. If \( N \) is an even number, \( N/2 \) is an integer, but if \( N \) is an odd number, then \( N/2 \) is not an integer. Since the values of \( \text{Low} \), \( \text{High} \), and \( \text{Index} \) must be integer values, we use the \( \text{Int} \) function. When \( N \) is odd, then \( \text{Index} \) will not be the exact middle, of course, but will be 0.5 below the mathematical middle. However, the search is not affected by this. In this general pseudocode, \( \text{Array} \) can be an array of any data type and \( \text{Key} \) is a variable of the same data type as \( \text{Array} \).

```pascal
Declare Low As Integer
Declare High As Integer
Declare N As Integer
Declare Index As Integer
Declare Found As Integer
Set Low = 0
Set High = N
Set Index = Int(N/2)
Set Found = 0
While (Found == 0) AND (Low <= High)
    If Key == Array[Index] Then
        Set Found = 1
    End If
    If Key > Array[Index] Then
        Set Low = Index + 1
        Set Index = Int((High + Low)/2)
    End If
    If Key < Array[Index] Then
        Set High = Index - 1
        Set Index = Int((High + Low)/2)
    End If
End While
```

**What Happened?**

In this pseudocode, after initializing the values of \( \text{Low} \), \( \text{High} \), \( \text{Index} \), and \( \text{Found} \), we enter the While loop. This loop will be reentered if the search key has still not been located, in which case \( \text{Found} = 0 \), and if there are still array elements to check, in which case \( \text{Low} <= \text{High} \).

Within the loop, we deal with the following three possible cases:

1. If \( \text{Key} == \text{Array}[\text{Index}] \), then we’ve located the search key, and \( \text{Found} \) is set equal to 1.
2. If \( \text{Key} > \text{Array}[\text{Index}] \), then we’ve got to check the upper half of the remaining array elements, and \( \text{Low} \) is adjusted accordingly.
3. If \( \text{Key} < \text{Array}[\text{Index}] \), then we’ve got to check the lower half of the remaining array elements, and \( \text{High} \) is adjusted accordingly.

In the first case, the loop is exited. In the last two cases, \( \text{Index} \) is reset to the middle of the remaining part of the array and the loop is reentered unless we’ve exhausted the array elements to be searched. Example 6.10 demonstrates a binary search.
Example 6.10  Finding the Right House

Table 6.1 shows how a binary search for the word House proceeds in the given list of eleven words.

Table 6.1  A Binary search for the word House

<table>
<thead>
<tr>
<th>Subscript</th>
<th>Word</th>
<th>First Pass</th>
<th>Second Pass</th>
<th>Third Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Aardvark</td>
<td>Low</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Book</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Dog</td>
<td>Index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>House</td>
<td>Low, Index</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Job</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Month</td>
<td>Index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Start</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Top</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Work</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Zebra</td>
<td>High</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The words are stored in an array of strings named Words[]. The pseudocode for this example is as follows, assuming Words[11] has been loaded, sorted alphabetically, and integer variables named Low, High, Index, N, and Found have been declared. A string variable, Key, has also been declared.

```
1  Set N = 10
2  Set Key = "House"
3  Set Low = 0
4  Set High = N
5  Set Found = 0
6  Set Index = Int(N/2)
7  While (Found == 0) AND (Low <= High)
8    If Key < Words[Index] Then
9      Set High = Index - 1
10     Set Index = Int((High + Low)/2)
11  Else
12    If Key > Words[Index] Then
13      Set Low = Index + 1
14     Set Index = Int((High + Low)/2)
15    Else
16      Set Found = 1 // Key must = Word[Index]
17  End If
18  End If
19 End While
```

What Happened?

On the first pass through the While loop:
- $N = 10$ so Index is $\text{Int}(N/2)$ which is 5
"House" < Words[5] since the element of Words[] that has the index 5 is Month.

High is set equal to 4 because High = Index - 1.

Index becomes Int((0 + 4)/2) = 2

Now the range of items to be searched has been reduced to Words[0] through Words[4] and the middle of this range is Words[2]. On the second pass through the loop:

- "House" is greater than Words[Index] because Words[2] = "Dog"
- Low now becomes Index + 1 = 3
- Index is now Int((4 + 3)/2) = 3

The range of items to be searched is now between Words[3] (Low) and Words[4] (High). The middle of this range is 3.5 but Int(3.5) = 3. On the third pass the search ends:

- "House" which is the 4th element of the array has the subscript 3. So Words[3] = "House" and a match is found.

Example 6.11 Combining Parallel Arrays and a Binary Search

Professor Crabtree saves all her student records in parallel arrays. The array Names[] holds the students' names, listed alphabetically by last name. Each time she gives an exam or grades a homework assignment, she adds a new parallel array: Exam1[], Exam2[], HW1[], HW2[], and so on. Now Dr. Crabtree needs to locate the record for one student, Julio Vargas. She writes a program which allows her to search for and retrieve Julio's whole record very quickly.

The pseudocode that corresponds to this program, shown here, assumes the following parallel arrays have been declared, filled with data, and that the following variables have been declared:

- Names[100] is an array of Strings with each element holding a student's last name
- First[100] is an array of Strings with each element holding a student's first name
- Exam1[100], HW1[100], and HW2[100] are parallel arrays of Floats.
- Low, High, Found, N, and Index are Integer variables.
- Student is a String variable.

Set N = 99
Set Low = 0
Set High = N
Set Found = 0
Set Index = Floor(N/2)
Write "Enter a student's name: 
Input Student
While (Found == 0) AND (Low <= High)
If Student < Names[Index] Then
Set High = Index - 1
Set \( \text{Index} = \text{Floor}((\text{High} + \text{Low})/2) \)
Else
  If \( \text{Student} > \text{Names}[\text{Index}] \) Then
    Set \( \text{Low} = \text{Index} + 1 \)
    Set \( \text{Index} = \text{Floor}((\text{High} + \text{Low})/2) \)
  Else
    Set \( \text{Found} = 1 \)
  End If
End If
End While
If \( \text{Found} == 0 \) Then
  Write "Student record not found."
Else
  Write "Student record for: "
  Write \( \text{First}[\text{Index}] + " " \) + \( \text{Names}[\text{Index}] \)
  Write "Exam 1: " + \( \text{Exam1}[\text{Index}] \)
  Write "Homework 1: " + \( \text{HW1}[\text{Index}] \)
  Write "Homework 2: " + \( \text{HW2}[\text{Index}] \)
End If

What Happened?

This program searches through the \( \text{Names}[\] \) array for the search key which is the value of the variable \( \text{Student} \). This program differs from the previous example because, in this case, the user can enter the item to be found so the program can be re-used to search for any student's record.

Once the search record has located the desired student's last name, we know the index number of that element in the \( \text{Names}[\] \) array. However, since all the arrays are parallel, we now also know the index number of that student's information in all the arrays. In fact, with very little modification, this program could be used to display all homework grades, all exam scores, or any combination of items. This is the value of using parallel arrays.

We also note that this program substitutes the \( \text{Floor}() \) function where the \( \text{Int}() \) function had been used in Example 6.10. For the purpose of this program, the two functions are interchangeable.

You can add more functionality to this program, such as finding a student's exam or homework average, computing a final course grade, and so on. The Self Check questions for this section will ask you to do some of these things.

Selection Sort

The selection sort procedure is a more efficient way to sort data stored in an array than the bubble sort technique presented in Section 6.2. The basic idea behind the selection sort is fairly simple. Here is how we would use it to sort an array in ascending order—from smallest to largest. We make several passes through the array as follows:

- On the first pass, we locate the smallest array element and swap it with the first array element
On the second pass, we locate the second smallest element and swap it with the second element of the array.

On the third pass through the array, we locate the next smallest element and swap it with the third element of the array.

And so forth.... If the array contains \( N \) elements, it will be completely sorted after at most \( N-1 \) passes.

To illustrate the selection sort, first we will do a simple example by hand. Figure 6.6 demonstrates the process for a data set consisting of the numbers 9, 13, 5, 8, 6. It displays the given data in the first (leftmost) column and the results of the four passes through this data in the next four columns, using arrows to indicate which data values have been swapped. Example 6.12 presents the general pseudocode for the selection sort.

**Example 6.12 General Pseudocode for the Selection Sort**

Now we will construct the pseudocode to sort an array named \( \text{Array} \) in ascending order. The array, \( \text{Array} \), can be an array of \( \text{Integers} \), \( \text{Floats} \), \( \text{Strings} \), or \( \text{Characters} \) and \( \text{Littlest} \) must be of the same data type. For simplicity, we will declare \( \text{Array} \) as an array of \( \text{Floats} \) in this example. The array contains \( N + 1 \) elements so the highest index value is \( N \). Therefore, we need to make \( N - 1 \) passes through the loop to complete the selection sort.

The general skeleton of this program segment is as follows:

\[
\text{For (} K = 0; \ K < N; \ K++ \text{)}
\]

\[
\text{Find the smallest, } \text{Littlest}, \text{ of } \text{Array}[K], \text{ Array}[K+1], \ldots, \text{ Array}[N]
\]

\[
\text{If } \text{Littlest} \neq \text{Array}[K] \text{ Then}
\]

\[
\text{Swap array elements } \text{Littlest} \text{ and } \text{Array}[K]
\]

\[
\text{End If}
\]

\[
\text{End For}
\]

Of course, each of the two statements in the \textbf{For} loop requires further refinement, described as follows:

- To find the smallest element, \( \text{Littlest} \), in a set of numbers, we set \( \text{Littlest} \) equal to the first number and then compare \( \text{Littlest} \), succes-


sively, to the remaining numbers. Whenever we find a number smaller than Littlest, we set that number equal to Littlest and record the current subscript, Index. Thus, this step in the pseudocode above becomes:

\[
\begin{align*}
&\text{Set } \text{Littlest} = \text{Array}[K] \\
&\text{Set } \text{Index} = K \\
&\text{For } (J = K + 1; J \leq N; J++) \\
&\quad \text{If } \text{Array}[J] < \text{Littlest} \text{ Then} \\
&\quad\quad \text{Set } \text{Littlest} = \text{Array}[J] \\
&\quad\quad \text{Set } \text{Index} = J \\
&\quad \text{End If} \\
&\text{End For}
\end{align*}
\]

To swap array elements with subscripts \(K\) and \(\text{Index}\), we use the swap routine technique described in Section 6.2 as follows:

\[
\begin{align*}
&\text{Set } \text{Temp} = \text{Array}[K] \\
&\text{Set } \text{Array}[K] = \text{Array}[\text{Index}] \\
&\text{Set } \text{Array}[\text{Index}] = \text{Temp}
\end{align*}
\]

Thus, the refined pseudocode for sorting the array \(\text{Array}\), consisting of \(N+1\) elements, in ascending order using the selection sort procedure is as follows:

\[
\begin{align*}
&\text{Declare } \text{Array}[K] \text{ As Float} \\
&\text{Declare } \text{Littlest} \text{ As Float} \\
&\text{Declare } K, N, \text{Index}, \text{Temp} \text{ As Integer} \\
&\text{For } (K = 0; K < N; K++) \\
&\quad \text{Set } \text{Littlest} = \text{Array}[K] \\
&\quad \text{Set } \text{Index} = K \\
&\quad \text{For } (J = K + 1; J \leq N; J++) \\
&\quad\quad \text{If } \text{Array}[J] < \text{Littlest} \text{ Then} \\
&\quad\quad\quad \text{Set } \text{Littlest} = \text{Array}[J] \\
&\quad\quad\quad \text{Set } \text{Index} = J \\
&\quad\quad \text{End If} \\
&\quad \text{End For}(J) \\
&\quad \text{If } K != \text{Index} \text{ Then} \\
&\quad\quad \text{Set } \text{Temp} = \text{Array}[K] \\
&\quad\quad \text{Set } \text{Array}[K] = \text{Array}[\text{Index}] \\
&\quad\quad \text{Set } \text{Array}[\text{Index}] = \text{Temp} \\
&\quad \text{End If} \\
&\text{End For}(K)
\end{align*}
\]

**Example 6.13  Sorting Large Arrays**

There are many uses for the selection sort. In this example we assume that an array of student ages is part of each student’s record in Professor Crabtree’s classes. The professor wants to see the range of ages of her students and, at a future date, plans to use the sorted data to compile other statistics. She asks you to write a program to sort the array named \(\text{Ages[]}\) in ascending order. To save space we will assume:

- \(\text{Ages}[200]\) is an array of Integers with each element holding the age of a student (in years).
- \(\text{Youngest}, J, K, M, N, \text{Temp} \text{ and, } \text{Index} \) are Integer variables.
The pseudocode for this program is as follows:

Set \( N = 199 \)
Set \( M = 0 \)
Set \( Temp = 0 \)
Set \( K = 0 \)
While \( K < N \)
  Set \( Youngest = Ages[K] \)
  Set \( Index = K \)
  Set \( J = K + 1 \)
  While \( J <= N \)
    If \( Ages[J] < Youngest \) Then
      Set \( Youngest = Ages[J] \)
      Set \( Index = J \)
    End If
    Set \( J = J + 1 \)
  End While(J)
  If \( K \neq Index \) Then
    Set \( Temp = Ages[K] \)
    Set \( Ages[K] = Ages[Index] \)
    Set \( Ages[Index] = Temp \)
  End If
  Set \( K = K + 1 \)
End While(K)
Write "Ages sorted: 
While \( M < N + 1 \)
  Write \( Ages[M] \)
  Set \( M = M + 1 \)
End While(M)

What Happened?

The outer \texttt{While} loop of this program segment performs 199 iterations. Why do we only need 199 iterations to sort 200 items? On the first pass through the outer loop the initial value of the variable \texttt{Youngest} is set equal to the value of the first element of the array, \texttt{Ages[0]}. This is then compared to the second element of the array, \texttt{Ages[0 + 1]}. The subscript \( J \) is one larger than \( K \). On the second pass the value of \texttt{Youngest} is compared to the third element of \texttt{Ages[ ]}. Continuing on with this reasoning, on the 199\textsuperscript{th} pass, the value of \texttt{Youngest} is compared to the 200\textsuperscript{th} element of \texttt{Ages[ ]}. This is why, for \( N \) elements in an unsorted array, we need \( N - 1 \) iterations of the outer loop to sort the elements.

Once inside the outer loop, the inner loop begins. The first \texttt{If-Then} statement checks to see if the value of the element in question is smaller than \texttt{Youngest}. If it is, \texttt{Youngest} is now set to that value. Also, \texttt{Index} is now set to the present value of \( J \) (the subscript of the element being checked). If this element is not smaller than \texttt{Youngest}, nothing changes. Then \( J \) is incrementated and the loop continues, checking all elements of the array against \texttt{Youngest} and continually replacing the value
of Youngest as a smaller and smaller value is discovered. After this loop has completed all its iterations one time, Youngest now holds the smallest value in the whole array.

Whenever the value of Youngest is replaced, Index takes on the value of J which identifies the element that holds that new small value. At the end of this inner loop, the value of Index identifies the element of the array that has the smallest value in the whole array. A check is now made, in the second If-Then statement, to see if the element that the outer loop started with (K) is the same as Index. If it is, then Ages[K] already holds the smallest value in the array. If it is not, we need to swap the value of Ages[K] with Ages[Index] to put the smallest value in this spot. The swap routine does this. If this is unclear consider this scenario:

Assume that the values of a 3-element array, Ages[] are:


On the first pass, K = 0 so Youngest now equals the value of Ages[0] which is 5. At the end of the first pass through inner While loop, Youngest has been compared to Ages[1] and no switch is made since Ages[1] has a value greater than Youngest. At this point J = 1. Then it is incremented to 2. When Ages[2] is checked, since its value (3) is less than Youngest, the value of Youngest is replaced by 3. Now Index is set equal to J so Index now equals 2. The inner loop ends with Youngest = 3 and Index = 2. (J, by the way, equals 3, but this is irrelevant since its value will be set back appropriately before entering the inner loop again).

Now the program proceeds to the second If-Then statement. It checks to see if K (which is still 0) is the same as Index (which is 2). Since this results in a Boolean True (i.e., it is true that K != Index), the swap occurs and the new value of Ages[0] is the value of Ages[Index] which is Ages[2] which is our smallest value. Now the smallest value is in the first element of the array.

The fact that now Ages[2] has the value of 5 and 5 is smaller than the value of Ages[1] (8) is irrelevant. All this will be sorted out (literally and figuratively!) in subsequent iterations.

After all this has happened, the outer loop begins again, now starting with the second value in the array. At the end of the second pass, the second-smallest value is stored in the second element of the array. At the end of 2 passes through the outer loop the array has been sorted from smallest to largest. In this pseudocode, we include a small loop at the end to display the sorted array.

But perhaps the most interesting thing about this example is that a computer can complete this sort for an array containing many thousands of elements in a lot less time than it takes a person to read through this explanation.

Self Check for Section 6.3

6.14 Indicate whether each of the following statements is true or false.

a. T F The binary search requires that the list of table keys be ordered.

b. T F The binary search cannot be used to locate a numeric search key.
6.15 Rewrite the binary search pseudocode given in this section to perform a search for an element Key in an array Array with N elements that has been sorted in descending order.

Self Check Question 6.16 refers to Example 6.11

6.16 Assume Professor Crabtree gave three exams and a final exam during one semester. The scores have been loaded into parallel arrays of floats named Exam1[], Exam2[], Exam3[], and Final[]. Arrays for students’ names are also loaded, as described in Example 6.11. Write pseudocode to do the following:
- Find the record for a student named Mary Reilly. Calculate Mary’s exam average for the course using the criteria that Professor Crabtree counts the final exam twice in her exam average:
  \[
  \text{ExamAvg} = \frac{(\text{Exam1} + \text{Exam2} + \text{Exam3} + 2 \times \text{Final})}{5}
  \]
- Output Mary Reilly’s full name and her exam average.

6.17 Indicate whether each of the following statements is true or false.
- a. T F The selection sort method requires that the given array is already ordered.
- b. T F The selection sort method requires that the programming language contains a swap statement.

Self Check Question 6.18 refers to Example 6.13

6.18 Add pseudocode to Example 6.15 to produce the output given below.
Assume that the ages in Ages[200] run from 16 years to 70 years.
Number of students younger than 17: XXX
Number of students between 17 - 22: XXX
Number of students between 23 - 30: XXX
Number of students between 31 - 45: XXX
Number of students older than 45: XXX

6.4 Strings as Arrays of Characters

In the first three sections of this chapter, we described some examples of one-dimensional arrays and algorithms used with them. In this section, we will discuss how arrays are related to the topic of character strings.

In Chapter 1, we introduced character strings (or more simply, strings) as one of the basic data types. Some programming languages do not contain a string data type. In those languages, strings are implemented as arrays whose elements are characters. Even in programming languages that contain this data type, strings can be formed as arrays of characters. In this section, we will consider strings from this point of view.

When defining a string as an array of characters, we will always indicate the data type in the Declare statement. This practice is required when writing actual code. For example, the following statement:

```
Declare FirstName[15] As Character
Declare LastName[20] As Character
```
defines the variables FirstName and LastName to be strings of at most, 15 and 20 characters, respectively.

**Concatenation Revisited**

Whether we consider strings as a built-in data type or as arrays of characters, we can perform certain basic operations on them, as shown in Example 6.14.

**Example 6.14** **Stringing Arrays Together**

This program segment inputs two strings from the user, concatenates them, and displays the result. You will recall that concatenation means to join two items and the \( + \) symbol is used to perform the concatenation operation.

```
Declare String1[25] As Character
Declare String2[25] As Character
Declare String3[50] As Character
Write "Enter two character strings."
Input String1
Input String2
Set String3 = String1 + String2
Write String3
```

In this pseudocode, notice that String1, String2, and String3 are defined as arrays of characters, but when they are used in the program, the array brackets do not appear. For example, we write

```
Input String1
Input String2
```

rather than

```
Input String1[25]
Input String2[25]
```

This usage is typical of actual programming languages and also conforms to our previous way of referencing strings when they were considered as a built-in data type.

After the two strings have been input, the statement

```
Set String3 = String1 + String2
```

concatenates them and the Write statement displays the result. If code corresponding to this pseudocode is run and the user enters the strings "Part" and "Time" for String1 and String2, the program's output would be as follows:

```
PartTime
```

**Concatenation versus Addition**

The use of the concatenation operator in the following line of Example 6.14

```
Set String3 = String1 + String2
```

merits a short comment at this point.
If, for example, a person had three integer variables as follows:

\[
\text{Var1} = 10, \quad \text{Var2} = 15, \quad \text{Var3} = 0
\]

the statement:

\[
\text{Set Var3} = \text{Var1} + \text{Var2}
\]

would result in \( \text{Var3} = 25 \).

However, if \text{Var1}, \text{Var2}, and \text{Var3} were string variables with values as follows:

\[
\text{Var1} = "10", \quad \text{Var2} = "15", \quad \text{Var3} = "0"
\]

the statement

\[
\text{Set Var3} = \text{Var1} + \text{Var2}
\]

would result in \( \text{Var3} = "1015" \).

In our pseudocode, as well as in programming languages that use the same symbol for concatenation as for addition, it is the data type declaration that tells the computer which operation (addition or concatenation) to perform.

### String Length versus Array Size

The length of a string is the number of characters it contains. For example, the array \text{String3} of Example 6.14 is declared as an array of 50 elements, but when "PartTime" is assigned to \text{String3}, only the first eight array elements are used. Thus, the length of the string "PartTime" is 8.

In some algorithms (see Example 6.15), it is useful to know the length of a string that has been assigned to a given array of characters. For this purpose, programming languages contain a \text{Length} function, which we will write as follows:

\[
\text{Length_Of(String)}
\]

We have discussed this function earlier in the text and will now review its characteristics and applications in the context of character arrays.

The \text{Length_Of} function may be used in a program wherever a numeric constant is valid. The value of this function is the length of the given string or string variable and may be used in a program wherever a numeric constant is valid. For example, when code corresponding to the following pseudocode is run:

\[
\begin{align*}
\text{Declare Str[10] As Character} \\
\text{Set Str} &= "HELLO" \\
\text{Write Length_Of(Str)}
\end{align*}
\]

the output will be the number 5 because the string "HELLO" is made up of five characters.

Recall that when an array is declared, the number specified in the declaration statement determines the number of storage locations in the computer’s memory allocated to that array. If the array represents a string (an array of characters), then each storage location consists of one byte of memory (see Chapter 0). When a string is assigned to this array, the beginning elements of the array are filled with the characters that make up the string, a special symbol is placed in the next stor-
age location, and the rest of the array elements remain unassigned. For example, a string named Str declared as an array of 10 characters and assigned the value "HELLO" can be pictured to look in memory as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>&quot;H&quot;</td>
<td>&quot;E&quot;</td>
<td>&quot;L&quot;</td>
<td>&quot;L&quot;</td>
<td>&quot;O&quot;</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, the symbol # represents the character that is automatically placed at the end of the assigned string. Thus, to determine the length of the string contained in Str, the computer simply counts the storage locations (bytes) associated with the variable Str until the terminator symbol, #, is reached.

Example 6.15 illustrates how strings can be manipulated by manipulating the arrays in which they are contained.

**Example 6.15 Using the Length_Of Function with Character Arrays**

This program segment inputs a person's full name with first name first, then a space, and then the last name. It stores the initials of that person as characters, and displays the name in the following form:

```
LastName, FirstName
```

This pseudocode uses three strings. One stores the input name as FullName, and the other two store the first and last names as FirstName and LastName. It also makes use of two character variables to store the initials, FirstInitial and LastInitial. The trick to identifying which part of the input string is the first name and which part is the last name is to locate the blank space between them.

We will assume that the following arrays and variables have been declared:

- **Character arrays:** `FullName[30], FirstName[15], LastName[15]`
- **Character variables:** `FirstInitial, LastInitial`
- **Integer variables:** `J, K, Count`

1. Write "Enter a name in the following form: firstname lastname:"
2. Input `FullName`
3. Set `Count = 0`
4. While `FullName[Count] != " "`
5.   Set `FirstName[Count] = FullName[Count]`
6.   Set `Count = Count + 1`
7. End While
8. Set `FirstInitial = FullName[0]`
9. Set `LastInitial = FullName[Count+1]`
10. Set `J = 0`
11. For `(K = Count + 1; K <= Length_Of(FullName) - 1; K++)`
12.   Set `LastName[J] = FullName[K]`
13.   Set `J = J + 1`
14. End For
15. Write `LastName " , " + FirstName`
16. Write "Your initials are " + `FirstInitial + LastInitial`
What Happened?

After the person's full name is input, the counter-controlled while loop on lines 4–7 assigns the characters in FullName to FirstName until the blank between the first and last names is encountered. At this point, the value of Count is the length of FirstName, and because the blank corresponds to the index Count, the first character in LastName corresponds to the index Count + 1. Thus, the two assignment statements that follow the while loop on lines 8 and 9 correctly store the person’s initials. The new variable, J, (line 10) ensures that the letters of the last name are stored in the correct locations in the LastName array. The for loop on lines 11-14 copies the correct part of FullName to LastName. Finally, the write statement on line 15 displays the person’s name, last name first with a comma between the two parts of the name. We also display the person's initials, which have been stored in FirstInitial and LastInitial.

Self Check for Section 6.4

6.19 Indicate whether each of the following statements is true or false.
   a. T  F If a string variable Str has been declared to be an array of 25 characters, then the length of Str must be 25.
   b. T  F To input a string from the user, we must know how many characters are in that string.

6.20 Suppose that a string variable, Name, has been declared as an array of characters and has been assigned a value. Write a program segment that displays the first and last characters of Name. (Hint: the Length function comes in handy here.)

6.21 Write a program segment that declares string variables String1 and String2 to be arrays of 25 characters, inputs a value for String1 from the user, and copies this value into String2.

6.5 Two-Dimensional Arrays

In the arrays you have seen so far, the value of an element has depended upon a single factor. For example, if one element in an array holds a student’s ID number, the value of this number depends on which student is being processed. Sometimes it's convenient to use arrays whose elements are determined by two factors. For example, we might have several test scores associated with each student, in which case the value we look for would depend on two factors—the particular student and the particular test in which we are interested. Another example might be the records of the monthly sales for salespeople for a year. Each salesperson has 12 numbers (one for each month’s sales) associated with him or her, so the value we look for would depend on which salesperson and which month is of interest. In these cases, we use two-dimensional arrays.

An Introduction to Two-Dimensional Arrays

A two-dimensional array is a collection of elements of the same type stored in consecutive memory locations, all of which are referenced by the same variable.
name using two subscripts. For example, $\text{MyArray}[2,3]$ is one element of a two-dimensional array named $\text{MyArray}$. Example 6.16 illustrates one use of two-dimensional arrays.

**Example 6.16 Introducing Two-Dimensional Arrays**

Suppose we want to input the scores of 30 students on five tests into a program. We can set up a single two-dimensional array named $\text{Scores}$ to hold all these test results. The first subscript of $\text{Scores}$ references a particular student; the second subscript references a particular test. For example, the array element $\text{Scores}[0,0]$ contains the score of the first student on the first test and the array element $\text{Scores}[8,1]$ contains the score of the ninth student on the second test.

This situation may be easier to understand if you picture the array elements in a rectangular pattern of horizontal rows and vertical columns. The first row gives the scores of the first student, the second row gives the scores of the second student, and so forth. Similarly, the first column gives the scores of all students on the first test, the second column gives all scores on the second test, and so forth (see Figure 6.7). The entry in the box at the intersection of a given row and column represents the value of the corresponding array element. The following is shown in Figure 6.7:

- $\text{Scores}[1,3]$, the score of Student 2, Boynton, on Test 4, is 73
- $\text{Scores}[29,1]$, the score of Student 30, Ziegler, on Test 2, is 76

**Figure 6.7** The two-dimensional array named $\text{Scores}$

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student 1: Arroyo</td>
<td>92</td>
<td>94</td>
<td>87</td>
<td>83</td>
<td>90</td>
</tr>
<tr>
<td>Student 2: Boynton</td>
<td>78</td>
<td>86</td>
<td>64</td>
<td>73</td>
<td>84</td>
</tr>
<tr>
<td>Student 3: Chang</td>
<td>72</td>
<td>68</td>
<td>77</td>
<td>91</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Student 30: Ziegler</td>
<td>88</td>
<td>76</td>
<td>93</td>
<td>69</td>
<td>52</td>
</tr>
</tbody>
</table>

**Declaring Two-Dimensional Arrays**

Like their one-dimensional counterparts, two-dimensional arrays must be declared before they are used. We will use a declaration statement for two-dimensional arrays that is similar to the one we’ve been using for one-dimensional arrays. For example, we will declare the array of Example 6.16 using the following statement:

```
Declare Scores[30,5] As Integer
```

The numbers 30 and 5 inside the brackets indicate the number of elements in this array. This statement allocates 150 ($30 \times 5$) consecutive storage locations in the computer’s internal memory to hold the 150 elements of the array $\text{Scores}$. Example 6.17 illustrates some basic points about using two-dimensional arrays.
Example 6.17  The Basics of Two-Dimensional Arrays

Consider the following pseudocode:

3. Declare FirstPlace As Integer
4. Set FirstPlace = 5
5. Set ArrayA[FirstPlace,10] = 6
7. Write ArrayA[5,2*FirstPlace]
8. Write ArrayB[7]

What Happened?

Here, the Declare statements on lines 1 and 2 declare two arrays—the first is two-dimensional with 10 rows and 20 columns (200 elements) and the second is one-dimensional with 20 elements. Because FirstPlace is 5, the following is true:

- The assignment statement on line 5 sets ArrayA[5,10] equal to 6. In other words, the value of the 6th row, 11th column of ArrayA is equal to 6.
- The assignment statement on line 6 sets ArrayB[7] equal to the value of ArrayA[5,10], which is a 6. So now the value of the 8th element in ArrayB = 6.
- The Write statements on lines 7 and 8 display the value of the element in the 6th row, 11th column of ArrayA and the value of the 8th element of ArrayB so the number 6 will be displayed twice.

Using Two-Dimensional Arrays

As you have seen, counter-controlled loops, especially the For variety, provide a valuable tool for manipulating one-dimensional arrays. In the two-dimensional case, as shown in Example 6.18, nested For loops (see Chapter 5) are especially useful.

Example 6.18  Nested Loops for Loading a Two-Dimensional Array

This program segment inputs data into a two-dimensional array, Scores, whose elements are test scores. The first subscript of Scores refers to the student being processed; the second subscript refers to the test being processed. For each of the 30 students, the user is to input five test scores. The pseudocode is as follows:

Declare Scores[30,5] As Integer
Declare Student As Integer
Declare Test As Integer
For (Student = 0; Student < 30; Student++)
    Write "Enter 5 test scores for student " + (Student + 1)
    For (Test = 0; Test < 5; Test++)
        Input Scores[Student,Test]
    End For(Test)
End For(Student)
What Happened?
This pseudocode results in one input prompt for each student. The prompt
instructs the user to enter the five scores for that student. For example, if code
_corresponding to this pseudocode is run, the following text will be displayed:

Enter 5 test scores for student 1

and then execution will pause for input. After the user enters the five scores, he or
she will see the following:

Enter 5 test scores for student 2

and so forth.

Example 6.19 demonstrates the use of nested loops to display the contents of the
array.

Example 6.19 Using Nested Loops to Display the Contents
of a Two-Dimensional Array
Suppose that a two-dimensional array Scores has been declared and assigned data
as described in Example 6.18. Index is an integer variable that is used to identify
the element of Scores for a particular student’s record. Notice that, if the user
enters a value of 4 for Index, this actually corresponds to the the elements in the
3rd row of Scores. The following pseudocode displays the test scores of a student
specified by the user:

Write "Enter the number of a student, and"
Write "his or her test scores will be displayed."
Input Index
For (Test = 0; Test < 5; Test++)
    Write Scores[Index - 1,Test]
End For

The pseudocode shown in Examples 6.18 and 6.19 is somewhat user unfriendly
because it requires the user to refer to students by number (student 1, student 2,
and so forth) rather than by name. Example 6.20 presents a more comprehensive
example that corrects this defect by using a second (parallel) one-
dimensional array of names.

Example 6.20 The Friendly Version Puts It All Together
This program inputs the names and test scores for a class of students and then dis-
plays each name and that student's average test score. It uses a one-dimensional
array, Names, whose elements are strings. This array holds the student names; a
two-dimensional array, Scores, of numbers to hold the test scores.

We assume that there is a maximum of 30 students in the class, but that the exact
number is unknown prior to running the program. Thus, we need a sentinel-
controlled while loop to input the data. However, during the input process, we dis-
cover how many students are in the class, and we can use that number to process
and display the array elements. For the sake of clarity, we declare all the variables used in this program.

```
1 Declare Names[30] As String
2 Declare Scores[30,5] As Integer
3 Declare Count As Integer
4 Declare Test As Integer
5 Declare K As Integer
6 Declare J As Integer
7 Declare StudentName As String
8 Declare Sum As Float
9 Declare Average As Float
10 Set Count = 0
11 Write "Enter a student's name; enter * when done."
12 Input StudentName
13 While StudentName != "*
14    Set Names[Count] = StudentName
15    Write "Enter 5 test scores for " + Names[Count]
16    Set Test = 0
17    While Test < 5
18       Input Scores[Count,Test]
19       Set Test = Test + 1
20   End While(Test)
21    Set Count = Count + 1
22    Write "Enter a student's name; enter * when done."
23   Input StudentName
24 End While(StudentName)
25 Set K = 0
26 While K <= Count - 1
27    Set Sum = 0
28    Set J = 0
29    While J < 5
30       Set Sum = Sum + Scores[K,J]
31       Set J = J + 1
32   End While(J)
33    Set Average = Sum/5
34    Write Names[K] + "": " + Average
35    Set K + K + 1
36 End While(K)
```

What Happened?

By now you can easily recognize what many parts of the pseudocode do—therefore, rather than go through this program line by line, we will concentrate on the more advanced logic.

- Lines 1–12 declare all the variables, set the initial counter to 0, and get the “seed” value for the next loop (the first student's name). To avoid inputting the sentinel value, *, into the array Names[], we temporarily assign each input string to the variable StudentName. If the value of StudentName is not
“*”, the While loop is entered and that string is assigned to the next element of Names[]. As soon as * is entered, the While loop is exited so it is never input into the Names[] array.

- The While loop on lines 13-24 is the most complicated part. It inputs the student names into the one-dimensional array, Names[], and the five test scores into the two-dimensional array, Scores[]. How does it do this?
- The statement on line 14, Set Names[Count] = StudentName, loads the first student into the first element of the array, Names[0]. On the next time around this loop, the next name entered will be loaded into the second element of Names[1], and so forth.
- Line 15 asks the user to enter the five test scores for whichever student was entered on the previous line. The inner loop, on lines 17–20, loads each of the test scores into the two-dimensional array. The first dimension of this array, Scores[30, 5] is the number (Count) that identifies the student and the second dimension is a score. If Martin has been entered as the third student and his test scores are 98, 76, 54, 92, and 89, then the following values are stored in Scores:
  - Scores[2,0] = 98
  - Scores[2,1] = 76
  - Scores[2,2] = 54
  - Scores[2,3] = 92
  - Scores[2,4] = 89
- Line 21 increments the value of Count to prepare for the next student.
- After the five test scores for one student have been loaded into the two-dimensional array, lines 22 and 23 prompt for and input another student’s name. If there are no more students, Count has the value of the number of entries in the array, Names[], and each student is identified in Scores[] by the number corresponding to his/her array subscript in Names[]. Note that Count has the value of the number of entries, which is one more than the highest subscript.
- When all the students and their test scores have been loaded into the two arrays, the program control goes to the outer While loop on lines 26–36. In effect, the outer loop says “Using the student identified by number K, add up his/her test scores and divide that sum by 5 to get the average.”
- The inner loop on lines 29–32 gets each test score and does the sum.
- Line 33 computes the average and line 34 displays the name of the student and his/her average.
- The outer loop makes another pass, as long as K (the variable that identifies the student) is less than or equal to Count - 1.

Higher-Dimensional Arrays

Although they are not used very often, arrays with three or even more subscripts are allowed in some programming languages. These higher-dimensional arrays can be used to store data that depends upon more than two factors (subscripts).
Self Check for Section 6.5

6.22 How many storage locations are allocated by each statement?
   - Declare A[4,9] As Integer
   - Declare Left[10], Right[10,10] As Float

6.23 A two-dimensional array named Fog has two rows and four columns:
   5 10 15 20
   25 30 35 40
   a. What are the values of Fog[0,1] and Fog[1,2]?
   b. Which elements of Fog contain the numbers 15 and 25?

6.24 What is displayed when code corresponding to the following pseudocode is executed?
   Declare A[2,3] As Integer
   Declare K As Integer
   Declare J As Integer
   Set K = 0
   While K <= 1
      Set J = 0
      While J <= 2
         Set J = J + 1
      End While(J)
   End While(K)
   Write A[0,1] + " " + A[1,0] + " " + A[1,2]

6.25 How many times are the prompts on lines 3 and 4 of the following pseudocode displayed?
   1  For (I = 0; I < 5; I++)
   2    For (J = 0; J < 12; J++)
   3      Write "Enter rainfall in state " + I
   4      Write " in month " + J
   5      Input Rain[I, J]
   6    End For(J)
   7  End For(I)

6.26 Write a program segment that determines the largest element, Max, of a two-dimensional array named PositiveIntegers of positive integers that has already been declared and has had data already input. The input is three rows and five columns.

6.6 Focus on Problem Solving:
   A Grade Management Program

In this section, we will use a common programming problem to practice using the concepts learned so far, especially most of those presented in this chapter. We will create a program for a college professor to help view the grades for each student
in a class and to allow the professor to evaluate a class’s results by displaying some simple statistics. The program will make use of parallel arrays and search and sort routines.

Problem Statement

Professor Hirsch has asked you to write a program to help manage his grades for his Technical Writing class. He wants the program to do several things. After entering the final numerical grade for each student, he wants the number to be translated into a letter grade. He also wants to be able to enter any student’s name and see the data relating to that student (the student’s ID number, numerical final grade, and letter grade). Finally, he wants to see a report that includes all the students and their information and a statistical summary that includes the class average, the highest and lowest scores in the class, and the number of people who scored above, below, and exactly at the class average.

Problem Analysis

We begin our analysis of the problem by examining the required output. In this problem, there are several types of required output.

Professor Hirsch wants to see all the students with their information (names, ID’s, numerical scores, and letter grades) as well as a statistical summary at the bottom of the report. The statistical summary will look as follows:

Summary
The average score for this class is: Class Average
High score for the class: High Score
Low score for the class: Low Score
Number of scores above the average: Number Above
Number of scores below the average: Number Below
Number of scores equal to the average: Number At Mean

A typical report for a sample class is shown in Figure 6.8.

Professor Hirsch also wants to be able to pull up the record for any single student. In this case, Professor Hirsch should be able to input a student’s name and see that student’s record displayed in a form similar to the following:

Student Name:
Student ID Number:
Student’s final score:
Student’s letter grade:

For this program we will store the data for each student in four parallel arrays as follows:

- **Names[50]** is a String array that will hold the name of each student in the form: lastname, firstname
- **IDNum[50]** is an Integer array that will hold an identification number for each student. This is a number usually assigned by the college or, if Professor Hirsch desires, he can create his own student ID numbers.
- **Final[50]** is a Float array that will hold each student’s numerical grade at the end of the semester.
Grade[50] is a Character array that will hold each student’s letter grade (A, B, C, D, or F).

The arrays in this program will be declared to have 50 elements but, of course, that number can be changed to hold as many elements as Professor Hirsch desires. Later, if the Professor is happy with this program, we can add modules to hold homework grades, exam grades, attendance records, and so on. These values could be used by a module that will calculate the final numeric grade and send the results to the Final array for the reports.

We need to load the students’ data into the first three arrays (the Grade array will be populated within our program). To do this, we will use a loop and we need several input variables as follows:

- **StudentName** is a String variable
- **StudentID** is an Integer variable
- **StudentScore** is a Float variable

The following output variables are also necessary:

- **ClassAvg** is a Float variable and will contain the average of all the scores
- **HighScore** is a Float variable and will contain the highest score in the class

**Figure 6.8** Sample report for Grade Management program

<table>
<thead>
<tr>
<th>Student Name</th>
<th>ID Number</th>
<th>Numerical Score</th>
<th>Letter Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Venit, Stewart</td>
<td>1231</td>
<td>98.2</td>
<td>A</td>
</tr>
<tr>
<td>Kim, John</td>
<td>1245</td>
<td>97.3</td>
<td>A</td>
</tr>
<tr>
<td>Vargas, Orlando</td>
<td>1268</td>
<td>94.6</td>
<td>A</td>
</tr>
<tr>
<td>Lee, Nancy</td>
<td>1288</td>
<td>88.7</td>
<td>B</td>
</tr>
<tr>
<td>Voglio, Nicholas</td>
<td>1271</td>
<td>86.9</td>
<td>B</td>
</tr>
<tr>
<td>Stein, Mandy</td>
<td>1213</td>
<td>84.2</td>
<td>B</td>
</tr>
<tr>
<td>Ettcity, Kate</td>
<td>1222</td>
<td>83.3</td>
<td>B</td>
</tr>
<tr>
<td>Lopez, Maria</td>
<td>1263</td>
<td>80.0</td>
<td>B</td>
</tr>
<tr>
<td>Moser, Hans</td>
<td>1244</td>
<td>78.9</td>
<td>C</td>
</tr>
<tr>
<td>Smith, Jane</td>
<td>1208</td>
<td>78.5</td>
<td>C</td>
</tr>
<tr>
<td>Goshdigian, Anne</td>
<td>1212</td>
<td>78.2</td>
<td>C</td>
</tr>
<tr>
<td>Alerov, Mark</td>
<td>1216</td>
<td>76.3</td>
<td>C</td>
</tr>
<tr>
<td>Iijima, Kazuko</td>
<td>1225</td>
<td>75.4</td>
<td>C</td>
</tr>
<tr>
<td>Fitch, James</td>
<td>1275</td>
<td>72.8</td>
<td>C</td>
</tr>
<tr>
<td>Chen, Karen</td>
<td>1236</td>
<td>71.5</td>
<td>C</td>
</tr>
<tr>
<td>Baptiste, Etienne</td>
<td>1279</td>
<td>70.1</td>
<td>C</td>
</tr>
<tr>
<td>Cooper, Martha</td>
<td>1260</td>
<td>68.2</td>
<td>D</td>
</tr>
<tr>
<td>McDonell, Chris</td>
<td>1251</td>
<td>64.5</td>
<td>D</td>
</tr>
<tr>
<td>Montas, Eric</td>
<td>1246</td>
<td>62.3</td>
<td>D</td>
</tr>
<tr>
<td>Drake, Elizabeth</td>
<td>1218</td>
<td>59.4</td>
<td>F</td>
</tr>
</tbody>
</table>

**Summary**

- The average score for this class is: 78.5
- High score for the class: 98.2
- Low score for the class: 59.4
- Number of scores above the average: 9
- Number of scores below the average: 10
- Number of scores equal to the average: 1
LowScore is a Float variable and will contain the lowest score in the class

NumAbove is an Integer variable that will contain the number of scores above the mean

NumBelow is an Integer variable that will contain the number of scores below the mean

NumAvg is an Integer variable that will contain the number of scores that are equal to the mean

Program Design

As with virtually all programs, we will begin this program with a Welcome_Message module. In this case, the Welcome_Message module will explain briefly what the program does and will offer Professor Hirsch several options.

We need a module for input so Professor Hirsch can enter his students' names, ID numbers, and final scores. This will be the Enter_Info module. Once the data has been input, it will be processed. First, the numerical grade for each student will be converted to a letter grade and stored in the parallel array, Grades. We will use a Case Statement to accomplish this in a Letter_Grade module.

The Statistics module will process the data. First, it will compute the average score. Once we have this value, we will use the bubble sort routine to sort the scores in descending order so that the report will display the scores from highest to lowest. Later, when Professor Hirsch offers to pay us more to increase the functionality of this program, modules can be added to allow the report to display in sorted order by names or by ID numbers. Remember that, when we sort one array, we must be sure to sort all the parallel arrays so all the information for any particular student can be easily accessed through its index number (subscript). Once the numeric scores in the Final array have been sorted, it is possible to identify the highest and lowest scores. The Statistics module will also count the number of scores above, below, or exactly the same as the mean (average). This module will also generate a report, as shown in the sample report of Figure 6.8.

Finally, the Display_Student module will allow Professor Hirsch to input one student's name. Using the serial search technique, that student's complete record will be located and displayed.

Thus, our program will consist of the following modules.

1. The Main module calls its submodules into action.
2. The Welcome_Message module displays a welcome message and explains how to use the program.
3. The Enter_Info module accepts all the information for each student and stores that information in three parallel arrays.
4. The Letter_Grade module converts the numerical score of each student into a letter grade which is stored in another parallel array.
5. The Statistics module processes the data. It computes the average score of all the students as well as determines the highest and lowest scores. It also counts the number of scores at, above, and below the average and generates a report to display this information along with detailed information about every student in the class.
6. The Display_Student module allows the user to search for the record of any student and have that information displayed.

Modules 2 and 3 are called from the Main module. Module 3 calls modules 4, 5, and 6. This division of programming tasks is illustrated in the hierarchy chart shown in Figure 6.9.

In our program we will declare all variables and arrays that are used throughout the program in the Main module for simplicity. However, in most real programs, variables are declared “locally” (i.e., within their specific modules) and their values are passed from one module to another as necessary. The topic of where to declare variables and how to pass their values from one submodule to another is a very important topic and will be covered in depth in Chapter 8. While, for now, we are declaring our variables and arrays in the Main module (“globally”), it should be noted that we are doing it only for simplicity at this point and there is a better way to do it. With that in mind, the pseudocode for each module is given next.

**Main Module**

```
Begin
    Declare Names[50] As String
    Declare IDNum[50] As Integer
    Declare Final[50] As Float
    Declare Grade[50] As Character
    Declare StudentName As String
    Declare StudentID, NumAbove, NumBelow, NumAvg, StudentCount As Integer
    Declare StudentScore, ClassAvg, HighScore, LowScore, Sum As Float
    Call Welcome_Message module
    Call Enter_Info module
End Program
```

**Figure 6.9** Hierarchy chart for the Grade Management program
Welcome Message Module

This module will present the program title, identify the programmer and other program data, and provide a brief explanation of what the program does. It will explain that the user must enter the data and then he or she will see a course statistical summary and be given the option to locate and view the record of a single student. This model consists solely of Write statements.

Enter_Info Module

This module will allow Professor Hirsch to input data for all the students in a class using a While loop. While the program is written to allow up to 50 students, a sentinel value is employed so the program can be used for fewer than 50 students. Recall that we have already declared all the necessary arrays and variables needed for this module so the pseudocode shown begins with entering the data. The variable StudentCount is initialized here and, at the end of this module, it holds the number of students in the class. For the purposes of this program, it is assumed that StudentCount retains its value when used in subsequent submodules. The pseudocode for this module is as follows:

Set StudentCount = 0
Write "Enter a student's full name; enter '*' when done."
Write "Use the form 'LastName, FirstName' for each entry."
Input StudentName
While StudentName != "*"
    Set Names[StudentCount] = StudentName
    Write "Enter this student's ID number: "
    Input StudentID
    Set IDNum[StudentCount] = StudentID
    Write "Enter the final score for this student: "
    Input StudentScore
    Set Final[StudentCount] = StudentScore
    Set StudentCount = StudentCount + 1
    Write "Enter another student's name; enter '*' when done."
    Input StudentName
End While
Call Letter_Grade module
Call Display_Student module
Call Statistics module

At this point, three parallel arrays have been filled. StudentCount has the value corresponding to the number of elements in each array and also is one higher than the highest subscript of each array.

Letter_Grade Module

This module will combine a For loop and a Case statement with what we have learned about parallel arrays. The numerical grade is stored in the Final array. Now we will convert that number to a letter grade according to the grade scale shown in Figure 6.10, provided by Professor Hirsch. The letter grades will be stored in a fourth parallel array, Grade. Since StudentCount has the value of the
number of students in the class, we use `StudentCount` as the limit value of the `For` loop, comparing that value to a new counter. The pseudocode for this module is as follows:

```
Declare I As Integer
For (I = 0; I < StudentCount; I++)
    Set StudentScore = Final[I]
    Select Case of StudentScore
        Case >= 90.0:
            Set Grade[I] = "A"
        Case >= 80.0:
            Set Grade[I] = "B"
        Case >= 70.0:
            Set Grade[I] = "C"
        Case >= 60.0:
            Set Grade[I] = "D"
        Default:
            Set Grade[I] = "F"
    End Case
End For
```

**Display_Student Module**

This module will use the serial search technique to search for a specific student and, because of the characteristics of parallel arrays, will display all the information for the requested student. This pseudocode is written to search for a student by his or her ID number. However, it could easily be changed to allow the search to be based on the student’s name. The pseudocode for this module is as follows:

```
Declare J As Integer
Declare IDKey As Integer
Declare Found As Integer
Set Found = 0
Set J = 0
Write "Enter the ID number for one student to"
Write "view all the information about that student: ":
```

---

**Figure 6.10 Grade scale for the Grade Management program**

<table>
<thead>
<tr>
<th>Grade Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical Score</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>90.0 – 100.0</td>
</tr>
<tr>
<td>80.0 – 89.9</td>
</tr>
<tr>
<td>70.0 – 79.9</td>
</tr>
<tr>
<td>60.0 – 69.9</td>
</tr>
<tr>
<td>under 60.0</td>
</tr>
</tbody>
</table>
Input IDKey
While (Found == 0) AND (J < StudentCount)
    If IDNum[J] == IDKey Then
        Set Found = 1
    End If
    Set J = J + 1
End While
If Found == 0 Then
    Write "Student ID was not found."
Else
    Write "Student name: " + Names[J - 1]
    Write "Student ID number: " + IDNum[J - 1]
    Write "Student's final score: " + Final[J - 1]
    Write "Student's letter grade: " + Grade[J - 1]
End If

Note that J is incremented at the end of the While loop. Therefore, when the correct student ID number is located and the loop ends, the value of J is one higher than the subscript of the desired array element. Therefore, the subscript of the array element to be displayed is J - 1.

Statistics Module

This module is the longest and most complicated of the program. The bubble sort technique is used to sort the elements of the numerical scores array, Final, in descending order. To keep all the information about each student together, the subscripts of all the parallel arrays must be changed to match the sorted elements of the Final array.

After the Final array is sorted, all the scores are summed in order to compute the average. The program also counts the number of scores at, above, and below the average, and the high score and low score are identified. This last part is very simple since, after the scores are sorted, the first element of the Final array is the highest score and the last element of this array is the lowest score.

The last thing this module does is to create a report which includes all the information about all the students in descending order, by their final grade scores, and the summary of the statistics.

The pseudocode for this module is as follows. Comments have been included to explain what each part of the module accomplishes.

/*comment: This section sorts the scores in descending order. It also puts the parallel arrays into the same order and identifies the highest and lowest scores */
Declare Flag As Integer
Declare K As Integer
Declare TempID As Integer
Declare TempFinal As Float
Declare TempName As String
Declare TempGrade As Character
Set Flag = 0
While Flag == 0
Set Flag = 1
For (K = 0; K < StudentCount – 2; K++)
If Final[K] < Final[K+1] Then
//swap for Final[] array
Set TempFinal = Final[K]
Set Final[K] = Final[K + 1]
Set Final[K + 1] = TempFinal
//swap for IDNum[] array
Set TempID = IDNum[K]
Set IDNum[K] = IDNum[K + 1]
Set IDNum[K + 1] = TempID
//swap for Names[] array
Set TempName = Names[K]
Set Names[K] = Names[K + 1]
Set Names[K + 1] = TempName
//swap for Grade[] array
Set TempGrade = Grade[K]
Set Grade[K] = Grade[K + 1]
Set Grade[K + 1] = TempGrade
Set Flag = 0
End If
End For
Declare HighScore As Float
Declare LowScore As Float
Declare Sum As Float
Declare NumAbove As Integer
Declare NumBelow As Integer
Declare NumAvg As Integer
End While
Set HighScore = Final[0]
Set LowScore = Final[StudentCount – 1]
/*comment: This section computes the average score and determines how many scores fall at, above, and below that average. */
Set Sum = 0
Set NumAbove = 0
Set NumBelow = 0
Set NumAvg = 0
Set K = 0
While K < StudentCount //loop to get sum of all grades
Set Sum = Sum + Final[K]
Set K = K + 1
End While
Set ClassAvg = Sum/StudentCount //compute average
For (K = 0; K < StudentCount; K++) //loop to count number > & < mean
If Final[K] > ClassAvg Then
Set NumAbove = NumAbove + 1
Else
    If Final[K] < ClassAvg Then
        Set NumBelow = NumBelow + 1
    Else
        Set NumAvg = NumAvg + 1
    End If
End If
End For
//comment: This section displays the report
//comment: The "/t" indicates a tab
Write "Student Name /t ID Number /t Final Score /t Letter Grade"
For (K = 0; K < StudentCount; K++)
    Write Names[K] + "/t" + IDNum[K] + "/t" + Final[K] + "/t" + Grade[K]
End For
Write "The average score for this class is: " + ClassAvg
Write "High score for the class: " + HighScore
Write "Low score for the class: " + LowScore
Write "Number of scores above the average: " + NumAbove
Write "Number of scores below the average: " + NumBelow
Write "Number of scores equal to the average: " + NumAvg

Program Code

The program code is now written using the design as a guide. At this stage, header comments and step comments are inserted into each module, providing internal documentation for the program. The following are several important points concerning the coding:

- The welcome message should be displayed on a blank screen using the programming language’s clear screen statement.
- To produce a professional report similar to the one shown in Figure 6.8, we must format the output to ensure that the data in the report lines up in columns and that the numbers displayed have a consistent decimal representation. For a report such as this, unless Professor Hirsch specifies otherwise, the final scores for each student should be rounded to the nearest tenth or hundredth. This can be accomplished using the programming language’s special print formatting statements.
- This program, as written, does not contain any error checking. Before the program is submitted to Professor Hirsch, error traps and error checking code should be added. The Self Check questions for this section will ask you to add some error checking.

Program Test

This program can be tested as written by entering data for a small class—perhaps just three or four students. The statistics should be calculated by hand and checked against the program’s results. A more extensive test can be done by entering exactly the same data as shown in Figure 6.8 and comparing the results.
However, in a real programming environment, the code would include error traps in many places. Typical errors that the program should catch would include:

- Data entered as an incorrect data type, such as a name entered for an ID number
- Negative numbers entered for numerical grades

The program should also be tested to see what would happen if the user attempted to enter more than the 50 allowed entries.

**Self Check for Section 6.6**

*Self Check Questions 6.27–6.30 refer to the Grade Management program of this section.*

6.27 Add code to the Enter_Info module to ensure that the user enters no more than 50 students.

6.28 Add code to the Enter_Info module to ensure that the user enters an appropriate ID number (an integer) and an appropriate score (a floating point number).

6.29 Since ClassAvg has been declared as a Float, it is unlikely that there will be many student averages which match the ClassAvg exactly. Add code to the Statistics Module to calculate how many grades fall within a range of ClassAvg. The range should be between the Floor() of the ClassAvg and the Ceiling() of ClassAvg. (The Ceiling Function is discussed in Chapter 4.)

6.30 Add code to the Statistics module to check that there are actually some scores in the Final array. If there are no scores in this array, a division by zero error would occur so be sure your code will prevent the program from attempting to calculate an average in this case.
Chapter Summary

In this chapter, we discussed the following topics:

1. One-dimensional arrays as follows:
   - The Declare statement is used to define a one-dimensional array
   - Arrays and parallel arrays are used in input, processing, and output operations
   - There are many advantages to using arrays, including reducing the number of program variables and creating more efficient and more general programs

2. Searching and sorting a one-dimensional array as follows:
   - The serial search examines array elements in order, one-by-one, until the desired element is found
   - The bubble sort makes passes through the array, comparing consecutive pairs of elements on each pass and interchanging them if they are not in the correct order

3. More efficient search and sort procedures as follows:
   - Pseudocode for a binary search for the element Key in an array called Array which has N + 1 elements and is sorted in either ascending or descending order, with the highest subscript in the array equal to N.
   - Pseudocode for a selection sort of an array called Array which has N + 1 elements and is sorted in ascending order, with the highest subscript in the array equal to N.

4. Viewing strings as arrays of characters as follows:
   - A String variable can be declared as an array with elements that are of Character type
   - The Length_Of() function is used to find the length of a String
   - Strings can be manipulated by examining the array in which the string is located

5. Two-dimensional arrays as follows:
   - Two-dimensional arrays are declared as Array[X,Y]
   - Two-dimensional arrays are used in input, processing, and output operations

Review Questions

Fill in the Blank

1. A variable that is used to indicate whether or not a certain action has taken place is called a(n) __________.

2. In performing a bubble sort to arrange the numbers 5, 30, 25, 15 in ascending order, __________ interchanges will take place.

3. The Declare statement allocates __________ storage locations (bytes) to the array Name[25].

4. The length of the string called MyName which contains the name Arnold is __________.
5. If the arrays \( \text{Score1} \), \( \text{Score2} \), and \( \text{Score3} \) have the same size and corresponding elements contain related data, they are said to be ________ arrays.

**True or False**

6. T F The elements of an array are stored in consecutive storage locations in the computer’s internal memory.

7. T F One advantage of using subscripted variables (an array) is that they take up fewer storage locations than the same number of unsubscripted variables.

8. T F An array may have some elements that are numbers and other elements that are strings.

9. T F If a declaration statement allocates 100 storage locations to an array, the program must assign values to all 100 elements.

10. T F The following statement:
    ```
    Declare Array1[10], Array2[20] As Integer
    ```
    allocates storage space for 200 variables.

11. T F In two parallel arrays, all corresponding elements must be of the same data type.

12. T F Before using the serial search method, you must sort the table keys in ascending order.

13. T F The bubble sort method cannot be used to arrange numeric data in descending order.

14. T F The binary search procedure can only be used with numeric data.

15. T F A binary search requires that the data be sorted before beginning the search.

16. T F The length of a string in a Character array that has been declared as:
    ```
    Declare Chr[10] As Character
    ```
    is always 10.

17. T F One- and two-dimensional arrays may be declared in the same statement.

18. T F If we know that 100 elements have been allocated to the two-dimensional array \( \text{A} \), then both subscripts of \( \text{A} \) must run from 0 to 9; that is, \( \text{A} \) must have 10 rows and 10 columns.

**Short Answer**

19. Write a program segment that inputs up to 25 whole numbers (integers) from the user, terminated by 0, into an array called \( \text{Numbers} \).

20. Write a program segment that displays the contents of a previously declared array of strings called \( \text{Names} \). Assume that the last entry in \( \text{Names} \) is “ZZZ”, which should not be displayed.
21. What is the output of code corresponding to the following pseudocode?

```
Declare N As Integer
Declare K As Integer
Declare X[100] As Integer
Set N = 4
Set K = 1
While K <= N
    Set X[K] = K^2
    Set K = K + 1
End While
Write X[N/2]
Write X[1] + " " + X[N – 1]
```

22. What is the output of code corresponding to the following pseudocode if
the user inputs 2, 3, 4, 5?

```
Declare Number As Integer
Declare Count As Integer
Declare Sums[5] As Integer
Set Count = 0
Set Sums[0] = 0
While Count < 4
    Write "Enter a number: 
    Input Number
    Set Sums[Count + 1] = Sums[Count] + Number
    Set Count = Count + 1
End While
While Count >= 0
    Write Sums[Count]
    Set Count = Count – 1
End While
```

23. What is the output of code corresponding to the following pseudocode?

```
Declare A[20] As Integer
Declare B[20] As Integer
Declare K As Integer
For (K = 1; K <= 3; K++)
    Set B[K] = K
End For
For (K = 1; K <= 3; K++)
    Write A[K] + " " + B[K]
End For
```

24. The following program segment is supposed to search an array A consisting
of N elements for a value Key and set Found equal to 1 or 0, depending on
whether or not Key is located. It contains two errors. Correct them. Assume
that the array A and the other variables have already been properly
declared.

Set Index = 0
Set Found = 0
While (Found == 1) AND (Index < N)
  If A[Index] == Key Then
    Set Found = 0
    Set Index = Index + 1
  End If
End While

25. In Exercise 24, which variable is the flag for this program segment?

26. What are the values of A[K] and A[K+1] after code corresponding to the following pseudocode is run?
   Set A[K] = 10
   Set A[K+1] = 20
   Set Temp = A[K]
   Set A[K + 1] = Temp
   Write A[K]
   Write A[K + 1]

27. What are the values of A[K] and A[K+1] after code corresponding to the following pseudocode is run?
   Set A[K] = 10
   Set A[K+1] = 20
   Write A[K]
   Write A[K + 1]

28. The following program segment is supposed to sort an array A consisting of N numbers in ascending order. It contains two errors. Correct them. Assume that the array A and the other variables have already been properly declared.
   Set Flag = 0
   While Flag == 0
     Set Flag = 1
     For (K = 0; K <= StudentCount – 1; K++)
         Set Temp = A[K]
         Set A[K + 1] = Temp
         Set Flag = 1
       End If
     End For
   End While

Short Answer Questions 29 and 30 refer to the following pseudocode:

Declare Name[20] As Character
Declare K As Integer
Set K = 0
While K < 8
    Set Name[K] = "A"
    Set K = K + 1
End While
Set Name[8] = " "
Set Name[9] = "B"

29. Write a single statement that displays the first and last characters in the string Name.

30. Write a program segment that displays the characters in Name except the blank.

In Short Answer Questions 31–34, an array has been declared by the following:

Declare FullName[25] As Character
and contains a person’s first and last names, separated by a blank.

31. Write a statement that displays the length of the string in the array FullName.

32. Write a program segment that displays the number of letters in the person’s first name.

33. Write a program segment that displays the person’s initials, with a period after each initial.

34. Write a program segment that displays the person’s last name.

Short Answer Questions 35–38 refer to the following pseudocode used to perform a binary search of the names Arnold, Draper, Gomez, Johnson, Smith, Wong (stored in Array), for the name Gomez. (Assume the array and all the variables have already been declared with their appropriate data types.)

Set N = 5
Set Key = "Gomez"
Set Low = 0
Set High = N
Set Index = Int(N/2)
Set Found = 0
While (Found == 0) AND (Low <= High)
    If Key == Array[Index] Then
        Set Found = 1
    End If
    If Key > Array[Index] Then
        Set Low = Index + 1
        Set Index = Int((High+Low)/2)
    End If
    If Key < Array[Index] Then
        Set High = Index - 1
        Set Index = Int((High + Low)/2)
    End If
End While
35. On entering the While loop for the first time, what is the value of \textit{Index}?

36. After the first pass through the While loop, what are the values of \textit{Low} and \textit{High}?

37. How many passes are made through the While loop?

38. After the While loop is exited, what is the value of \textit{Found}?

Short Answer Questions 39–42 refer to the following pseudocode, which is used to perform a selection sort in ascending order on the names Wong, Smith, Johnson, Gomez, Draper, and Arnold (which are stored in \textit{Array}). (Assume the array and all the variables have already been declared with their appropriate data types.)

```
Set N = 5
For (K = 0; K <= N; K++)
    Set Min = \textit{Array}[K]
    Set Index = K
    For (J = K + 1; K < N; J++)
        If \textit{Array}[J] < Min Then
            Set Min = \textit{Array}[J]
            Set Index = J
        End If
    End For(J)
    If K != Index Then
        Set Temp = \textit{Array}[K]
        Set \textit{Array}[K] = \textit{Array}[Index]
        Set \textit{Array}[Index] = Temp
    End If
End For(K)
```

39. How many passes are made through the outer \textit{For} loop?

40. After the first pass through the first inner \textit{For} loop, what is the value of \textit{Index}?

41. After the first pass through the outer \textit{For} loop, which name is stored in \textit{Array}[1]?

42. What changes must be made to this pseudocode to sort an array of six names in descending order?

43. Write a program segment that declares a two-dimensional array \textit{X} of integers with five rows and five columns and inputs 25 integers into this array from the user.

44. Write a program segment that sums the elements in each row of the array \textit{X} of Exercise 43 and displays these five numbers.

45. What is the output of code corresponding to the following pseudocode?

```
Declare \textit{Q}[10,10] As Integer
Declare R, C, As Integer
For (R = 1; R <= 3; R++)
    For (C = 1; C <= 3; C++)
        If R == C Then
            Set Q[R,C] = 1
```

Else
    Set Q[R,C] = 0
End If
End For(C)
End For(R)
For (R = 1; R <= 3; R++)
    For (C = 1; C <= 3; C++)
        Write Q[R,C]
    End For(C)
End For(R)

Programming Problems

For each of the following Programming Problems, use the top-down modular approach and pseudocode to design a suitable program to solve it.

1. Input a list of positive numbers, terminated by 0, into an array Numbers[]. Then display the array and the largest and smallest number in it.

2. Nancy and Ned Norton run a children's soccer camp. This year they want to create 3 leagues: PeeWee, Junior, and Senior. Sixty children have signed up and the Nortons need to find age ranges for the leagues. Create a program that will allow them to load the children's ages in an array named Ages[] and will sort the ages in ascending order. Then, by identifying the youngest and oldest, find the range of ages represented. Finally, create age cutoffs for the three leagues. (Hint: the pseudocode from Example 6.15 will help you get started.) Your final display should look like this where the ?'s are replaced by the ages you calculate:

   PeeWee League: ages ?? through ??
   Junior League: ages ?? through ??
   Senior League: ages ?? through ??

3. If \( X[0], X[1], \ldots, X[N-1] \) is a list of \( N \) numbers and the mean (average) of these numbers is \( M \), then we define their standard deviation to be the square root of the number:

\[
\left( (X[0] - M)^2 + (X[1] - M)^2 + \ldots + (X[N] - M)^2 \right) / (N - 1)
\]

Use a loop to allow a user to input up to 10 numbers in an array. The user should indicate that he is done by entering 0. Find the mean (average) of these numbers. Then use the mean to find the standard deviation. Your output should display both the mean and the standard deviation.

Name your variables as follows:
- the array of numbers should be \( X[10] \)
- the first sum should be named \( \text{Sum1} \)
- the average (mean) should be \( \text{Mean} \)
- the second sum should be named \( \text{Sum2} \)
- the standard deviation should be \( \text{StandardDeviation} \)
- the number to be input by the user should be \( \text{Num} \)
- counters can be named as desired; I like \( \text{Count}, K, \) or \( J \)
If you name your variables as shown above, you will use the following formulas:

- to sum all the numbers to calculate the mean: \( \text{Sum} = \text{Sum} + X[K] \)
- to calculate the average: \( \text{Mean} = \text{Sum}/(\text{Count} - 1) \)
- to calculate the first part of the standard deviation, you need to find the difference between each number, \( X[K] \), and the mean and then square the result: \( (X[K] - \text{Mean})^2 \)
- then you need to sum all of these parts up so use this formula in a loop: \( \text{Sum2} = \text{Sum2} + (X[K] - \text{Mean})^2 \)
- the standard deviation is calculated with the following formula:
  \[
  \text{StandardDeviation} = \sqrt{\text{Sum2}/(\text{Count} - 1)}
  \]

4. Input a list of employee names and salaries, and determine the mean (average) salary as well as the number of salaries above and below the mean.

5. Determine the median selling price of all homes in a subdivision named Botany Bay sold during one year. Allow the user to enter the number of houses sold and store their selling prices in an array. The median of a list of \( N \) numbers is as follows:
   a. The middle number of the sorted list, if \( N \) is odd
   b. The average of the two middle numbers in the sorted list, if \( N \) is even
   (Hint: after inputting the prices into an array, sort that array.)

6. A magic square is a two-dimensional array of positive integers in which the following is true:
   - The number of rows equals the number of columns
   - Every row, every column, and the two diagonals add up to the same number

   Input a two-dimensional array with four rows and four columns and determine if it is a magic square.

   (Hint: If we call the array Magic, then the sums of the elements on the two diagonals are:
   \[
   \text{Diagonal1} = \text{Magic}[1,1] + \text{Magic}[2,2] + \text{Magic}[3,3] + \text{Magic}[4,4]
   \]
   \[
   \text{Diagonal2} = \text{Magic}[1,4] + \text{Magic}[2,3] + \text{Magic}[3,2] + \text{Magic}[4,1]
   \]