7

Arrays

Objectives

- To introduce the array data structure.
- To understand how arrays store, sort and search lists and tables of values.
- To understand how to declare an array, initialize an array and refer to individual elements of an array.
- To be able to pass arrays to methods.
- To understand basic sorting techniques.
- To be able to declare and manipulate multiplesubscript arrays.

With sobs and tears he sorted out Those of the largest size ...

Lewis Carroll

Attempt the end, and never stand to doubt; Nothing's so hard, but search will find it out. Robert Herrick

Now go, write it before them in a table, and note it in a book.

Isaiah 30:8

'Tis in my memory lock'd, And you yourself shall keep the key of it. William Shakespeare



Outline

- 7.1 Introduction
- 7.2 Arrays
- 7.3 Declaring and Allocating Arrays
- 7.4 Examples Using Arrays
 - 7.4.1 Allocating an Array and Initializing Its Elements
 - 7.4.2 Totaling the Elements of an Array
 - 7.4.5 Using Arrays to Analyze Survey Results
 - 7.4.3 Using Histograms to Display Array Data Graphically
 - 7.4.4 Using the Elements of an Array as Counters
 - 7.4.5 Using Arrays to Analyze Survey Results
- 7.5 Passing Arrays to Methods
- 7.6 Passing Arrays by Value and by Reference
- 7.7 Sorting Arrays
- 7.8 Searching Arrays: Linear Search and Binary Search
 - 7.8.1 Searching an Array with Linear Search
 - 7.8.2 Searching a Sorted Array with Binary Search
- 7.9 Multiple-Subscripted Arrays
- 7.10 foreach Repetition Structure

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

7.1 Introduction

This chapter serves as an introduction to data structures. *Arrays* are data structures consisting of data items of the same type. Arrays are "static" entities, in that they remain the same size once they are created. We begin by learning about creating and accessing arrays, then use this knowledge to begin more complex manipulations of arrays, including powerful searching and sorting techniques. We then demonstrate creating more sophisticated arrays that have multiple dimensions. Chapter 24, Data Structures, introduces dynamic data structures such as lists, queues, stacks and trees that can grow and shrink as programs execute. We also introduce C#'s predefined data structures that enable the programmer to use existing data structures for lists, queues, stacks and trees, rather than having to "reinvent the wheel."

7.2 Arrays

An array is a group of contiguous memory locations that all have the same name and type. To refer to a particular location or element in the array, we specify the name of the array and the *position number* (a value that indicates a specific location within the array) of the element to which we refer.

Figure 7.1 shows an integer array called \mathbf{c} . This array contains 12 *elements*. A program can refer to any element of an array by giving the name of the array followed by the position

number of the element in square brackets ([]). The first element in every array is the *zeroth element*. Thus, the first element of array \mathbf{c} is referred to as \mathbf{c} [0], the second element of array \mathbf{c} is referred to as \mathbf{c} [6] and so on. The *i*th element of array \mathbf{c} is referred to as \mathbf{c} [6] and so on. The *i*th element of array \mathbf{c} is referred to as \mathbf{c} [i - 1]. Array names follow the same conventions as other variable names, as discussed in Chapter 3, Introduction to C# Programming.

The position number in square brackets is more formally called a *subscript* (or an *index*). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, the program evaluates the expression first to determine the subscript. For example, if variable **a** is equal to **5** and variable **b** is equal to **6**, then the statement

adds 2 to array element **c[11]**. Note that a subscripted array name is an *lvalue*—it can be used on the left side of an assignment to place a new value into an array element.

Let us examine array c in Fig. 7.1 more closely. The name of the array is c. Every array in C# "knows" its own length. The length of the array is determined by the expression:

c.Length

The array's 12 elements are referred to as c[0], c[1], c[2], ..., c[11]. The value of c[0] is -45, the value of c[1] is 6, the value of c[2] is 0, the value of c[7] is 62 and the value of c[11] is 78. To calculate the sum of the values contained in the first three elements of array c and to store the result in variable sum, we would write

$$sum = c[0] + c[1] + c[2];$$

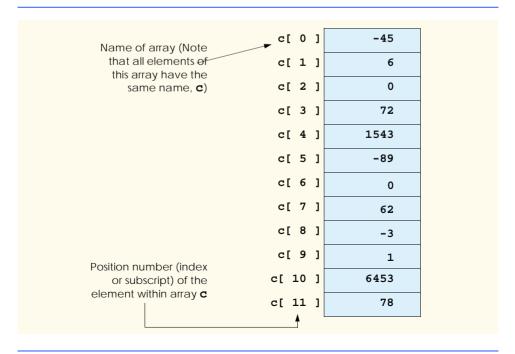


Fig. 7.1 A 12-element array.

To divide the value of the seventh element of array \mathbf{c} by $\mathbf{2}$ and assign the result to the variable \mathbf{x} , we would write

x = c[6] / 2;

Common Programming Error 7.1

It is important to note the difference between the "seventh element of the array" and "array element seven." Array subscripts begin at 0, thus the "seventh element of the array" has a subscript of 6, while "array element seven" has a subscript of 7 and is actually the eighth element of the array. This confusion is a source of "off-by-one" errors.

The brackets that enclose the subscript of an array are operators. Brackets have the same level of precedence as parentheses. The chart in Fig. 7.2 shows the precedence and associativity of the operators introduced to this point in the text. They are displayed top to bottom in decreasing order of precedence, with their associativity and type. The reader should note that the ++ and -- operators in the first row represent the postincrement and postdecrement operators, while the ++ and -- operators in the second row represent the preincrement and predecrement operators. Also, notice that in the first row the associativity is mixed. This is because the associativity of the postincrement and postdecrement operators is right to left, while the associativity for the other operators is left to right.

7.3 Declaring and Allocating Arrays

Arrays occupy space in memory. The programmer specifies the type of the elements and uses operator **new** to allocate dynamically the number of elements required by each array. Arrays are allocated with **new** because arrays are objects and all objects must be created with **new**. We will see an exception to this rule shortly.

Operators	Associativity	Туре
()[].++	left to right	highest (unary postfix)
++ + - ! (type)	right to left	unary (unary prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	logical AND
*	left to right	logical exclusive OR
1	left to right	logical inclusive OR
&&	left to right	conditional AND
П	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 7.2 Precedence and associativity of the operators discussed so far.

The declaration

```
int[] c = new int[ 12 ];
```

allocates 12 elements for integer array \mathbf{c} . The preceding statement can also be performed in two steps as follows:

When arrays are allocated, the elements are initialized to zero for the numeric primitive-data-type variables, to **false** for **bool** variables and to **null** for reference types.

Common Programming Error 7.2

Unlike in C or C++, in C# the number of elements in the array is never specified in the square brackets after the array name. The declaration int[12]c; causes a syntax error.

Memory may be reserved for several arrays with a single declaration. The following declaration reserves 100 elements for **string** array **b** and 27 elements for **string** array **x**:

```
string[] b = new string[ 100 ], x = new string[ 27 ];
```

Similarly, the following declaration reserves 10 elements for **array1** and 20 elements for **array2** (both of type **double**):

Arrays may be declared to contain most data types. In an array of value types, every element of the array contains one value of the declared type. For example, every element of an **int** array is an **int** value.

In an array of reference types, every element of the array is a reference to an object of the data type of the array. For example, every element of a **string** array is a reference to a **string**. Each of these **string** references has the value **null** by default.

7.4 Examples Using Arrays

This section presents several examples using arrays that demonstrate declaring arrays, allocating arrays, initializing arrays and manipulating array elements in various ways. For simplicity, the examples in this section use arrays that contain elements of type **int**. Please remember that a program can declare arrays of most data types.

7.4.1 Allocating an Array and Initializing Its Elements

Figure 7.3 creates three integer arrays of 10 elements and displays those arrays in tabular format. The program demonstrates several techniques for declaring and initializing arrays.

```
1  // Fig 7.3: InitArray.cs
2  // Different ways of initializing arrays.
3
4  using System;
5  using System.Windows.Forms;
```

Fig. 7.3 Initializing element arrays in three different ways. (Part 1 of 2.)

```
6
7
    class InitArray
8
9
       // main entry point for application
10
       static void Main( string[] args )
11
12
          string output = "";
13
14
          int[] x;
                                // declare reference to an array
15
          x = new int[ 10 ]; // dynamically allocate array and set
16
                                // default values
17
18
          // initializer list specifies number of elements
19
           // and value of each element
20
          int[] y = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
21
22
          const int ARRAY SIZE = 10; // named constant
23
          int[] z;
                                        // reference to int array
24
25
          // allocate array of ARRAY SIZE (i.e., 10) elements
26
          z = new int[ ARRAY SIZE ];
27
28
          // set the values in the array
29
          for ( int i = 0; i < z.Length; i++)
30
              z[i] = 2 + 2 * i;
31
32
          output += "Subscript\tArray x\tArray y\tArray z\n";
33
34
          // output values for each array
35
          for ( int i = 0; i < ARRAY_SIZE; i++ )</pre>
36
              output += i + "\t" + x[i] + "\t" + y[i] +
37
                 "\t" + z[i] + "\n";
38
39
          MessageBox.Show( output,
40
              "Initializing an array of int values",
41
              MessageBoxButtons.OK, MessageBoxIcon.Information );
42
43
       } // end Main
44
45
    } // end class InitArray
                        Initializing an array of int values
                                                ×
                               Subscript Array x
                                         Anay y
                                              Array z
                                         27
                                    0
                                    ŏ
                                         64
                                    Ü
                                         18
                                              8
                                    0
                                         95
                                              10
                                    Ö
                                              12
                               6
                                    D
                                         90
                                              14
                                         70
                                    D
                                              16
                                    0
                                              18
                                    D
                                     0K
```

Fig. 7.3 Initializing element arrays in three different ways. (Part 2 of 2.)

Line 14 declares **x** as a reference to an array of integers. Each element in the array is of type **int**. The variable **x** is of type **int**[], which denotes an array whose elements are of type **int**. Line 15 allocates the 10 elements of the array with **new** and assigns the array to reference **x**. Each element of this array has the default value 0.

Line 20 creates another **int** array and initializes each element using an *initializer list*. In this case, the number of elements in the initializer list determines the array's size. For example, line 20 creates a 10-element array with the indices **0–9** and the values **32**, **27**, **64**, and so on. Note that this declaration does not require the **new** operator to create the array object—the compiler allocates memory for the object when it encounters an array declaration that includes an initializer list.

On line 22, we create constant integer **ARRAY_SIZE** using keyword **const**. A constant must be initialized in the same statement where it is declared and cannot be modified thereafter. If an attempt is made to modify a **const** variable after it is declared, the compiler issues a syntax error.

Constants also are called *named constants*. They often are used to make a program more readable and are usually denoted with variable names in all capital letters.

Common Programming Error 7.3



On lines 23 and 26, we create integer array **z** of length 10 using the **ARRAY_SIZE** named constant. The **for** structure in lines 29–30 initializes each element in array **z**. The values are generated by multiplying each successive value of the loop counter by **2** and adding **2** to the product. After this initialization, array **z** contains the even integers **2**, **4**, **6**, ..., **20**. The **for** structure in lines 35–37 uses the values in arrays **x**, **y** and **z** to build an output string, which will be displayed in a **MessageBox**. Zero-based counting (remember, array subscripts start at 0) allows the loop to access every element of the array. The constant **ARRAY_SIZE** in the **for** structure condition (line 29) specifies the arrays' lengths.

7.4.2 Totaling the Elements of an Array

Often, the elements of an array represent series of values to be used in calculations. For example, if the elements of an array represent the grades for an exam in a class, the professor may wish to total the elements of an array, then calculate the class average for the exam.

The application in Fig. 7.4 sums the values contained in the 10-element integer array **a** (declared, allocated and initialized on line 12). Line 16 in the body of the **for** loop performs the addition using the array element at position **i** during each loop iteration. Note that the values being supplied as initializers for array **a** normally would be read into the program. For example, in a Windows application, the user could enter the values through a **TextBox**, or the values could be read from a file on disk. (See Chapter 17, Files and Streams.)

```
1  // Fig. 7.4: SumArray.cs
2  // Computing the sum of the elements in an array.
3
4  using System;
5  using System.Windows.Forms;
```

Fig. 7.4 Computing the sum of the elements of an array. (Part 1 of 2.)

```
6
 7
    class SumArray
 8
 9
        // main entry point for application
10
        static void Main( string[] args )
11
12
           int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
           int total = 0;
14
15
           for ( int i = 0; i < a.Length; i++ )</pre>
16
              total += a[ i ];
17
18
           MessageBox.Show( "Total of array elements: " + total,
19
              "Sum the elements of an array",
20
              MessageBoxButtons.OK, MessageBoxIcon.Information );
21
22
        } // end Main
23
24
    } // end class SumArray
                           Sum the elements of an array
                                  Total of array elements: 55
                                      DK
```

Fig. 7.4 Computing the sum of the elements of an array. (Part 2 of 2.)

7.4.3 Using Histograms to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values often are displayed as bars in a bar chart. In such a chart, longer bars represent larger numeric values. One simple way to display numeric data graphically is with a *histogram* that shows each numeric value as a bar of asterisks (*).

Our next application (Fig. 7.5) reads numbers from an array and graphs the information in the form of a bar chart, or histogram. The program displays each number followed by a bar consisting of a corresponding number of asterisks. The nested **for** loops (lines 18–24) append the bars to the **string** that will be displayed in the **MessageBox**. Note the loop continuation condition of the inner **for** structure on line 22 (**j <= n[i]**). Each time the program reaches the inner **for** structure, the loop counts from **1** to **n[i]**, using a value in array **n** to determine the final value of the control variable **j** and the number of asterisks to display.

```
1  // Fig. 7.6: Histogram.cs
2  // Using data to create a histogram.
3
4  using System;
5  using System.Windows.Forms;
```

Fig. 7.5 Program that prints histograms. (Part 1 of 2.)

```
6
 7
    class Histogram
 8
 9
        // main entry point for application
10
        static void Main( string[] args )
11
12
           int[] n = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
13
           string output = "";
14
15
           output += "Element\tvalue\tHistogram\n";
16
17
           // build output
18
           for ( int i = 0; i < n.Length; i++ )</pre>
19
20
              output += "\n" + i + "\t" + n[i] + "\t";
21
22
              for ( int j = 1; j <= n[ i ]; j++ ) // print a bar</pre>
23
                  output += "*";
24
           }
25
26
           MessageBox. Show( output, "Histogram Printing Program",
27
              MessageBoxButtons.OK, MessageBoxIcon.Information );
28
29
        } // end Main
30
31
    } // end class Histogram
                        Histogram Printing Program
                                                   ×
                                Element value
                                           Histogram
                          i
                                     19
                                     15
                                     11
                                     13
                                8
                                     17
                                     OK
```

Fig. 7.5 Program that prints histograms. (Part 2 of 2.)

7.4.4 Using the Elements of an Array as Counters

Sometimes programs use a series of counter variables to summarize data, such as the results of a survey. In Chapter 6, Methods, we used a series of counters in our dice-rolling program to track the number of occurrences of each side on a six-sided die as the program rolled 12 dice at a time. We also indicated that there is a more elegant method than that in Fig. 6.11 for writing the dice-rolling program. An array version of this application is shown in Fig. 7.6.

The program uses the seven-element array frequency to count the occurrences of each side of the die. Line 94, which uses the random **face** value as the subscript for array **fre-**

quency to determine which element should be incremented during each iteration of the loop, replaces lines 95–115 of Fig. 6.11. The random number calculation on line 88 produces numbers 1–6 (the values for a six-sided die); thus, the **frequency** array must be large enough to allow subscript values of 1–6. The smallest number of elements required for an array to have these subscript values is seven elements (subscript values 0–6). In this program, we ignore element 0 of array **frequency**. Lines 75–80 replace lines 69–81 from Fig. 6.11. We can loop through array **frequency**; therefore, we do not have to enumerate each line of text to display in the **Label**, as we did in Fig. 6.11.

```
1
    // Fig. 7.7: RollDie.cs
 2
    // Rolling 12 dice.
 3
 4 using System;
 5
   using System.Drawing;
    using System.Collections;
 7
    using System.ComponentModel;
 8
    using System.Windows.Forms;
 9
    using System.Data;
10
    using System.IO;
11
12
    public class RollDie : System.Windows.Forms.Form
13
14
       private System.Windows.Forms.Button rollButton;
15
16
       private System.Windows.Forms.RichTextBox displayTextBox;
17
18
       private System.Windows.Forms.Label dieLabel1;
19
       private System.Windows.Forms.Label dieLabel2;
20
       private System.Windows.Forms.Label dieLabel3;
21
       private System.Windows.Forms.Label dieLabel4;
22
       private System.Windows.Forms.Label dieLabel5;
23
       private System.Windows.Forms.Label dieLabel6;
24
       private System.Windows.Forms.Label dieLabel7;
25
       private System.Windows.Forms.Label dieLabel8;
26
       private System.Windows.Forms.Label dieLabel9;
27
       private System.Windows.Forms.Label dieLabel10;
28
       private System.Windows.Forms.Label dieLabel11;
29
       private System.Windows.Forms.Label dieLabel12;
30
31
       private System.ComponentModel.Container components = null;
32
33
       Random randomNumber = new Random();
34
       int[] frequency = new int[ 7 ];
35
36
       public RollDie()
37
38
          InitializeComponent();
39
       }
40
41
       // Visual Studio .NET generated code
42
```

Fig. 7.6 Using arrays to eliminate a **switch** structure. (Part 1 of 3.)

```
43
       [STAThread]
44
       static void Main()
45
46
          Application.Run( new RollDie() );
47
       }
48
49
       private void rollButton Click(
50
          object sender, System.EventArgs e )
51
       {
52
          // pass the labels to a method that will
53
          // randomly assign a face to each die
54
          DisplayDie( dieLabel1 );
55
          DisplayDie( dieLabel2 );
56
          DisplayDie( dieLabel3 );
57
          DisplayDie( dieLabel4 );
58
          DisplayDie( dieLabel5 );
59
          DisplayDie( dieLabel6 );
60
          DisplayDie( dieLabel7 );
61
          DisplayDie( dieLabel8 );
62
          DisplayDie( dieLabel9 );
63
          DisplayDie( dieLabel10 );
64
          DisplayDie( dieLabel11 );
65
          DisplayDie( dieLabel12 );
66
67
          double total = 0;
68
69
          for ( int i = 1; i < 7; i++ )
70
             total += frequency[ i ];
71
72
          displayTextBox.Text = "Face\tFrequency\tPercent\n";
73
74
          // output frequency values
75
          for ( int x = 1; x < frequency.Length; x++ )</pre>
76
          {
77
             displayTextBox.Text += x + "\t" +
78
                 frequency[ x ] + "\t\t" + String.Format( "{0:N}",
79
                 frequency[ x ] / total * 100 ) + "%\n";
80
          }
81
82
       } // end Main
83
84
       // simulates roll, display proper
85
       // image and increment frequency
86
       public void DisplayDie( Label dieLabel )
87
       {
88
          int face = randomNumber.Next( 1, 7 );
89
90
          dieLabel.Image = Image.FromFile(
91
             Directory.GetCurrentDirectory() +
92
              "\\images\\die" + face + ".gif" );
93
94
          frequency[ face ]++;
95
       }
```

Fig. 7.6 Using arrays to eliminate a **switch** structure. (Part 2 of 3.)

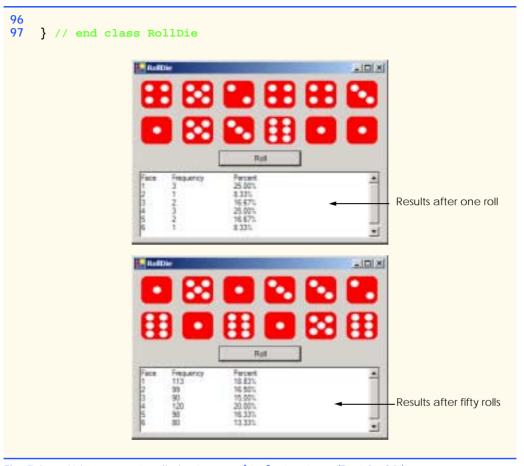


Fig. 7.6 Using arrays to eliminate a **switch** structure. (Part 3 of 3.)

7.4.5 Using Arrays to Analyze Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the following problem statement:

Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10, with 1 being awful and 10 being excellent. Place the 40 responses in an integer array and summarize the frequency for each rating.

This is a typical array processing application (Fig. 7.7). We wish to summarize the number of responses of each type (i.e., 1–10). The array **responses** is a 40-element integer array of the students' responses to the survey. We use an 11-element array **frequency** to count the number of occurrences of each response. We ignore the first element, **frequency** [0], because it is more logical to have a response of 1 increment **frequency** [1] than **frequency** [0]. We can use each response directly as a subscript on the **frequency** array. Each element of the array is used as a counter for one of the survey responses.

```
// Fig. 7.5: StudentPoll.cs
 2
    // A student poll program.
 3
 4
    using System;
 5
    using System.Windows.Forms;
 6
 7
    class StudentPoll
 8
9
       // main entry point for application
10
       static void Main( string[] args )
11
12
          int[] responses = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1,
13
              6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
14
              5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
15
16
          int[] frequency = new int[ 11 ];
17
          string output = "";
18
19
           // increment the frequency for each response
20
          for ( int answer = 0; answer < responses.Length; answer++ )</pre>
21
              ++frequency[ responses[ answer ] ];
22
23
          output += "Rating\tFrequency\n";
24
25
           // output results
26
          for ( int rating = 1; rating < frequency.Length; rating++ )</pre>
27
              output += rating + "\t" + frequency[ rating ] + "\n";
28
29
          MessageBox.Show( output, "Student poll program",
30
              MessageBoxButtons.OK, MessageBoxIcon.Information );
31
32
       } // end method Main
33
34
    } // end class StudentPoll
                            Student poll program
                                        Frequency
                                  5
                                  8
                                  9
                                  10
                                    ÜK
```

Fig. 7.7 Simple student-poll analysis program.



Good Programming Practice 7.1

Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time for writing clearer programs.

The **for** loop (lines 20–21) takes the responses from the array **response** one at a time and increments one of the 10 counters in the **frequency** array (**frequency** 1 1 to **frequency** 10 1). The key statement in the loop is on line 21, which increments the appropriate counter in the **frequency** array, depending on the value of element **responses** [answer].

Let us consider several iterations of the for loop. When counter answer is 0, responses[answer] is the value of the first element of array responses (i.e., 1). In this case, the program interprets ++frequency[responses[answer]]; as ++frequency[1];, which increments array element one. In evaluating the expression, start with the value in the innermost set of square brackets (answer). Once you know the value of answer, plug that value into the expression and evaluate the next outer set of square brackets (responses[answer]). Use that value as the subscript for the frequency array to determine which counter to increment.

When answer is 1, responses[answer] is the value of the second element of array responses (i.e., 2), so the program interprets

```
++frequency[ responses[ answer ] ];
```

as **++frequency**[**2**];, which increments array element two (the third element of the array). When **answer** is **2**, **responses**[**answer**] is the value of the third element of array **responses** (i.e., **6**), so the program interprets

```
++frequency[ responses[ answer ] ];
```

as **++frequency**[**6]**;, which increments array element six (the seventh element of the array) and so on. Note that, regardless of the number of responses processed in the survey, only an 11-element array is required (ignoring element zero) to summarize the results, because all the response values are between 1 and 10, and the subscript values for an 11-element array are 0–10. The results are correct, because the elements of the **frequency** array were initialized to zero when the array was allocated with **new**.

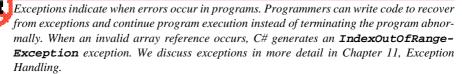
If the data contained invalid values, such as 13, the program would attempt to add 1 to **frequency[13]**. This is outside the bounds of the array. In the C and C++ programming languages, no checks are performed to prevent programs from reading data outside the bounds of arrays. At execution time, the program would "walk" past the end of the array to where element number 13 would be located and add 1 to whatever data are stored at that location in memory. This could potentially modify another variable in the program or even result in premature program termination. The .NET framework provides mechanisms to prevent accessing elements outside the bounds of arrays.

V

Testing and Debugging Tip 7.1

When a C# program executes, array element subscripts are checked for validity (i.e., all subscripts must be greater than or equal to 0 and less than the length of the array).

Testing and Debugging Tip 7.2





Common Programming Error 7.4

Referring to an element outside the array bounds is a logic error.



Testing and Debugging Tip 7.3

When looping through an array, the array subscript never should go below 0 and should always be less than the total number of elements in the array (one less than the length of the array). The loop-terminating condition should prevent accessing elements outside this range.



Testing and Debugging Tip 7.4

Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.

7.5 Passing Arrays to Methods

To pass an array argument to a method, specify the name of the array without using brackets. For example, if array **hourlyTemperatures** declared as

```
int[] hourlyTemperatures = new int[ 24 ];
```

the method call

```
ModifyArray( hourlyTemperatures );
```

passes array **hourlyTemperatures** to method **ModifyArray**. Every array object "knows" its own size (via the **Length** instance variable), so when we pass an array object into a method, we do not pass the size of the array as an argument separately.

Although entire arrays are passed by reference, individual array elements of primitive data types are passed by value, the same way as simple variables are. (The objects referred to by individual elements of a nonprimitive-type array are still passed by reference.) Such simple single pieces of data are sometimes called *scalars* or *scalar quantities*. To pass an array element to a method, use the subscripted name of the array element as an argument in the method call.

For a method to receive an array through a method call, the method's parameter list must specify that an array will be received. For example, the method header for method **ModifyArray** might be written as

```
public void ModifyArray( int[] b )
```

indicating that **ModifyArray** expects to receive an integer array in parameter **b**. Arrays are passed by reference; when the called method uses the array name **b**, it refers to the actual array in the caller (array **hourlyTemperatures**).

The application in Fig. 7.8 demonstrates the difference between passing an entire array and passing an array element.

The **for** loop on lines 32–33 appends the five elements of integer array **a** to the **Text** property of **outputLabel**. Line 33 invokes method **ModifyArray** and passes to it array **a**. Method **ModifyArray** multiplies each element by 2. To illustrate that array **a**'s elements were modified, the **for** loop on lines 41–42 appends the five elements of integer array **a** to the **Text** property of **outputLabel**. As the screen capture indicates, the elements of **a** are modified by **ModifyArray**.

```
// Fig. 7.8: PassArray.cs
   // Passing arrays and individual elements to methods.
3 using System;
4 using System.Drawing;
5 using System.Collections;
   using System.ComponentModel;
7
    using System.Windows.Forms;
8
   using System.Data;
9
  public class PassArray : System.Windows.Forms.Form
10
11
12
       private System.Windows.Forms.Button showOutputButton;
13
       private System.Windows.Forms.Label outputLabel;
14
15
       // Visual Studio .NET generated code
16
17
       [STAThread]
18
       static void Main()
19
20
          Application.Run( new PassArray() );
21
       }
22
23
       private void showOutputButton_Click( object sender,
24
          System. EventArgs e )
25
       {
26
          int[] a = { 1, 2, 3, 4, 5 };
27
28
          outputLabel.Text = "Effects of passing entire array " +
29
             "call-by-reference:\n\nThe values of the original " +
30
             "array are:\n\t";
31
32
          for ( int i = 0; i < a.Length; i++ )
33
             34
35
          ModifyArray( a ); // array is passed by reference
36
37
          outputLabel.Text +=
38
             "\n\nThe values of the modified array are:\n\t";
39
40
          // display elements of array a
41
          for ( int i = 0; i < a.Length; i++ )</pre>
42
             outputLabel.Text += " + a[ i ];
43
44
          outputLabel.Text += "\n\nEffects of passing array " +
45
             "element call-by-value:\n\na[ 3 ] before " +
46
             "ModifyElement: " + a[ 3 ];
47
48
          // array element passed call-by-value
49
          ModifyElement( a[ 3 ] );
50
51
          outputLabel.Text +=
52
             "\na[ 3 ] after ModifyElement: " + a[ 3 ];
53
       }
```

Fig. 7.8 Passing arrays and individual array elements to methods. (Part 1 of 2.)

```
54
55
         // method modifies the array it receives,
56
         // original will be modified
         public void ModifyArray( int[] b )
57
58
59
             for ( int j = 0; j < b.Length; j++ )</pre>
60
                b[ j ] *= 2;
61
         }
62
63
         // method modifies the integer passed to it
64
         // original will not be modified
65
         public void ModifyElement( int e )
66
67
             outputLabel.Text +=
68
                "\nvalue received in ModifyElement: " + e;
69
70
             e *= 2;
71
72
             outputLabel.Text +=
73
                 "\nvalue calculated in ModifyElement: " + e;
74
         }
75
     }
                          PassArray
                                                         Show Output
                         Effects of passing entire array call-by-reference:
                          The values of the original array are:
                           1 2 3 4 5
                          The values of the modified array are:
                          Effects of passing array element call-by-value:
                          a[3] before ModifyElement: 8
                          value received in ModifyElement 8
                          value calculated in ModifyElement: 16
                          a[3] after ModifyElement: 8
```

Fig. 7.8 Passing arrays and individual array elements to methods. (Part 2 of 2.)

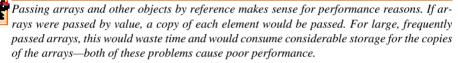
To show the value of a[3] before the call to ModifyElement, lines 44–46 append the value of a[3] (and other information) to outputLabel.Text. Line 44 invokes method ModifyElement and passes a[3]. Remember that a[3] is a single int value in the array a. Also, remember that values of primitive types always are passed to methods by value. Therefore, a copy of a[3] is passed. Method ModifyElement multiplies its argument by 2 and stores the result in its parameter e. The parameter of ModifyElement is a local variable, so when the method terminates, the local variable is destroyed. Thus, when control is returned to PassArray, the unmodified value of a[3] is appended to the outputLabel.Text (line 51–52).

7.6 Passing Arrays by Value and by Reference

In C#, a variable that "stores" an object, such as an array, does not actually store the object itself. Instead, such a variable stores a reference to the object (i.e., the location in the computer's memory where the object itself is stored). The distinction between reference variables and primitive data type variables raises some subtle issues that programmers must understand to create secure, stable programs.

When a program passes an argument to a method, the called method receives a copy of that argument's value. Changes to the local copy do not affect the original variable that the program passed to the method. If the argument is of a reference type, the method makes a local copy of the reference itself, not a copy of the actual object to which the reference refers. The local copy of the reference also refers to the original object in memory. Thus, reference types are always passed by reference, which means that changes to those objects in called methods affect the original objects in memory.

Performance Tip 7.1



C# also allows methods to pass references with keyword ref. This is a subtle capability, which, if misused, can lead to problems. For instance, when a reference-type object like an array is passed with ref, the called method actually gains control over the passed reference itself, allowing the called method to replace the original reference in the caller with a different object or even with null. Such behavior can lead to unpredictable effects, which can be disastrous in mission-critical applications. The program in Fig. 7.9 demonstrates the subtle difference between passing a reference by value and passing a reference with keyword ref.

Lines 26 and 29 declare two integer array variables, **firstArray** and **firstArray**-Copy (we make the copy so we can determine whether reference **firstArray** gets overwritten). Line 26 initializes **firstArray** with the values **1**, **2** and **3**. The assignment statement on line 29 copies reference **firstArray** to variable **firstArrayCopy**, causing these variables to reference the same array object in memory. The **for** structure on lines 38–39 prints the contents of **firstArray** before it is passed to method **FirstDouble** (line 42) so we can verify that this array is passed by reference (i.e., the called method indeed changes the array's contents).

The **for** structure in method **FirstDouble** (lines 99–100) multiplies the values of all the elements in the array by **2**. Line 103 allocates a new array containing the values **11**, **12** and **13**; the reference for this array then is assigned to parameter **array** (in an attempt to overwrite reference **firstArray**—this, of course, will not happen, because the reference was passed by value). After method **FirstDouble** executes, the **for** structure on lines 48–49 prints the contents of **firstArray**, demonstrating that the values of the elements have been changed by the method (and confirming that in C# arrays are always passed by reference). The **if/else** structure on lines 52–57 uses the **==** operator to compare references **firstArray** (which we just attempted to overwrite) and **firstArrayCopy**. The expression on line 40 evaluates to **true** if the operands to binary operator **==** indeed reference the same object. In this case, the object represented is the array allocated in line 26—not the array allocated in method **FirstDouble** (line 103).

```
1
    // Fig. 7.9: ArrayReferenceTest.cs
 2
    // Testing the effects of passing array references
   // by value and by reference.
 4 using System:
 5 using System.Drawing;
 6 using System.Collections;
    using System.ComponentModel;
 8 using System.Windows.Forms;
 9
   using System.Data;
10
11
   public class ArrayReferenceTest : System.Windows.Forms.Form
12
13
       private System.Windows.Forms.Label outputLabel;
14
       private System.Windows.Forms.Button showOutputButton;
15
16
       [STAThread]
17
       static void Main()
18
       {
19
          Application.Run( new ArrayReferenceTest() );
20
       }
21
22
       private void showOutputButton_Click( object sender,
23
          System.EventArgs e )
24
       {
25
          // create and initialize firstArray
26
          int[] firstArray = { 1, 2, 3 };
27
28
          // copy firstArray reference
29
          int[] firstArrayCopy = firstArray;
30
31
          outputLabel.Text =
32
             "Test passing firstArray reference by value";
33
34
          outputLabel.Text += "\n\nContents of firstArray " +
35
             "before calling FirstDouble:\n\t":
36
37
          // print contents of firstArray
38
          for ( int i = 0; i < firstArray.Length; i++ )</pre>
39
             outputLabel.Text += firstArray[ i ] + " ";
40
41
          // pass reference firstArray by value to FirstDouble
42
          FirstDouble( firstArray );
43
44
          outputLabel.Text += "\n\nContents of firstArray after " +
45
             "calling FirstDouble\n\t";
46
47
          // print contents of firstArray
48
          for ( int i = 0; i < firstArray.Length; i++ )</pre>
49
             outputLabel.Text += firstArray[ i ] + " ";
50
```

Fig. 7.9 Passing an array reference by value and by reference (Part 1 of 3.).

```
51
          // test whether reference was changed by FirstDouble
52
          if ( firstArray == firstArrayCopy )
53
             outputLabel.Text +=
54
                "\n\nThe references refer to the same array\n";
55
          else
56
             outputLabel.Text +=
57
                 "\n\nThe references refer to different arrays\n";
58
59
          // create and initialize secondArray
60
          int[] secondArray = { 1, 2, 3 };
61
62
          // copy secondArray reference
63
          int[] secondArrayCopy = secondArray;
64
65
          outputLabel.Text += "\nTest passing secondArray " +
66
             "reference by reference";
67
68
          outputLabel.Text += "\n\nContents of secondArray " +
69
             "before calling SecondDouble:\n\t";
70
71
          // print contents of secondArray before method call
72
          for ( int i = 0; i < secondArray.Length; i++ )</pre>
73
             outputLabel.Text += secondArray[ i ] + " ";
74
75
          SecondDouble( ref secondArray );
76
77
          outputLabel.Text += "\n\nContents of secondArray " +
78
             "after calling SecondDouble:\n\t";
79
80
          // print contents of secondArray after method call
81
          for ( int i = 0; i < secondArray.Length; i++ )</pre>
82
             outputLabel.Text += secondArray[ i ] + " ";
83
84
          // test whether reference was changed by SecondDouble
85
          if ( secondArray == secondArrayCopy )
86
             outputLabel.Text +=
87
                "\n\nThe references refer to the same array\n";
88
          else
89
             outputLabel.Text +=
90
                 "\n\nThe references refer to different arrays\n":
91
92
       } // end method showOutputButton Click
93
94
       // modify elements of array and attempt to modify
95
       // reference
96
       void FirstDouble( int[] array )
97
98
          // double each element's value
99
          for ( int i = 0; i < array.Length; i++ )</pre>
100
             array[ i ] *= 2;
101
```

Fig. 7.9 Passing an array reference by value and by reference (Part 2 of 3.).

```
102
              // create new reference and assign it to array
103
             array = new int[] { 11, 12, 13 };
104
          }
105
106
          // modify elements of array and change reference array
107
          // to refer to a new array
108
         void SecondDouble( ref int[] array )
109
110
              // double each element's value
111
             for ( int i = 0; i < array.Length; i++ )</pre>
112
                 array[ i ] *= 2;
113
114
             // create new reference and assign it to array
115
             array = new int[] { 11, 12, 13 };
         }
116
117 }
                           ArrayReferenceTest
                                                            _ | D | X |
                                         Show Output
                          Test passing firstArray reference by value
                          Contents of firstArray before calling FirstDouble.
                          Contents of firstArray after calling FirstDouble
                          246
                          The references refer to the same array
                          Test passing secondArray reference by reference
                          Contents of secondArray before calling
                          SecondDouble:
                          Contents of secondArray after calling SecondDouble:
                          11 12 13
                          The references refer to different arrays
```

Fig. 7.9 Passing an array reference by value and by reference (Part 3 of 3.).

Lines 60–90 perform similar tests, using array variables **secondArray** and **secondArrayCopy** and method **SecondDouble** (lines 108–116). Method **SecondDouble** performs the same operations as **FirstDouble**, but receives its array argument using keyword **ref**. In this case, the reference stored in **secondArray** after the method call is a reference to the array allocated on line 115 of **SecondDouble**, demonstrating that a reference passed with keyword **ref** can be modified by the called method so that the reference actually points to a different object, in this case an array allocated in procedure **SecondDouble**. The **if/else** structure in lines 85–90 demonstrates that **secondArray** and **secondArrayCopy** no longer refer to the same array.

Software Engineering Observation 7.1

When a method receives a reference-type object parameter by value, the object is not passed by value—the object still passes by reference. Rather, the object's reference is passed by value. This prevents a method from overwriting references passed to that method. In the vast majority of cases, protecting the caller's reference from modification is the desired behavior. If you encounter a situation where you truly want the called procedure to modify the caller's reference, pass the reference-type using keyword ref—but, again, such situations are rare.

Software Engineering Observation 7.2

In C#, reference-type objects (including arrays) always pass by reference. So, a called procedure receiving a reference to an object in a caller can change the caller's object.

7.7 Sorting Arrays

Sorting data (i.e., arranging the data into some particular order, such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name, and within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and in many cases, massive amounts of it. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the computer science field. In this section, we discuss one of the simplest sorting schemes. In the exercises, we investigate more sophisticated sorting algorithms.

Performance Tip 7.2

Sometimes, the simplest algorithms perform poorly. Their virtue is that they are easy to write, test and debug. Complex algorithms sometimes are needed to realize maximum performance of a program.

Figure 7.10 sorts the values of the 10-element array **a** into ascending order. The technique we use is called the *bubble sort*, because smaller values gradually "bubble" their way to the top of the array (i.e., toward the first element) like air bubbles rising in water. The technique sometimes is called the *sinking sort*, because the larger values sink to the bottom of the array. Bubble sort uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the values remain in the same order. If a pair is in decreasing order, the bubble sort swaps the values in the array. The program contains methods **Main**, **BubbleSort** and **Swap**. Method **sortButton_Click** (lines 23–41) creates array **a**, invokes **BubbleSort** and displays output. Line 34 of **sortButton_Click** invokes method **BubbleSort** (lines 44–52) to sort array **a**. Line 51 in method **BubbleSort** calls method **Swap** (lines 55–62) to exchange two elements of the array.

```
1  // Fig. 7.10: BubbleSorter.cs
2  // Sorting an array's values into ascending order.
3  using System;
4  using System.Drawing;
5  using System.Collections;
```

Fig. 7.10 Sorting an array with bubble sort. (Part 1 of 3.)

```
using System.ComponentModel;
 7
    using System.Windows.Forms;
8
    using System.Data;
9
10
   public class BubbleSorter : System.Windows.Forms.Form
11
12
       private System.Windows.Forms.Button sortButton;
13
       private System.Windows.Forms.Label outputLabel;
14
15
       // Visual Studio .NET generated code
16
17
       [STAThread]
18
       static void Main()
19
20
          Application.Run( new BubbleSorter() );
21
       }
22
23
       private void sortButton_Click( object sender,
24
             System.EventArgs e )
25
       {
26
          int[] a = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
27
28
          outputLabel.Text = "Data items in original order\n";
29
30
          for ( int i = 0; i < a.Length; i++ )</pre>
31
             outputLabel.Text += " " + a[ i ];
32
33
          // sort elements in array a
34
          BubbleSort( a );
35
36
          outputLabel.Text += "\n\nData items in ascending order\n";
37
38
          for ( int i = 0; i < a.Length; i++ )</pre>
39
             outputLabel.Text += "
                                     " + a[ i ];
40
41
       } // end method sortButton Click
42
43
       // sort the elements of an array with bubble sort
44
       public void BubbleSort( int[] b )
45
       {
46
          for ( int pass = 1; pass < b.Length; pass++ ) // passes</pre>
47
48
             for ( int i = 0; i < b.Length - 1; i++ )  // one pass</pre>
49
50
                if ( b[ i ] > b[ i + 1 ] )
                                                 // one comparison
51
                    Swap( b, i );
                                                  // one swap
52
       }
53
54
       // swap two elements of an array
55
       public void Swap( int[] c, int first )
56
57
          int hold;  // temporary holding area for swap
58
```

Fig. 7.10 Sorting an array with bubble sort. (Part 2 of 3.)

```
59
             hold = c[ first ];
60
             c[ first ] = c[ first + 1 ];
61
             c[ first + 1 ] = hold;
62
         }
63
     }
                                                         Sort
                          Data items in original order
                           2 6 4 8 10 12 89 68 45 37
                          Data items in ascending order
                           2 4 6 8 10 12 37 45 68 89
```

Fig. 7.10 Sorting an array with bubble sort. (Part 3 of 3.)

Method **BubbleSort** receives the array as parameter **b**. The nested **for** loop on lines 46–51 performs the sort. The outer loop controls the number of passes of the array. The inner loop controls the comparisons and necessary swapping of the elements during each pass.

Method BubbleSort first compares b[0] to b[1], then b[1] to b[2], then b[2] to b[3] and so on, until it completes the pass by comparing b[8] to b[9]. Although there are 10 elements, the comparison loop performs only nine comparisons. As a result of the way the successive comparisons are made, a large value may move down the array (sink) many positions (and sometimes all the way to the bottom of the array) on a single pass. However, a small value may move up (bubble) only one position. On the first pass, the largest value is guaranteed to sink to the bottom element of the array, b[9]. On the second pass, the second largest value is guaranteed to sink to b[8]. On the ninth pass, the ninth largest value sinks to b[1]. This leaves the smallest value in b[0], so only nine passes are needed to sort a 10-element array.

If a comparison reveals that the two elements appear in descending order, **BubbleSort** calls **Swap** to exchange the two elements so they will be in ascending order in the array. Method **Swap** receives a reference to the array (which it calls **c**) and one integer representing the subscript of the first element of the array to be exchanged. Three assignments on lines 59–61 perform the exchange, where the extra variable **hold** temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```
c[ first ] = c[ first + 1 ];
c[ first + 1 ] = c[ first ];
```

If c[first] is 7 and c[first + 1] is 5, after the first assignment, both elements of the array contain 5 and the value 7 is lost—hence, the need for the extra variable hold.

The advantage of the bubble sort is that it is easy to program. However, the bubble sort runs slowly, which becomes apparent when sorting large arrays. More advanced courses (often titled "Data Structures" or "Algorithms" or "Computational Complexity") investigate sorting and searching in greater depth. Note that the .NET framework includes a built-in array-sorting capability that implements a high-speed sort. To sort the array **a** in Fig. 7.10, you can use the statement

```
Array.Sort( a );
```

7.8 Searching Arrays: Linear Search and Binary Search

Often, programmers work with large amounts of data stored in arrays. It might be necessary in this case to determine whether an array contains a value that matches a certain *key value*. The process of locating a particular element value in an array is called *searching*. In this section, we discuss two searching techniques—the simple *linear search* technique and the more efficient *binary search* technique. Exercises 7.8 and 7.9 at the end of this chapter ask you to implement recursive versions of the linear and binary search.

7.8.1 Searching an Array with Linear Search

In the program in Fig. 7.11, method **LinearSearch** (defined on lines 44–54) uses a **for** structure containing an **if** structure to compare each element of an array with a *search key* (line 44). If the search key is found, the method returns the subscript value for the element to indicate the exact position of the search key in the array. If the search key is not found, the method returns **-1**. (The value **-1** is a good choice because it is not a valid subscript number.) If the elements of the array being searched are not in any particular order, it is just as likely that the value will be found in the first element as in the last. On average, the program will have to compare the search key with half the elements of the array. The program contains a 100-element array filled with the even integers from 0–198. The user types the search key in a **TextBox** (called *inputTextBox*) and clicks the **findButton** to start the search. [*Note*: The array is passed to **LinearSearch** even though the array is an instance variable of the class. This is done because an array normally is passed to a method of another class for searching.]

```
// Fig. 7.11: LinearSearcher.cs
 2
    // Demonstrating linear searching of an array.
 3 using System;
 4 using System.Drawing;
 5 using System.Collections;
    using System.ComponentModel;
 7
    using System.Windows.Forms;
 8
    using System.Data;
 9
10
    public class LinearSearcher: System.Windows.Forms.Form
11
12
       private System.Windows.Forms.Button searchButton;
13
       private System.Windows.Forms.TextBox inputTextBox;
14
       private System.Windows.Forms.Label outputLabel;
15
       int[] a = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26,
16
17
                   28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50 };
18
19
       // Visual Studio .NET generated code
20
21
       [STAThread]
22
       static void Main()
23
```

Fig. 7.11 Linear search of an array. (Part 1 of 2.)

```
24
           Application.Run( new LinearSearcher() );
25
       }
26
27
       private void searchButton Click( object sender,
28
           System. EventArgs e )
29
       {
30
           int searchKey = Int32.Parse( inputTextBox.Text );
31
32
           int elementIndex = LinearSearch( a, searchKey );
33
34
           if ( elementIndex != -1 )
35
              outputLabel.Text =
36
                  "Found value in element " + elementIndex;
37
38
           else
39
              outputLabel.Text = "Value not found";
40
41
        } // end method searchButton Click
42
43
       // search array for the specified key value
44
       public int LinearSearch( int[] array, int key )
45
46
           for ( int n = 0; n < array.Length; n++ )</pre>
47
48
              if ( array[ n ] == key )
49
                 return n;
50
51
52
           return -1;
53
54
        } // end method LinearSearch
    }
55
              LinearSearch
                             LinearSearch
                                                          Search
                                            Search
                                          Value not found
             Found value in element 2
```

Fig. 7.11 Linear search of an array. (Part 2 of 2.)

7.8.2 Searching a Sorted Array with Binary Search

The linear search method works well for small or unsorted arrays. However, for large arrays, linear searching is inefficient. If the array is sorted, the high-speed *binary search* technique can be used. The binary search algorithm eliminates half of the elements in the array being searched after each comparison. The algorithm locates the middle array element and compares it with the search key. If they are equal, the search key has been found, and the subscript of that element is returned. Otherwise, the problem is reduced to searching half of the array. If the search key is less than the middle array element, the first half of the array is searched; otherwise, the second half of the array is searched. If the search key is not the middle element in the specified subarray (a piece of the original array), the algorithm is re-

peated in one quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).

In a worst-case scenario, searching an array of 1024 elements will take only 10 comparisons by using a binary search. Repeatedly dividing 1024 by 2 (after each comparison we eliminate from consideration half the array) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 (2¹⁰) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 (2²⁰) elements takes a maximum of 20 comparisons to find the key. An array of one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous increase in performance over the linear search, which required comparing the search key with an average of half the elements in the array. For a one-billion-element array, the difference is between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

Figure 7.12 presents the iterative version of method **BinarySearch** (lines 59–85). The method receives two arguments—an integer array called **array** (the array to search) and an integer **key** (the search key). The array is passed to **BinarySearch** even though the array is an instance variable of the class. Once again, this is done because an array normally is passed to a method of another class for searching. Line 67 calculates the middle element of the array being searched by determining the number of elements in the array and dividing this value by 2. Recall that using the / operator with integers performs an integer division, which truncates the result. So, when there is an even number of elements in the array there is no "middle" element—the middle of our array is actually between two elements. When this occurs, the calculation on line 67 returns the smaller index of the two middle elements.

```
// Fig. 7.12: BinarySearchTest.cs
 2
    // Demonstrating a binary search of an array.
 3
 4
   using System;
    using System.Drawing;
    using System.Collections;
 7
    using System.ComponentModel;
 8
    using System.Windows.Forms;
 9
    using System.Data;
10
11
    public class BinarySearchTest: System.Windows.Forms.Form
12
13
       private System.Windows.Forms.Label promptLabel;
14
15
       private System.Windows.Forms.TextBox inputTextBox;
16
17
       private System.Windows.Forms.Label resultLabel;
18
       private System.Windows.Forms.Label displayLabel;
19
       private System.Windows.Forms.Label outputLabel;
20
21
       private System.Windows.Forms.Button findButton;
22
```

Fig. 7.12 Binary search of a sorted array. (Part 1 of 4.)

```
23
       private System.ComponentModel.Container components = null;
24
25
       int[] a = { 0, 2, 4, 6, 8, 10, 12, 14, 16,
26
                   18, 20, 22, 24, 26, 28 };
27
28
       // Visual Studio .NET generated code
29
30
       // main entry point for application
31
       [STAThread]
32
       static void Main()
33
34
          Application.Run( new BinarySearchTest() );
35
       }
36
37
       // searches for an element by calling
38
       // BinarySearch and displaying results
39
       private void findButton_Click( object sender,
40
          System. EventArgs e )
41
       {
42
          int searchKey = Int32.Parse( inputTextBox.Text );
43
44
          // initialize display string for the new search
45
          outputLabel.Text = "Portions of array searched\n";
46
47
          // perform the binary search
48
          int element = BinarySearch( a, searchKey );
49
50
          if ( element != -1 )
51
             displayLabel.Text = "Found value in element " +
52
                element;
53
          else
54
             displayLabel.Text = "Value not found";
55
56
       } // end findButton Click
57
58
       // searchs array for specified key
59
       public int BinarySearch( int[] array, int key )
60
61
          int low = 0;
                                         // low subscript
62
          int high = array.Length - 1; // high subscript
63
          int middle:
                                         // middle subscript
64
65
          while ( low <= high )</pre>
66
67
             middle = ( low + high ) / 2;
68
69
             // the following line displays the portion
70
             // of the array currently being manipulated during
71
             // each iteration of the binary search loop
72
             BuildOutput( a, low, middle, high );
73
74
             if ( key == array[ middle ] ) // match
75
                return middle;
```

Fig. 7.12 Binary search of a sorted array. (Part 2 of 4.)

```
76
                else if ( key < array[ middle ] )</pre>
77
                   high = middle - 1; // search low end of array
78
               else
79
                   low = middle + 1;
80
81
            } // end BinarySearch
82
83
            return -1; // search key not found
84
85
        } // end method BinarySearch
86
87
        public void BuildOutput(
88
            int[] array, int low, int mid, int high )
89
90
            for ( int i = 0; i < array.Length; i++ )</pre>
91
92
               if ( i < low || i > high )
93
                   outputLabel.Text += "
94
95
                // mark middle element in output
96
               else if ( i == mid )
97
                   outputLabel.Text +=
98
                       array[ i ].ToString( "00" ) + "* ";
99
               else
100
                   outputLabel.Text +=
101
                       array[ i ].ToString( "00" ) + " ";
102
            }
103
104
            outputLabel.Text += "\n";
105
106
        } // end BuildOutput
107
108 } // end class BinarySearchTest
                RinarySearch
                                                             _ D X
                    Enter key 6
                                         Result: Found value in element 3
                                       Find Key
                  Portions of array searched
                    02 04 06 08 10 12
02 04 06 08 10 12
                                        14* 16 18 20 22 24 26 28
                E BinarySearch
                                                             _ D X
                    Enter key 8
                                         Result: Found value in element 4
                                       Find Key
                  Portions of array searched
                  00 02 04 06 08 10 12
00 02 04 06 08 10 12
                                       14* 16 18 20 22 24 26 28
                           08 10* 12
                           08=
```

Fig. 7.12 Binary search of a sorted array. (Part 3 of 4.)

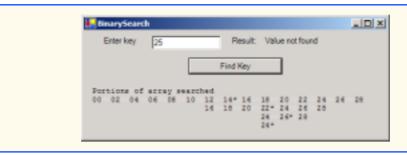


Fig. 7.12 Binary search of a sorted array. (Part 4 of 4.)

If key matches the middle element of a subarray (line 74), BinarySearch returns middle (the subscript of the current element), indicating that the value was found and the search is complete. If key does not match the middle element of a subarray, Binary-Search adjusts the low subscript or high subscript (both declared in the method) so that a smaller subarray can be searched. If key is less than the middle element (line 76), the high subscript is set to middle - 1, and the search continues on the elements from low to middle - 1. If key is greater than the middle element (line 78), the low subscript is set to middle + 1, and the search continues on the elements from middle + 1 to high. These comparisons occur in the nested if/else structure on lines 74–79.

The program uses a 15-element array. The first power of 2 greater than the number of array elements is 16 (2⁴)—so at most four comparisons are required to find the **key**. To illustrate this concept, method **BinarySearch** calls method **BuildOutput** (lines 87–106) to output each subarray during the binary search process. **BuildOutput** marks the middle element in each subarray with an asterisk (*) to indicate the element with which the **key** is compared. Each search in this example results in a maximum of four lines of output—one per comparison. Note that the .NET framework includes a built-in array-searching capability that implements the binary-search algorithm. To search for the key 7 in the sorted array **a** in Fig. 7.12, you can use the statement

Array.BinarySearch(a, 7);

7.9 Multiple-Subscripted Arrays

So far we have studied *single-subscripted* (or *one-dimensional*) arrays—i.e., those that contain single lists of values. In this section, we introduce *multiple-subscripted* (often called *multidimensional*) arrays. Such arrays require two or more subscripts to identify particular elements. Arrays that require two subscripts to identify a particular element commonly are called *double-subscripted arrays*. We concentrate on *double-subscripted arrays* (often called *two-dimensional arrays*). There are two types of multiple-subscripted arrays—*rectangular* and *jagged*. Rectangular arrays with two subscripts often represent *tables* of values consisting of information arranged in *rows* and *columns*, where each row is the same size, and each column is the same size. To identify a particular table element, we must specify the two subscripts—by convention, the first identifies the element's row and the second identifies the element's column. Multiple-subscripted arrays can have more than two subscripts. Figure 7.13 illustrates a double-subscripted array, a, containing three rows and four columns (i.e., a 3-by-4 array). An array with *m* rows and *n* columns is called an *m-by-n array*.

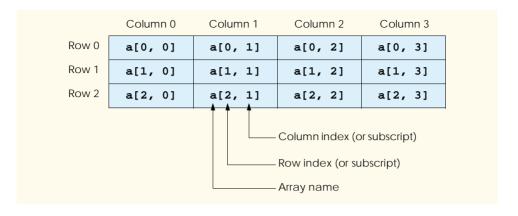


Fig. 7.13 Double-subscripted array with three rows and four columns.

Every element in array \mathbf{a} is identified in Fig. 7.13 by an element name of the form $\mathbf{a}[\mathbf{i},\mathbf{j}]$, in which \mathbf{a} is the name of the array, and \mathbf{i} and \mathbf{j} are the subscripts that uniquely identify the row and column of each element in \mathbf{a} . Notice that the names of the elements in the first row all have a first subscript of $\mathbf{0}$; the names of the elements in the fourth column all have a second subscript of $\mathbf{3}$.

Multiple-subscripted arrays can be initialized in declarations like single-subscripted arrays. A double-subscripted array **b** with two rows and two columns could be declared and initialized with

```
int[,] b = new int[ 2, 2 ];
b[ 0, 0 ] = 1;
b[ 0, 1 ] = 2;
b[ 1, 0 ] = 3;
b[ 1, 1 ] = 4;
```

or this can be written on one line using an *initializer list* as shown below:

```
int[,] b = { { 1, 2 }, { 3, 4 } };
```

The values are grouped by row in braces. Thus, 1 and 2 initialize b[0,0] and b[0,1], and 3 and 4 initialize b[1,0] and b[1,1]. The compiler determines the number of rows by counting the number of sub-initializer lists (represented by sets of braces) in the main initializer list. The compiler determines the number of columns in each row by counting the number of initializer values in the sub-initializer list for that row. Method GetLength returns the length of a particular array dimension. In the preceding example, b.GetLength(0) returns the length of the zeroth dimension of b, which is 2.

Jagged arrays are maintained as arrays of arrays. Unlike in rectangular arrays, the arrays that compose jagged arrays can be of different lengths. The declaration

```
int[][] c = new int[ 2 ][]; // allocate rows
// allocate and initialize elements in row 0
c[ 0 ] = new int[] { 1, 2 };
```

```
// allocate and initialize elements in row 0
c[ 1 ] = new int[] { 3, 4, 5 };
```

creates integer array **c** with row **0** (which is an array itself) containing two elements (**1** and **2**), and row **1** containing three elements (**3**, **4** and **5**). The **Length** property of each subarray can be used to determine the size of each column. For the jagged array **c**, the size of the **zeroth** column is **c**[**0**]. **Length**, which is **2**.

The application in Fig. 7.14 demonstrates the initialization of double-subscripted arrays in declarations and the use of nested **for** loops to traverse the arrays (i.e., to manipulate each array element).

```
// Fig. 7.14: TwoDimensionalArrays.cs
    // Initializing two-dimensional arrays.
 3 using System;
 4 using System.Drawing;
 5 using System.Collections;
 6 using System.ComponentModel;
    using System.Windows.Forms;
 8
    using System.Data;
 9
10
   public class TwoDimensionalArrays : System.Windows.Forms.Form
11
12
       private System.Windows.Forms.Button showOutputButton;
13
       private System.Windows.Forms.Label outputLabel;
14
15
       // Visual Studio .NET generated code
16
17
       [STAThread]
18
       static void Main()
19
20
          Application.Run( new TwoDimensionalArrays() );
21
       }
22
23
       private void showOutputButton_Click( object sender,
24
          System.EventArgs e )
25
       {
26
          // declaration and initialization of rectangular array
27
          int[,] array1 = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };
28
29
          // declaration and initialization of jagged array
30
          int[][] array2 = new int[ 3 ][];
31
          array2[ 0 ] = new int[] { 1, 2 };
32
          array2[ 1 ] = new int[] { 3 };
33
          array2[ 2 ] = new int[] { 4, 5, 6 };
34
35
          outputLabel.Text = "Values in array1 by row are\n";
36
37
          // output values in array1
38
          for ( int i = 0; i < array1.GetLength( 0 ); i++ )</pre>
39
40
             for ( int j = 0; j < array1.GetLength( 1 ); j++ )</pre>
41
                outputLabel.Text += array1[ i, j ] + " ";
```

Fig. 7.14 Initializing multidimensional arrays. (Part 1 of 2.)

```
42
43
               outputLabel.Text += "\n";
44
           }
45
46
           outputLabel.Text += "\nValues in array2 by row are\n";
47
48
           // output values in array2
49
           for ( int i = 0; i < array2.Length; i++ )</pre>
50
51
               for ( int j = 0; j < array2[ i ].Length; j++ )</pre>
52
                  outputLabel.Text += array2[ i ][ j ] + "
53
54
               outputLabel.Text += "\n";
55
           }
56
57
        } // end method showOutputButton Click
58
    }
                       TwoDimensionalArrays
                                                   Show Output
                        Values in array 1 by row are
                        123
                        Values in array2 by row are
                        456
```

Fig. 7.14 Initializing multidimensional arrays. (Part 2 of 2.)

The declaration of **array1** (line 27) provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3. The second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of **array2** (line 30) creates a jagged array of 3 arrays (specified by the 3 in the first set of square brackets). Lines 31–33 initialize each subarray so that the first subarray contains the values 1 and 2, the second contains the value 3 and the last contains the values 4, 5 and 6.

The for structure on lines 38–44 appends the elements of array1 to string output. Note the use of a nested for structure to output the rows of each double-subscripted array. In the nested for structures for array1, we use method GetLength to determine the number of elements in each dimension of the array. Line 38 determines the number of rows in the array by invoking array1.GetLength(0), and line 40 determines the number of columns in the array by invoking array1.GetLength(1). Arrays with additional dimensions would require more deeply nested for loops to process.

The nested **for** structures on lines 49–55 output the elements of jagged array **array2**. Recall that a jagged array is essentially an array that contains additional arrays as its elements. Line 49 uses the **Length** property of **array2** to determine the number of rows in the jagged array. Line 51 determines the **Length** of each subarray with the expression **array2**[i].**Length**.

Many common array manipulations use **for** repetition structures. For the remainder of this section, we will focus on manipulations of jagged arrays. Imagine a jagged array **a**,

which contains 3 rows, or arrays. The following **for** structure sets all the elements in the third row of array **a** to zero:

```
for ( int col = 0; col < a[ 2 ].Length; col++ )
   a[ 2 ][ col ] = 0;</pre>
```

We specified the *third* row; therefore, we know that the first subscript is always 2 (0 is the first row and 1 is the second row). The **for** loop varies only the second subscript (i.e., the column subscript). Notice the use of a[2]. Length in the **for** structure's conditional expression. This statement demonstrates that each row of a is an array in itself, and therefore the program can access a typical array's properties, such as **Length**. Assuming the length of array a[2] is 4, the preceding **for** structure is equivalent to the assignment statements

```
a[2][0]=0;
a[2][1]=0;
a[2][2]=0;
a[2][3]=0;
```

The following nested **for** structure determines the total of all the elements in array **a**. We use **a.Length** in the conditional expression of the outer **for** structure to determine the number of rows in **a**, in this case, 3.

```
int total = 0;
for ( int row = 0; row < a.Length; row++ )
  for ( int col = 0; col < a[ row ].Length; col++ )
    total += a[ row ][ col ];</pre>
```

The **for** structure totals the elements of the array one row at a time. The outer **for** structure begins by setting the **row** subscript to **0**, so the elements of the first row may be totaled by the inner **for** structure. Then the outer **for** structure increments **row** to **1**, so the second row can be totaled. Finally, the outer **for** structure increments **row** to **2**, so the third row can be totaled. The result can be displayed when the nested **for** structure terminates.

The program in Fig. 7.15 performs several other array manipulations on 3-by-4 array **grades**. Each row of the array represents a student, and each column represents a grade on one of the four exams that the student took during the semester. The array manipulations are performed by four methods. Method **Minimum** (lines 64–76) determines the lowest grade of any student for the semester. Method **Maximum** (lines 79–91) determines the highest grade of any student for the semester. Method **Average** (lines 94–102) determines a particular student's semester average.

Methods Minimum and Maximum use array grades and the variables students (number of rows in the array) and exams (number of columns in the array). Each method loops through array grades by using nested for structures. Consider the nested for structure from method Minimum (lines 68–73). The outer for structure sets i (i.e., the row subscript) to 0 so the elements of the first row can be compared with variable low-Grade in the body of the inner for structure. The inner for structure loops through the four grades of a particular row and compares each grade with lowGrade. If a grade is less than lowGrade, then lowGrade is set to that grade. The outer for structure then increments the row subscript by 1. The elements of the second row are compared with variable lowGrade. The outer for structure then increments the row subscript to 2. The elements

```
// Fig. 7.15: DoubleArray.cs
 2
    // Manipulating a double-subscripted array.
 3
   using System:
 4 using System.Drawing;
   using System.Collections;
 5
   using System.ComponentModel;
 7
    using System.Windows.Forms;
 8
    using System.Data;
 9
10
   public class DoubleArray : System.Windows.Forms.Form
11
12
       private System.Windows.Forms.Button showOutputButton;
13
       private System.Windows.Forms.Label outputLabel;
14
15
       int[][] grades;
16
       int students, exams;
17
18
       // Visual Studio .NET generated code
19
20
       [STAThread]
21
       static void Main()
22
       {
23
          Application.Run( new DoubleArray() );
24
       }
25
26
       private void showOutputButton_Click( object sender,
27
          System.EventArgs e )
28
29
       {
30
          grades = new int[ 3 ][];
31
          grades[ 0 ] = new int[]{ 77, 68, 86, 73 };
          grades[ 1 ] = new int[]{ 96, 87, 89, 81
32
33
          grades[ 2 ] = new int[]{ 70, 90, 86, 81 };
34
35
          students = grades.Length;
                                          // number of students
36
          exams = grades[ 0 ].Length;
                                         // number of exams
37
38
          // line up column headings
39
          outputLabel.Text = "
                                                   " ;
40
41
          // output the column headings
42
          for ( int i = 0; i < exams; i++ )</pre>
43
             outputLabel.Text += "[" + i + "] ";
44
45
          // output the rows
46
          for ( int i = 0; i < students; i++ )</pre>
47
          {
48
             outputLabel.Text += "\ngrades[" + i + "] ";
49
50
             for ( int j = 0; j < exams; j++ )</pre>
51
                outputLabel.Text += grades[ i ][ j ] + " ";
52
          }
```

Fig. 7.15 Example using double-subscripted arrays. (Part 1 of 3.)

```
53
54
           outputLabel.Text += "\n\nLowest grade: " + Minimum() +
55
              "\nHighest grade: " + Maximum() + "\n";
56
57
           for ( int i = 0; i < students; i++ )</pre>
58
             outputLabel.Text += "\nAverage for student " + i + " is " +
59
                 Average( grades[ i ] );
60
61
        } // end method showOutputButton Click
62
63
       // find minimum grade in grades array
64
       public int Minimum()
65
66
           int lowGrade = 100;
67
68
           for ( int i = 0; i < students; i++ )</pre>
69
70
              for ( int j = 0; j < exams; j++ )</pre>
71
72
                 if ( grades[ i ][ j ] < lowGrade )</pre>
73
                     lowGrade = grades[ i ][ j ];
74
75
           return lowGrade;
76
       }
77
78
       // find maximum grade in grades array
79
       public int Maximum()
80
81
           int highGrade = 0;
82
83
           for ( int i = 0; i < students; i++ )
84
85
              for ( int j = 0; j < exams; j++ )</pre>
86
87
                 if ( grades[ i ][ j ] > highGrade )
88
                    highGrade = grades[ i ][ j ];
89
90
          return highGrade;
91
       }
92
93
       // determine average grade for a particular student
94
       public double Average( int[] setOfGrades )
95
96
           int total = 0;
97
98
           for ( int i = 0; i < setOfGrades.Length; i++ )</pre>
99
              total += setOfGrades[ i ];
100
101
           return ( double ) total / setOfGrades.Length;
       }
102
103
104 } // end class DoubleArray
```

Fig. 7.15 Example using double-subscripted arrays. (Part 2 of 3.)

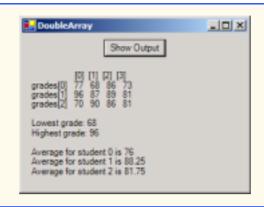


Fig. 7.15 Example using double-subscripted arrays. (Part 3 of 3.)

of the third row are compared with variable **lowGrade**. When execution of the nested structure is complete, **lowGrade** contains the smallest grade in the double-subscripted array. Method **Maximum** works similarly to method **Minimum**.

Method Average takes one argument—a single-subscripted array of test results for a particular student. When Average is called (line 59), the argument grades[i] specifies that a particular row of the double-subscripted array grades is to be passed to Average. For example, the argument grades[1] represents the four values (a single-subscripted array of grades) stored in the second row of the double-subscripted array grades. Remember that a jagged two-dimensional array is an array with elements that are single-subscripted arrays. Method Average calculates the sum of the array elements, divides the total by the number of test results and then returns the floating-point result cast as a double value (line 101).

7.10 foreach Repetition Structure

C# provides the **foreach** retition structure for iterating through values in data structures, such as arrays. When used with one-dimensional arrays, **foreach** behaves like a **for** structure that iterates through the range of indices from **0** to the array's **Length**. Instead of a counter, **foreach** uses a variable to represent the value of each element. The program in Fig. 7.16 uses the **foreach** structure to determine the minimum value in a two-dimensional array of grades.

```
// Fig. 7.16: ForEach.cs
// Demonstrating for/each structure.
using System;

class ForEach
// main entry point for the application
static void Main( string[] args )
{
```

Fig. 7.16 Using For Each/Next with an array. (Part 1 of 2.)

```
int[,] gradeArray = { { 77, 68, 86, 73 },
     { 98, 87, 89, 81 }, { 70, 90, 86, 81 } };
10
11
12
13
            int lowGrade = 100;
14
15
            foreach ( int grade in gradeArray )
16
            {
17
                if ( grade < lowGrade )</pre>
18
                    lowGrade = grade;
19
20
21
            Console.WriteLine( "The minimum grade is: " + lowGrade );
22
        }
23
     }
The minimum grade is: 68
```

Fig. 7.16 Using For Each/Next with an array. (Part 2 of 2.)

The header of the **foreach** structure (line 15) specifies a variable, **grade**, and an array, **gradeArray**. The **foreach** structure iterates through all elements in **grade-Array**, sequentially assigning each value to variable **grade**. Line 15 compares each value to variable **lowGrade**, which stores the lowest grade in the array.

For rectangular arrays, the repetition of the **foreach** structure begins with the element whose indices are all zero, then iterates through all possible combinations of indices, incrementing the rightmost index first. When the rightmost index reaches its upper bound, it is reset to zero, and the index to the left of it is incremented by 1. In this case, **grade** takes the values as they are ordered in the initializer list in lines 10–11. When all the grades have been processed, **lowGrade** is displayed (line 21).

Although many array calculations are handled best with a counter, **foreach** is useful when the indices of the elements are not important. The **foreach** structure is particularly useful for looping through arrays of objects, as we discuss in Chapter 10, Object-Oriented Programming: Polymorphism.

SUMMARY

- An array is a group of contiguous memory locations that all have the same name and type.
- To refer to a particular location or element in an array, specify the name of the array and the position number of the element within the array.
- The first element in every array is the zeroth element (i.e., element 0).
- The position number in square brackets is more formally called a subscript (or an index). This number must be an integer or an integer expression.
- To reference the i^{th} element of a single-dimensional array, use i-1 as the index.
- The brackets that enclose the subscript of an array are operators that have the same level of precedence as parentheses.
- When arrays are allocated, the elements are initialized to zero for the numeric primitive-data-type variables, to false for bool variables or to null for reference types.
- Arrays may be declared to contain most data types.

• In an array of primitive data types, every element of the array contains one value of the declared data type of the array.

- In an array of a reference type, every element of the array is a reference to an object of the data type of the array. For example, every element of a **string** array is a reference to a **string** and that reference has the value **null** by default.
- The elements of single-dimensional and rectangular arrays can be allocated and initialized in the array declaration by following the declaration with an equal sign and a comma-separated initializer list enclosed in braces ({ and }).
- A const variable must be declared and initialized in the same statement.
- Constants also are called named constants. They often are used to make a program more readable.
- Unlike its predecessors C and C++, .NET-compliant languages provide mechanisms to prevent accessing elements outside the bounds of the array.
- When a reference is made to a nonexistent element of an array, an IndexOutOfRangeException occurs.
- To pass an array argument to a method, specify the name of the array without any brackets.
- Although entire arrays are passed by reference, individual array elements of primitive data types are passed by value, as are simple variables.
- To pass an array element to a method, use the subscripted name of the array element as an argument in the method call.
- Sorting data (i.e., placing the data into a particular order, such as ascending or descending) is one of the most important computing applications.
- The chief virtue of the bubble sort is that it is easy to program. However, the bubble sort runs slowly, which becomes apparent when sorting large arrays.
- The linear search method works well for small or unsorted arrays. However, for large arrays, linear searching is inefficient.
- After each comparison, the binary search algorithm eliminates from consideration half the elements in the array being searched. The algorithm locates the middle array element and compares it to the search key. If they are equal, the search key has been found, and the subscript of that element is returned. Otherwise, the problem is reduced to searching half the array. If the search key is less than the middle array element, the first half of the array is searched; otherwise, the second half of the array is searched. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).
- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 that is greater than the number of elements in the array.
- There are two types of multiple-subscripted arrays—rectangular and jagged.
- In general, an array with m rows and n columns is referred to as an m-by-n array.
- Multiple-subscripted arrays can be initialized in declarations, as can single-subscripted arrays.
- The compiler determines the number of columns in each row by counting the number of initializer values in the sub-initializer list for that row.
- Jagged arrays are maintained as arrays of arrays. Unlike rectangular arrays, rows in jagged arrays
 can be of different lengths.
- Many common array manipulations use **for** repetition structures.
- When used with one-dimensional arrays, **foreach** behaves like a **for** structure that iterates through the range of indices from **0** to the array's **Length**.

Chapter 7 275 Arravs

• For rectangular arrays, the repetition of the foreach structure begins with the element whose indices are all zero, then iterates through all possible combinations of indices, incrementing the rightmost index first. When the rightmost index reaches its upper bound, it is reset to zero, and the index to the left of it is incremented by 1.

TERMINOLOGY

[], subscript operator array allocated with new array automatically initialized to zeros array bounds array declaration null array of arrays (jagged array) bar chart binary search algorithm brute force bubble sort column const constant variable declare an array dice-rolling program double-subscripted array element exception

foreach structure graph information histogram

ignoring element zero

initializer list

initializing double-subscripted arrays

in declarations

innermost set of square brackets

invalid array reference

jagged array key value length of an array linear search lvalue ("left value") *m*-by-*n* array

multiple-subscripted array

named constant nested for loop new operator

"off-by-one error" one-dimensional array

partition partitioning step pass of a bubble sort passing array to method

passing array element to method

position number read-only variable rectangular array search key searching

single-subscripted array

sinking sort size of an array sorting

square brackets, []

subarray

sub-initializer list

subscript swap table

table element TextBox

"walk" past end of an array

zero-based counting zeroth element

CELE DEVIEW EVEDOUCES

SELF-	KE	VIEW EXERCISES
7.1		l in the blanks in each of the following statements:
	a)	Lists and tables of values can be stored in
	b)	The elements of an array are related by the fact that they have the same and
	c)	The number that refers to a particular element of an array is called its
	d)	The process of placing the elements of an array in order is called the array.
	e)	Determining if an array contains a certain key value is called the array.
	f)	Arrays that use two or more subscripts are referred to as arrays

g)	arrays are maintained as arrays of arrays.		
h)	Avariable must be declared and initialized in the same statement, or a syntax		
	error will occur.		
i)	C# provides the repetition structure for iterating through values in data struc-		
	tures, such as arrays.		
j)	When an invalid array reference is made, an is generated.		

- 7.2 State whether each of the following is *true* or *false*. If *false*, explain why.
 - a) An array can store many different types of values at the same time.
 - b) An array subscript normally should be of data type **float**.
 - c) An individual array element that is passed to a method and modified in that method will contain the modified value when the called method completes execution.
 - d) The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.
 - e) There are two types of multiple-subscripted arrays—square and jagged.
 - f) A const variable must be declared and initialized in the same statement, or a syntax error will occur.
 - g) After each comparison, the binary search algorithm eliminates from consideration one third of the elements in the portion of the array that is being searched.
 - h) To determine the number of elements in an array, we can use the NumberOfElements property.
 - i) The linear search method works well for small or unsorted arrays.
 - j) In an *m*-by-*n* array, the *m* stands for the number of columns and the *n* stands for the number of rows.

ANSWERS TO SELF-REVIEW EXERCISES

- 7.1 a) arrays. b) name, type. c) subscript, index or position number. d) sorting. e) searching. f) multiple-subscripted. g.) Jagged. h) const. i) foreach. j) IndexOutofRangeException.
- 7.2 a) False. An array can store only values of the same type. b) False. An array subscript must be an integer or an integer expression. c) False. For individual primitive-data-type elements of an array, they are passed by value. If a reference to an array element is passed, then modifications to that array element are reflected in the original. An individual element of a reference type is passed to a method by reference. d) True. e) False. The two different types are called rectangular and jagged. f) True. g) False. After each comparison, the binary search algorithm eliminates from consideration half the elements in the portion of the array that is being searched. h) False. To determine the number of elements in an array, we can use the **Length** property. i) True. j) False. In an *m*-by-*n* array, the *m* stands for the number of rows and the *n* stands for the number of columns.

EXERCISES

- 7.3 Write statements to accomplish each of the following tasks:
 - a) Display the value of the seventh element of character array £.
 - b) Initialize each of the five elements of single-subscripted integer array g to 8.
 - c) Total the elements of floating-point array **c** of 100 elements.
 - d) Copy 11-element array **a** into the first portion of array **b** containing 34 elements.
 - e) Determine the smallest and largest values contained in 99-element floating-point array w.
- 7.4 Use a single-subscripted array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive \$200 per week, plus 9% of their gross sales for that week. For example, a salesperson who grosses \$5000 in sales in a week receives \$200 plus 9% of \$5000, or a total of \$650. Write a program (using an array of counters) that determines how many

of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

- a) \$200-\$299
- b) \$300-\$399
- c) \$400-\$499
- d) \$500-\$599
- e) \$600-\$699
- f) \$700-\$799
- g) \$800-\$899
- h) \$900-\$999
- i) \$1000 and over
- 7.5 Use a single-subscripted array to solve the following problem: Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it is not a duplicate of a number already read. Provide for the "worst case" (in which all 20 numbers are different). Use the smallest possible array to solve this problem.
- **7.6** (*Turtle Graphics*) The Logo language made famous the concept of *turtle graphics*. Imagine a mechanical turtle that walks around the room under the control of a program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about without writing anything. In this problem, you will simulate the operation of the turtle and create a computerized sketchpad.

Use a 20-by-20 array **floor**, which is initialized to zeros. Read commands from an array that contains them. At all times, keep track of the current position of the turtle and whether the pen is up or down. Assume that the turtle always starts at position 0,0 of the floor with its pen up. The set of turtle commands your program must process are as follows:

Command	Meaning
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5,10	Move forward 10 spaces (or a number other than 10)
6	Print the 20-by-20 array
9	End of data (sentinel)

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square, leaving the pen in the up position:

2 5,12 3 5,12 3 5,12 3 5,12 1 6

9

As the turtle moves with the pen down, set the appropriate elements of array **floor** to **1s**. When the **6** command (print) is given, wherever there is a **1** in the array, display an asterisk or another character. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities we have discussed. Write several turtle graphics programs to draw interesting shapes. Add commands to increase the power of your turtle graphics language.

SPECIAL SECTION: RECURSION EXERCISES

- 7.7 (*Palindromes*) A palindrome is a string that is spelled the same forward and backward. Some examples of palindromes are "radar," "able was i ere i saw elba" and, if blanks are ignored, "a man a plan a canal panama." Write a recursive method **testPalindrome** that returns **true** if the string stored in the array is a palindrome and **false** otherwise. The method should ignore spaces and punctuation in the string.
- 7.8 (*Linear Search*) Modify Fig. 7.11 to use recursive method **LinearSearch** to perform a linear search of the array. The method should receive an integer array and the size of the array as arguments. If the search key is found, return the array subscript; otherwise, return –1.
- 7.9 (*Binary Search*) Modify the program in Fig. 7.12 to use a recursive method **Binary-Search** to perform the binary search of the array. The method should receive an integer array and the starting and ending subscript as arguments. If the search key is found, return the array subscript; otherwise, return -1.
- **7.10** (*Quicksort*) In this chapter, we discussed the sorting technique bubble sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:
 - a) *Partitioning Step.* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.
 - b) Recursive Step. Perform step 1 on each unsorted subarray.

Each time Step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, it must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple, but how do we determine the final position of the first element of each subarray? Consider the following set of values (partitioning element in bold—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost element of the array, compare each element to 37 until an element less than 37 is found, then swap 37 and that element. The first element less than 37 is 12, so 37 and 12 are swapped. The new array is

12 2 6 4 89 8 10 **37** 68 45

Element 12 is italicized to indicate that it was just swapped with 37.

b) Starting from the left of the array, but beginning with the element after 12, compare each element to 37 until an element greater than 37 is found, then swap 37 and that element. The first element greater than 37 is 89, so 37 and 89 are swapped. The new array is

12 2 6 4 **37** 8 10 89 68 45

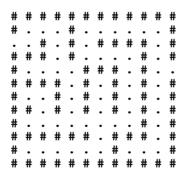
c) Starting from the right, but beginning with the element before 89, compare each element to 37 until an element less than 37 is found, then swap 37 and that element. The first element less than 37 is 10, so 37 and 10 are swapped. The new array is

d) Starting from the left, but beginning with the element after 10, compare each element to 37 until an element greater than 37 is found, then swap 37 and that element. There are no more elements greater than 37, so when we compare 37 to itself, we know that 37 has been placed in its final location of the sorted array.

Once the partition has been applied to the previous array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Using the preceding discussion, write recursive method **QuickSort** to sort a single-subscripted integer array. The method should receive as arguments an integer array, a starting subscript and an ending subscript. Method **Partition** should be called by **QuickSort** to perform the partitioning step.

7.11 (*Maze Traversal*) The following grid of #s and dots (.) is a double-subscripted array representation of a maze:



The #s represent the walls of the maze, and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

Write recursive method **MazeTraverse** to walk through the maze. The method should receive as arguments a 12-by-12 character array representing the maze and the starting location of the maze. As **MazeTraverse** attempts to locate the exit from the maze, it should place the character **X** in each square in the path. The method should display the maze after each move so the user can watch as the maze is solved.