4.10 Multidimensional Arrays

Multidimensional arrays with two dimensions are often used to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two indices. By convention, the first identifies the element's row and the second its column. Arrays that require two indices to identify a particular element are called two-dimensional arrays. C# supports two types of two-dimensional arrays—rectangular arrays and jagged arrays.

Rectangular Arrays

Rectangular arrays are used to represent tables of information in the form of rows and columns, where each row has the same number of columns. Figure 4.17 illustrates a rectangular array named a containing three rows and four columns—a three-by-four array.

Every element in array a is identified in Fig. 4.17 by an array-access expression of the form a[row, column]; a is the name of the array, and row and column are the indices that uniquely identify each element in array a by row and column number. The names of the elements in row 0 all have a first index of 0, and the names of the elements in column 3 all have a second index of 3. Like one-dimensional arrays, multidimensional arrays can be initialized with array initializers in declarations. A rectangular array b with two rows and two columns could be declared and initialized with nested array initializers as follows:

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

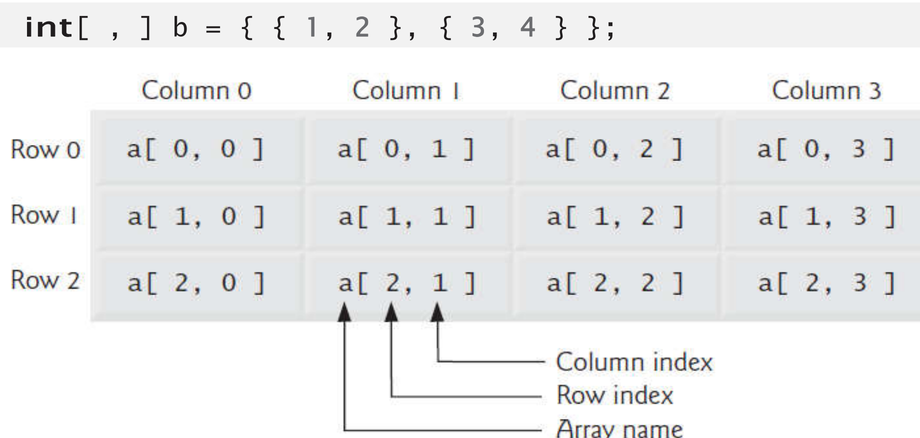

Fig. 4.17 | Rectangular array with three rows and four columns.

The initializer values are grouped by row in braces. So 1 and 2 initialize b[0, 0] and b[0, 1], respectively, and 3 and 4 initialize b[1, 0] and b[1, 1], respectively. The compiler counts the number of nested array initializers (represented by sets of two inner braces with- in the outer braces) in the initializer list to determine the number of rows in array b. The compiler counts the initializer values in the nested array initializer for a row to determine the number of columns (two) in that row. The compiler will generate an error if the num- ber of initializers in each row is not the same, because every row of a rectangular array must have the same length.

Jagged Arrays

A jagged array is maintained as a one-dimensional array in which each element refers to a one-dimensional array. The manner in which jagged arrays are represented makes them quite flexible, because the lengths of the rows in the array need not be the same. For example, jagged arrays could be used to store a single student's exam grades across multiple classes, where the number of exams may vary from class to class.

We can access the elements in a jagged array by an array-access expression of the form arrayName[row][column]—similar to the array-access expression for rectangular arrays, but with a separate set of square brackets for each dimension. A jagged array with three rows of different lengths could be declared and initialized as follows:

```
int[][] jagged = { new int[] { 1, 2 },
                   new int[] { 3 },
                   new int[] { 4, 5, 6 } };
```

In this statement, 1 and 2 initialize jagged[0][0] and jagged[0][1], respectively; 3 initializes jagged[1][0]; and 4, 5 and 6 initialize jagged[2][0], jagged[2][1] and jagged[2][2], respectively. Therefore, array jagged in the preceding declaration is actu- ally composed of four separate one-dimensional arrays—one that represents the rows, one containing the values in the first row ({1, 2}), one containing the value in the second row ({3}) and one containing the values in the third row ({4, 5, 6}). Thus, array jagged itself is an array of three elements, each a reference to a one-dimensional array of int values.

Observe the differences between the array-creation expressions for rectangular arrays and for jagged arrays. Two sets of square brackets follow the type of jagged, indicating that this is an array of int arrays. Furthermore, in the array initializer, C# requires the key- word new to create an array object for each row. Figure 4.18 illustrates the array reference jagged after it's been declared and initialized.

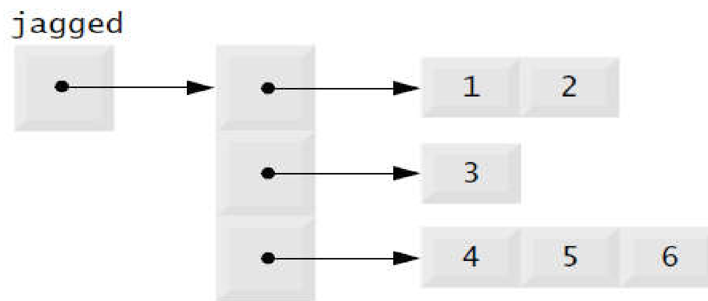

Fig. 4.18 | Jagged array with three rows of different lengths.

Creating Two-Dimensional Arrays with Array-Creation Expressions

A rectangular array can be created with an array-creation expression. For example, the following lines declare variable b and assign it a reference to a three-by-four rectangular array:

```
int[ , ] b;
b = new int[ 3, 4 ];
```

In this case, we use the literal values 3 and 4 to specify the number of rows and number of columns, respectively, but this is not required—applications can also use variables and expressions to specify array dimensions. As with one-dimensional arrays, the elements of a rectangular array are initialized when the array object is created.A jagged array cannot be completely created with a single array-creation expression. The following statement is a syntax error:

```
int[][] c = new int[ 2 ][ 5 ]; // error
```

Instead, each one-dimensional array in the jagged array must be initialized separately. A jagged array can be created as follows:

```
int[][] c;
c = new int[ 2 ][ ]; // create 2 rows
c[ 0 ] = new int[ 5 ]; // create 5 columns for row 0
c[ 1 ] = new int[ 3 ]; // create 3 columns for row 1
```

The preceding statements create a jagged array with two rows. Row 0 has five columns, and row 1 has three columns.

Two-Dimensional Array Example: Displaying Element Values

Figure 4.19 demonstrates initializing rectangular and jagged arrays with array initializers and using nested for loops to traverse the arrays (i.e., visit every element of each array).

Class InitArray's Main method creates two arrays. Line 12 uses nested array initializers to initialize variable rectangular with an array in which row 0 has the values 1, 2 and 3, and row 1 has the values 4, 5 and 6. Lines 17–19 uses nested initializers of different lengths to initialize variable jagged. In this case, the initializer uses the keyword new to create a one-dimensional array for each row. Row 0 is initialized to have two elements with values 1 and 2, respectively. Row 1 is initialized to have one element with value 3. Row 2 is initialized to have three elements with the values 4, 5 and 6, respectively.

Method OutputArray has been overloaded with two versions. The first version (lines 27–40) specifies the array parameter as int[,] array to indicate that it takes a rectangular array. The second version (lines 43–56) takes a jagged array, because its array parameter is listed as int[][] array.

```
1   // Fig. 4.19: InitArray.cs
2   // Initializing rectangular and jagged arrays.
3   using System;
4
5   public class InitArray
6   {
7      // create and output rectangular and jagged arrays
8      public static void Main( string[] args )
9      {
10        // with rectangular arrays,
11        // every column must be the same length.
12        int[ , ] rectangular = { { 1, 2, 3 }, { 4, 5, 6 } };
13
14        // with jagged arrays,
15        // we need to use "new int[]" for every row,
16        // but every column does not need to be the same length.
17        int[][] jagged = { new int[] { 1, 2 },
18                           new int[] { 3 },
19                           new int[] { 4, 5, 6 } };
20
```

```
21         OutputArray( rectangular ); // displays array rectangular by row
22         Console.WriteLine(); // output a blank line
23         OutputArray( jagged ); // displays array jagged by row
24      } // end Main
25
26      // output rows and columns of a rectangular array
27     public static void  OutputArray (int[ , ] array )
28      {
29         Console.WriteLine( "Values in the rectangular array by row are" );
30
31         // loop through array's rows
32         for ( int row = 0; row < array.GetLength( 0 ); row++ )
33         {
34            // loop through columns of current row
35            for ( int column = 0; column < array.GetLength( 1 ); column++ )
36               Console.Write( "{0}  ", array[ row, column ] );
37
38            Console.WriteLine(); // start new line of output
39         } // end outer for
40      } // end method OutputArray
41
42      // output rows and columns of a jagged array
43     public static void  OutputArray(int[][] array )
44      {
45         Console.WriteLine( "Values in the jagged array by row are" );
46
47         // loop through each row
48         foreach ( var row in array )
49         {
50            // loop through each element in current row
51            foreach ( var element in row )
52               Console.Write( "{0}  ", element );
53
54            Console.WriteLine(); // start new line of output
55         } // end outer foreach
56      } // end method OutputArray
57   } // end class InitArray
```

```
Values in the rectangular array by row are
1  2  3
4  5  6

Values in the jagged array by row are
1  2
3
4  5  6
```

Fig. 4.19 | Initializing jagged and rectangular arrays.

Line 21 invokes method OutputArray with argument rectangular, so the version of OutputArray at lines 27–40 is called. The nested for statement (lines 32–39) outputs the rows of a rectangular array. The loop-continuation condition of each for statement (lines 32 and 35) uses the rectangular array's GetLength method to obtain the length of each dimension. Dimensions are numbered starting from 0, so the method call GetLength(0) on array returns the size of the first dimension of the array (the number of rows), and the call GetLength(1) returns the size of the second dimension (the number of columns).

Line 23 invokes method OutputArray with argument jagged, so the version of OutputArray at lines 43–56 is called. The nested foreach statement (lines 48–55) outputs the rows of a jagged array. The inner foreach statement (lines 51–52) iterates through each element in the current row of the array. This allows the loop to determine the exact number of columns in each row. Since the jagged array is created as an array of arrays, we can use nested foreach statements to output the elements in the console window. The outer loop iterates through the elements of array, which are references to one-dimensional arrays of int values that represent each row.

Common Multidimensional-Array Manipulations Performed with for Statements

Many common array manipulations use for statements. As an example, the following for statement sets all the elements in row 2 of rectangular array a in Fig. 4.17 to 0:

```
for ( int column = 0; column < a.GetLength( 1 ); column++)
   a[ 2, column ] = 0;
```

We specified row 2; therefore, we know that the first index is always 2 (0 is the first row, and 1 is the second row). This for loop varies only the second index (i.e., the column index). The preceding for statement 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 statement totals the values of all the elements in array a:

```
int total = 0;

for ( int row = 0; row < a.GetLength( 0 ); row++ )
{
   for ( int column = 0; column < a.GetLength( 1 ); column++ )
      total += a[ row, column ];
} // end outer for
```

These nested for statements total the array elements one row at a time. The outer for statement begins by setting the row index to 0 so that row 0's elements can be totaled by the inner for statement. The outer for then increments row to 1 so that row 1's elements can be totaled. Then the outer for increments row to 2 so that row 2's elements can be totaled. The variable total can be displayed when the outer for statement terminates. In the next example, we show how to process a rectangular array in a more concise manner using foreach statements.

4.12 Variable-Length Argument Lists

Variable-length argument lists allow you to create methods that receive an arbitrary number of arguments. A one-dimensional array-type argument preceded by the keyword ***params*** *in a method's parameter list indicates that the method receives a variable number of arguments with the type of the array's elements. This use of a* ***params*** *modifier can occur only in the last entry of the parameter list. While you can use method overloading and ar- ray passing to accomplish much of what is accomplished with variable-length argument lists, using the* ***params*** *modifier is more concise.*

Figure 4.22 demonstrates method Average (lines 8–17), which receives a variable- length sequence of doubles (line 8). C# treats the variable-length argument list as a one- dimensional array whose elements are all of the same type. Hence, the method body can manipulate the parameter numbers as an array of doubles. Lines 13–14 use the foreach loop to walk through the array and calculate the total of the doubles in the array. Line 16 accesses numbers.Length to obtain the size of the numbers array for use in the averaging calculation. Lines 31, 33 and 35 in Main call method Average with two, three and four arguments, respectively. Method Average has a variable-length

argument list, so it can average as many double arguments as the caller passes. The output reveals that each call to method Average returns the correct value.

```
1  // Fig. 4.22: ParamArrayTest.cs
2  // Using variable-length argument lists.
3  using System;
4
5  public class ParamArrayTest
6  {
7     // calculate average
8     public static double Average( params double[] numbers )
9     {
10       double total = 0.0; // initialize total
11
12       // calculate total using the foreach statement
13       foreach ( double d in numbers )
14          total += d;
15
16       return total / numbers.Length;
17    } // end method Average
18
19    public static void Main( string[] args )
20    {
21       double d1 = 10.0;
22       double d2 = 20.0;
23       double d3 = 30.0;
24       double d4 = 40.0;
25
26       Console.WriteLine(
27          "d1 = {0:F1}\nd2 = {1:F1}\nd3 = {2:F1}\nd4 = {3:F1}\n",
28          d1, d2, d3, d4 );
29
30       Console.WriteLine( "Average of d1 and d2 is {0:F1}",
31          Average( d1, d2 ) );
32       Console.WriteLine( "Average of d1, d2 and d3 is {0:F1}",
33          Average( d1, d2, d3 ) );
34       Console.WriteLine( "Average of d1, d2, d3 and d4 is {0:F1}",
35          Average( d1, d2, d3, d4 ) );
36    } // end Main
37 } // end class ParamArrayTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 4.22 | Using variable-length argument lists.