Top of Form



**Introduction to C# Programming: Lesson 9**

**Chapter 1**

**Introduction**

Up to this point in the course, every time you created a variable, it was a simple variable. That meant that you declared the variable with some basic data type, and C# reserved a space in memory for a value. For each variable that you declared, you got exactly one spot in memory. Most of the variables you created were not really related to one another; therefore, you created simple variables. However, there will be times when you want C# to reserve a block of memory locations that are related to one another in some way. This block of memory is called an array.

In this lesson, you will learn what an array is and how to create one. You will then explore different ways to work with the data in an array and learn how to pass an array into and out of a method. Finally, you will learn about the two most important array manipulations: searching for an element in an array and sorting the elements of an array.

**Chapter 2**

**Creating Arrays**

In programming terminology, an *array* is a group of related variables that have the same name and same data type. For example, if you were asked to write a program that stored the names of your first three test scores, it would be very helpful to use an array. If you did not use an array, you would need three variables and three different variable names. In contrast, if you use an array, you would need one array variable. Often programmers will draw pictures to represent the data stored in memory. When they draw array variables, they usually draw either a column or a row of variables. Using the test scores example above, you might draw the array as follows:

  
Fig. 9.1. Representation of the three elements of the testScores array

Array variables will provide two important things to you as a programmer. First, array variables will allow you to better organize your code. Grouping your data together into an array variable will help you to keep all of your information together. Second, by using array variables, you will be able to write code that runs more efficiently. Oftentimes you will want your program to get data from an external file. However, reading data from files is a slow process for the computer, so you do not want to access external files very often. Instead, it would be better to get the data from the file and store it in an array variable. By the way, although this lesson will not deal with external data files, it will be covered later in the course.

Now that you have a little background about arrays, let's learn how to use them in our programs. The first thing that you need to do with an array variable is the same thing that you do with simple variables: declare them. To declare an array in C#, you type:

*<data type> [ ] <variable name> ;*

Notice how this line of code is in the same format as declaring a simple variable, except that there is the addition of square brackets [ ] after the data type. Another important difference between this statement and the declaration of a simple variable is regarding the allocation of memory. When you declare a simple variable, you immediately get a space in memory in which you can put a value. This is not the case with array variables. If you think about this, it should make sense. Remember that we said that an array is one or more spaces in memory. But when you declare an array variable using a statement like the one above, you do not specify the number of spaces in memory that you are needing.

To actually reserve the correct number of memory locations, you must use the reserved word *new* in the following format:

*<array name> = new <data type> [<number of memory locations>] ;*

For example, to declare an array named *testScores* that can hold three whole numbers, you would type:

int [] testScores;  
testScores = new int [3] ;

The first statement declares the testScores variable, and the second statement reserves three spots in memory for this variable. We will discuss the reason why this is done later in the course when we discuss classes. For now, just be sure to do both steps. Of course, many times you will want to declare the array variable and reserve the memory for it in the same step. This can be done in C# by combining the two statements as follows:

int [] testScores = new int [3] ;

Now, you may be curious about how you can store more than one value in one variable. This is done using what is called a subscript. The *subscript*, or *index*, of an array determines which *element*, or memory location, you are talking about. By using a unique index, C# can determine which memory location you are referring to in your code.

In C#, the arrays are *zero based*. That means that the elements are numbered starting with the number zero. That means that an array with three elements will have elements with subscripts of 0, 1, and 2. This can be very confusing for beginning programmers and can be an easy thing to forget, especially for people who already know how to program in another language that is *one based* (the first element has a subscript of one).

Now that you have declared your array variable and allocated the memory for the values, it is time to learn how to access the elements. This is actually quite an easy task. When you work with an individual array element, you can treat it exactly the same as a simple variable, except that you must remember to put the element's subscript value next to the array name in square brackets. For example, the following code will declare our array, reserve the memory, and then store the value *95* in the last element. Finally, this code will print that value out.

int [] testScores = new int [3] ;

testScores [2] = 95;  
Console.Out.WriteLine(testScores [2] );

Notice how the right-hand side of the assignment statement is exactly like that for a simple variable. The left side also looks similar except that there is a number surrounded by square brackets after the variable name.

When you learned how to declare your simple variables, you also learned that you could initialize those variables, or give them an initial value. This can also be done with array variables. By default when you declare an array, C# initializes the memory locations to the default values for that data type. The default for number variables is 0, for Boolean variables, it is false; and for characters and strings, it is null.

But there may be times when you want to initialize your elements to a value other than the default. For example, maybe you created your testScores variable to be used to calculate your test average for a course. Before the program starts, you might already know that your test scores are 100, 95, and 97. You could use three assignment statements. However, C# also allows you to list your initial values when you create the variable. There are three different versions of the list initializer statement in C#:

int [] testScores = new int [3] {100, 95,   
97};  
int [] testScores = new int [] {100, 95, 97};  
int [] testScores = {100, 95, 97};

Realize that in your program, you would only type one of the three lines of code above to declare your array variable and give it initial values. The first two statements are the same except that the second statement shows you that you do not have to tell C# how many elements there are in your array; it will count for you. The last statement shows that you do not even need to use the new operator. Providing three different ways to do this is just for convenience, you will find that you will like one method more than the others and tend to stick with that one.  
  
  
  
  
**Chapter 3**

**Reminder:** Whenever you see a " **¬** " character at the end of a line of code, this denotes a break in that code-line due to our HTML page size limitations. The remaining section of that code-line can be found on the next line below. After pasting your code into your text-editing program, simply select and delete that character, plus the hidden-break following it, to realign the code-line properly.

**Using Arrays**

Now that you know how to create your array variables and give them initial values, you need to learn what you can do with them and why they are so useful. The first thing I want to show you is how to print the elements in an array. You might think that this would be as easy as:

int [] testScores = new int [3] {100,   
95, 97};  
Console.Out.WriteLine(testScores);

Although these statements will compile, they will not give you what you think. Actually, running this code will display *System.Int32 []* on the screen. This is the data type of the testScores variable. This may be useful information and could be exactly what you were wanting. However, you wanted to print each of the elements of the array. To do this, you could just access each element of the array with a different line of code and type:

Console.Out.WriteLine(testScores [0] );  
Console.Out.WriteLine(testScores [1] );  
Console.Out.WriteLine(testScores [2] );

This code correctly displays the contents of each element in testScores. However, look at the redundant code that is present. Each line of code looks exactly the same, with the exception of the subscript. Since this is the case, it is better programming practice to write a loop as follows:

for(int count = 0; count < 3; count++)  
Console.Out.WriteLine(testScores [count] );

Notice how I took the redundant line of code and put it inside the loop. When I did this, I replaced the value of the index with the loop counter variable. In this example, I replaced three lines of code with two lines. This may not seem like a big deal to you, but imagine if my array had 10,000 elements. If I wanted to print out the values of each element, I would need only these same two lines of code (except I would have to change my loop condition). I think you will agree that this is very powerful. Using this same idea, you can now write code that will go through each element in an array and print the values, store the values, sum the values, whatever you want. All you need to know is how many elements are in the array.

But imagine the following example. Image if you wrote your program and you had many loops to go through each element of your testScores variable throughout your code. And just as you were finishing your program, you realized that there were four tests in your class and not three. How would you solve this problem? You might be thinking about using the **Find and Replace** feature of your word processor to replace all the 3s with 4s. However, even this can be time consuming and could lead to errors if you replaced a 3 that had nothing to do with the array.

To help you with this problem, C# provides a way to find out the number of elements in an array while the program is running. This is done using the Length field of the array variable. For example, to get the number of elements in your array, you could use:

testScores.Length

Rewriting the code above, the loop would become:

for(int count = 0; count < testScores.Length; ¬  
count++)  
Console.Out.WriteLine(testScores [count] );

You will see how this little feature will come in handy later when we discuss passing array variables as parameters.

Before we get to arrays as parameters, though, I want to show you one more way to navigate through the elements of an array: the *foreach* statement. Similar to the for statement, the foreach statement will go through each element of an array. However, this statement does not require you to use the subscripts explicitly. Again using the example from above, you could type:

foreach(int counter in testScores)  
Console.Out.WriteLine(counter);

Again, this is just provided by C# as a convenience. Some may find that this is very useful; others may think this is a little confusing. Just be sure to learn and practice both, and eventually you will find that you prefer one over the other depending on the situation.

Earlier I mentioned that you can pass array variables as parameters to your methods. Let's start by looking at how to pass one element of an array into a method and how to get an array element back from a method. I will show you how to do both of these in the same example.

using System;

public class ArrayMethod

{

public static void Main()

{

int[] arrayA = new int[5] {1, 2, 3, 4, 5};

int variableB = MyMethod1(arrayA[0]);

Console.Out.WriteLine("variableB is " + ¬

variableB);

}

public static int MyMethod1(int temp)

{

int[] tempArray = new int[5] {10, 9, 8, 7, 6};

Console.Out.WriteLine("temp is " + temp);

return tempArray[0];

}

}

Looking at this program, you will see that first an array named *arrayA* is created and given initial values. Next, method MyMethod1 is called and passed a single element from the array. You can see that this is a single element because of the square brackets and the subscript value. Notice the method header in MyMethod1. Do you see that in the parameter list all that is listed is a single int? Realize that the method does not know, or care, if that int is part of an array or not. Also, since the data type of arrayA [0] is an int, this value can be passed from Main to MyMethod1.

Next, MyMethod1 creates an array named *tempArray* and gives it some initial values. This method prints the value that was passed in and then returns a single element from its tempArray. Again, you can see that tempArray [0] has a data type of int, and therefore, the method header shows that this method returns a single int.

Passing single array elements in this way may not have been much of a surprise to you. If it was a little confusing, consider how we pass entire arrays to methods and back.

using System;

public class ArrayMethod2

{

public static void Main()

{

int[] arrayA = new int[5] {1, 2, 3, 4, 5};

arrayA = MyMethod2(arrayA);

for(int count=0; count < arrayA.Length; ¬

count++)

Console.Out.WriteLine(arrayA[count]);

}

public static int[] MyMethod2(int[] tempArray)

{

foreach(int counter in tempArray)

Console.Out.WriteLine(counter);

Console.Out.WriteLine();

int[] secondTempArray = new int[5] {10, 9, 8, ¬

7, 6};

return secondTempArray;

}

}

Once again, arrayA is created and initialized, but this time MyMethod2 is called and the entire array is passed to this method. Notice that the method header shows that a parameter of type int [] is expected. Therefore, the entire array is passed into a variable named *tempArray*, and then each element in this array is displayed on the screen. Next, the secondTempArray variable is created and initialized. Finally, this method returns the newly created array. Again, you should notice that the method header shows that an array of ints is being returned.

This array is passed back and is stored in Main's arrayA variable. This program finishes by printing each value in the new array.

One very interesting thing that needs to be pointed out about passing arrays to and from methods is the lack of the number of elements in the array. This is very important because that means that our methods can accept and return arrays with different numbers of elements. For example, you might want to write a method whose sole purpose is to print all the elements in an array. Using what you have learned about passing arrays and knowing about the Length property, you can now write:

public static void PrintArray(int[] tempArray)

{

for(int count=0; count < tempArray.Length;

count++)

Console.Out.WriteLine(tempArray[count]);

}

This method can print an array with any number of elements. The only requirement is that the array contains integers.

The final point I would like to make about passing arrays is regarding passing by reference versus passing by value. Remember when you first learned about passing variables to methods; you learned that all variables were passed by value. The same is true for variables that contain the entire array. That is, if you change the value of an array variable in a method, the effects will not be felt outside the method. However, if you change the value of an individual element inside a method, the effects will be felt outside the method. The following complete program demonstrates this idea.

using System;

public class ArrayChangeMethod

{

public static void Main()

{

int[] arrayA = new int[5] {1, 2, 3, 4, 5};

ChangeArray1(arrayA);

PrintArray(arrayA);

Console.Out.WriteLine();

ChangeArray2(arrayA);

PrintArray(arrayA);

}

public static void PrintArray(int [] tempArray)

{

for(int count=0; count < tempArray.Length; ¬

count++)

Console.Out.WriteLine(tempArray[count]);

}

public static void ChangeArray1(int[] tempArray1)

{

tempArray1= new int[5] {10, 9, 8, 7, 6};

}

public static void ChangeArray2(int[] tempArray2)

{

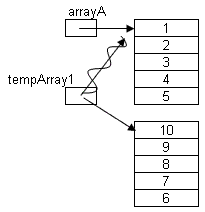
tempArray2[0] = 100;

}

}

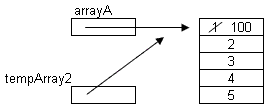
In this example, method ChangeArray1 changes the array to which arrayA is referencing. However, since this entire array was passed into the method by value, the effects are not felt outside the method. On the other hand, method ChangeArray2 is also passed an entire array, but this method changes the value stored in one of the elements of the array. Therefore, the effects of this change are felt outside the method.

This may seem a little confusing at first, but let me try to explain this with a picture of what is going on inside the computer. First, we will look at what happens inside ChangeArray1.

  
Fig. 9.2. Representation of memory values inside ChangeArray1

Notice how tempArray1 initially points to the same location as arrayA, but once inside the method, it is assigned to a new array. However, when control leaves the method, look where arrayA is pointing. Since only the value is in arrayA's memory location, that is the pointer to the place in memory where the array was stored, no effects were felt outside the method.

By contrast, look at what happened in ChangeArray2.

  
Fig. 9.3. Representation of memory values inside ChangeArray2

Here again, the value of arrayA's memory location was passed by value. But in ChangeArray2, the code did not attempt to change this value. Instead, it used this memory location to access the actual space in memory for element 0. It makes that change, and when control is passed back to the Main method, arrayA retains its original value, yet element 0 has been changed.

**Chapter 4**

**Searching and Sorting Arrays**

Now that you have learned a little more about how to use array variables, let's turn our attention to two of the most important and most widely used manipulations performed on arrays. The first array manipulation I want to talk about is searching an array for a specific value. For this example, let's think about a program that uses an array to store inventory identification numbers. You might want to write a program that will search through the array to determine if a certain number is a valid ID number. To make your code more flexible, you might decide to write a method that will accept an entire array of values and one value to which each element will be compared. Your method may look something like this:

public static Boolean CompareArray(int [] tempArray,

int value)

{

Boolean found = false;

for(int count=0; count < tempArray.Length;

count++)

if(tempArray[count] == value)

found = true;

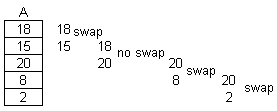
return found;

}

This method of searching is called *linear search* or *sequential search*, because it starts by looking at the first element. It continues by going through each element one by one sequentially until it reaches the end. This algorithm is by far the easiest to understand and is usually taught in all introductory programming courses. Unfortunately, while this method is the simplest to understand, it is also the slowest.

For example, imagine we are searching for an ID number that is not valid in an array of 100,000 elements. Think of how long that could possibly take. Fortunately, there are much faster and more efficient search methods available to us, but they come at a cost. The time for searching an array can be improved if our array elements have already been sorted into some order. Before we discuss better searching methods, let's first explore how to sort our arrays.

As the sequential search is the most common search algorithm for introductory programmers, the bubble sort is the most common sorting algorithm. The bubble sort algorithm works by comparing adjacent elements and interchanging the values of those elements when they are out of order.

  
Fig. 9.4. Picture of comparisons made during first pass of bubble sort

Notice how adjacent elements are compared and then, based on their values relative to one another, the values are interchanged. As you may have guessed, this is done using a loop. Notice after one pass through the loop, the largest value is stored in the last element.

Fig. 9.5. Representation of an array after first pass through bubble sort   
Fig. 9.5. Representation of an array after first pass through bubble sort

Notice that while the largest element is in place, none of the other values are sorted. This is because each pass through the loop only sorts a single number of the collection. For this reason, multiple passes are necessary for the bubble sort to work completely. The following code is for a method that will sort the elements of an integer array using the bubble sort algorithm.

public static int[] BubbleSort(int[] a)

{

int temp;

for(int j=1; j < a.Length; j++)

for(int i=0; i < a.Length-j; i++)

if(a[i] > a[i+1])

{

temp = a[i];

a[i] = a[i+1];

a[i+1] = temp;

}

return a;

}

You may now be wondering how important it is for you to understand how the bubble sort works. The answer is that it really isn't that important for you to understand exactly how it works. However, it is a useful piece of code that you might want to use in the future.

Actually, the writers of C# knew how common it would be for programmers to want to sort their data. For that reason, they provided a built-in method that you can use to sort your arrays. The method name is Sort(), and it can simply be used by writing:

*Array.Sort();*

This means that the above call to the BubbleSort method could have been replaced with the code:

Array.Sort(arrayA);

This method will arrange the values in the array in increasing order. C# also provides a method to reverse the elements in an array. This method is called Reverse() and is used the same way as the Sort() method. That is, the code is:

Array.Reverse(arrayA);

But be aware that Reverse() will NOT do any kind of sorting - either ascending or descending. Instead, Reverse() will simply reverse the order of the elements as they exist in the array.

Now you are probably really mad at me for showing you the bubble sort. You probably figure that you will never use it. However, I often find that students who learn how to do it for themselves first appreciate the built-in method better. Also, there is a possibility that you will decide that you want to learn to program in another language that does not have this built-in method. If this is the case, you will be happy that you learned about the bubble sort.

Finally, I want to close this lesson with a discussion of a better search algorithm called *binary search*. You have used the binary search algorithm before and probably never even knew it. For example, imagine how you would look for the number of Jo Smith in the phone book. First you might divide the book in half. You would probably be looking somewhere in the M's. At that point, you can eliminate the first half of the phone book, because you know that *Smith* comes after the M's. Next you might divide the half in your right hand in half again. You would continue to examine the page you opened to and determine if the name is in the left half or the right half. Eventually you would either end up at the listing for Jo Smith or you would not find the name at all. Realize that this algorithm only works if the elements, or in our case the names, are arranged in order.

I could go into more detail about how to implement the binary search algorithm with C# code, but C# has beaten me to the punch. Yes, you guessed it: The writers of C# figured that not only would it be very common for programmers to want to sort their arrays, it would also be very common for programmers to want to search their arrays for a value. The name of this method is BinarySearch(), and to use it, you would use the following syntax:

*Array.BinarySearch( , )*

Realize of course that this is a value-returning method, so that value must be either stored, printed, or used in the statement. The value that is returned is the index of the array element where the value is located. If the value is not found in the array, then a number less than zero is returned. For example if you wanted to print the index of the value 45.9 in an array of doubles called *values*, you would type:

Console.Out.WriteLine(Array.BinarySearch(  
values, 45.9));

This code would either print out the index where the value *45.9* is located in the values array or it would return a value less than zero.

**Chapter 5**

**Summary**

In this lesson, you have learned about array variables. You have seen that these variables are a block of memory that all have the same data type and the same name. Individual memory locations are referenced using a unique index or subscript.

You next learned different ways to work with these array variables. This included creating array variables, allocating memory for the elements, and giving the memory locations initial values. You then saw how to print all of the elements of an array to the screen and how to pass individual elements to a method. You also learned that entire arrays can be passed to and returned from a method.

Finally, this lesson showed you different ways to search for values in an array.

All of these different ways of using an array are helpful because arrays are an excellent way to manage and organize data within your programs. However, there is a limitation to arrays: Each variable can only hold data of one type. In the next lesson, you will learn about classes. Classes are the heart and soul of object-oriented programming because they allow you to store related pieces of information about things. Unlike arrays, classes allow you to store pieces of data that are of different types. In addition, classes also allow a way to write methods to manipulate those data items. This may seem a little confusing to you at this point in time, but after completing the next lesson, you will fully understand what a class is and how to use one.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [Programming Arrays in C#](http://www.c-sharpcorner.com/UploadFile/mahesh/WorkingWithArrays11232005064036AM/WorkingWithArrays.aspx)  http://www.c-sharpcorner.com/UploadFile/mahesh/WorkingWithArrays11232005064036AM/WorkingWithArrays.aspx |
| This page starts with a short description of one-dimensional arrays and moves on to discuss multidimensional, jagged, and mixed arrays. It also lists methods and properties of the Array class. |

**FAQs**   
  
**Q:** All of the examples that you showed us used integers and doubles. Can you declare an array of Strings or other Objects in C#?  
  
**A:** Yes, C# allows you to declare arrays of Objects. I used examples of primitive types because I did not want you to get caught up in the details of Objects just yet.  
  
  
**Q:** When I have my program call the Array.Sort() method, it always makes all of the values in my array zero. Why is this happening?  
  
**A:** This is a common problem and often happens when you try to sort an array that is not full. For example, if you create an array that can hold 50 integers and you only put 25 integers into the array and then sort them. Most people will only try to print out the first 25 elements, because they know that the remaining elements have not been given values. The problem is that there were values in those last 25 elements and that value was 0. When the data is sorted, the 0 values get sorted to the start of the list. That way when the first 25 elements are printed, only the 0's show.

**Assignment**   
  
  
Write a program that prompts the user for a list of up to 100 numbers to be stored in an array. The program should stop asking for numbers if the user enters the value *999* or if the array is full. When the user is finished entering values, calculate the average of only those numbers entered.

[Click here for solution: **SumNumbersArray.zip**](https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/SumNumbersArray.zip)

|  |
| --- |
| Copyright © 1997 - 2012 Education To Go. All rights reserved. The material on this site cannot be reproduced or redistributed unless you have obtained prior written permission from Education To Go. Education To Go and ed2go are registered trademarks of Education To Go, a part of Cengage Learning. |

Bottom of Form