Arrays

The array is a data structure representing a sequence of items of the same type. The access to the elements of an array is direct usually through the consecutive index of the item.

The indexing of C# arrays is zero based. This means that the first element of an array is at position zero. Let's see a diagram of an array of four integers – 100, 200, 300 and 400:

index	0	1	2	3
value	100	200	300	400

The syntax of declaring and accessing elements of the array is the following:

```
data_type[] variable_name = new data_type[size_of_the_array];
variable_name[index] = value;
variable_name[index];
```

All the items in an array are of the same type. See in the syntax above that the array is declared using the data type of its elements followed by square brackets. The arrays are a reference type thus the initial value of "variable_name" would be "null". That is why we need to create a new array providing its size explicitly as per the syntax example above.

Assigning values to the elements of the array is done through the element index as per the example above. Accessing values is done the same way.

Arrays vary based on dimensions and type. The example above showed us the syntax for working with standard one dimension arrays. Let's see the syntax for working with two dimension arrays:

```
data_type[,] variable_name = new data_type[size, size];
variable_name[index, index] = value;
variable_name[index, index];
```

In this case we're working with a matrix (two dimension array). Let's see a diagram showing a two dimensional array with size 3x3:

index / index	0	1	2
0	101	102	103
1	201	202	203
2	301	302	303

As you can see in multidimensional arrays elements are positioned at the intersections of the dimensions. For accessing elements and assigning values we need to specify the element's position by giving an index for each dimension. Declaring a three dimensional array (or an array of higher dimension) is done analogically.

There are two types of arrays in C# – rectangular arrays (the arrays that we've seen so far) and jagged arrays. Since we've already discussed rectangular arrays let's take a closer look at the jagged arrays.

When speaking of single-dimension arrays there are no differences between jagged and rectangular arrays. The difference appears clearly when speaking of multidimensional arrays.

In standard rectangular arrays the number of elements of the array is the product of the sizes of all dimensions. In jagged arrays this is not true. They are not a monolith construct as rectangular arrays but more like arrays of arrays. Let's see the syntax for declaring a two dimensional jagged array:

```
data_type[][] variable_name = new data_type[size][];
for(int i=0; i<size; i++)
{
   variable_name[i] = new data_type[size];
}
variable_name[index][index] = value;
variable_name[index][index];
```

By syntax the jagged arrays are first declared with the size of only one of their dimensions. Then for each element of this dimension we declare a new array. The new arrays don't need to be of the same size. This is where they get their name – jagged arrays:

index / index	0	1	2
0	101	102	103
1	201		
2	301	302	

Declaring arrays

Let's start by declaring the one-dimensional array we used as an example earlier:

```
int[] arr = new int[4];
arr[0] = 100;
arr[1] = 200;
arr[2] = 300;
arr[3] = 400;
```

This is a sample of declaration and initialization of one-dimensional array. Let's see the declaration and initialization of the other two examples – the rectangular and the jugged two-dimensional arrays.

```
int[,] arr = new int[3, 3];
arr[0, 0] = 101;
arr[0, 1] = 102;
arr[0, 2] = 103;
arr[1, 0] = 201;
arr[1, 1] = 202;
arr[1, 2] = 203;
arr[2, 0] = 301;
arr[2, 1] = 302;
arr[2, 2] = 303;
```

The last example will be the two-dimensional jugged array:

```
int[][] arr = new int[3][];
arr[0] = new int[3];
arr[1] = new int[1];
arr[2] = new int[2];

arr[0][0] = 101;
arr[0][1] = 102;
arr[0][2] = 103;
arr[1][0] = 201;
arr[2][0] = 301;
arr[2][1] = 302;
```

Note how in the jugged array declaration we only specify the first dimension and then for each of its elements we create new arrays thus making the structure array of arrays.

Arrays can be initialized on declaration using the following syntax:

```
data_type[] variable_name = new data_type[size]{item1_value, item2_value, …};
data_type[,] variable_name = new data_type[size, size]{{item11, item12, … },{item21, item22,…},…}
data_type[][] variable_name = new data_type[][]{
   new data_type[]{item11, item12, …},
   new data_type[]{item21, item22, …},
   …
}
```

The initialization of one-dimensional arrays is done by explicitly listing the values after the declaration in curly brackets. In case of multidimensional arrays each dimension is surrounded in curly brackets and at the bottom level (the dimension intersections) we list the values. Let's see how this works:

```
int[] arr1 = new int[4] { 101, 102, 103, 104 };
int[,] arr2 = new int[3, 3] { {101, 102, 103}, {201, 202, 203}, {301, 302, 303} };
int[][] arr3 = new int[][] {
    new int[]{ 101, 102, 103 },
    new int[]{ 201 },
    new int[]{ 301, 302 }
};
```

Looping through arrays and accessing elements

Let's start by doing a simple program that prints all elements of a one-dimensional array to the standard output:

```
using System;

namespace OutputAllItems
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] arr = new double[] { 3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7 };

            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine("Element at index [{0}] is {1:F2}", i, arr[i]);
            }

            Console.ReadKey(true);
        }
    }
}
```

At the first line we declare the array of "double" values and initialize it on declaration. With a "for" loop we iterate through all elements of the array and output its value to the standard output.

The "Length" is a property of all arrays. It represents the total number of elements in all dimensions of the array. In our case since we have a single dimension array we can use it to iterate though the indexes of our dimension.

Let's take a look at another example where we need to find the maximum element in a single-dimension array:

```
using System;
namespace OutputMaximumElement
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] arr = new double[] { 3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7 };
            double max = arr[0];
            for (int i = 1; i < arr.Length; i++)
            {
                if (max < arr[i])
                    max = arr[i];
            }
            Console.WriteLine("The maximum element is {0:F2}", max);
            Console.ReadKey(true);
        }
    }
}
```

In order to do that we need to declare a helper variable that will hold the maximum value we have found so far while iterating in the loop. We initialize our “max” variable with the first element of the array and we start iterating from the second element of the array (we don’t need to check the first element against itself). On each iteration we check whether the element we’re on is greater than the maximum value we have found so far. If so we assign its value to the “max” variable so that it would once again hold the maximum value we’ve encountered so far. When the loop goes over all the items of the array (it becomes greater than arr.Lenght) then “max” will hold the maximum value.

There are cases when we need the index of the element with the biggest value in an array. In these cases instead of the maximum value (“max” variable) we keep track of the index of the maximum value. Let’s see an example:

```
using System;

namespace OutputMaximumElement
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] arr = new double[] { 3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7 };
            int max_index = 0;

            for (int i = 1; i < arr.Length; i++)
            {
                if (arr[max_index] < arr[i])
                {
                    max_index = i;
                }
            }

            Console.WriteLine("The maximum element is {0:F2}", arr[max_index]);
            Console.ReadKey(true);
        }
    }
}
```

In this case we’re comparing two elements of the array instead of copying the value of the biggest element in a helper variable. This is accomplished by storing the index instead of the value itself. Let’s see another example. This time we’re writing a program that will identify the minimum number of bills we need in order to reach a certain amount of cash.

At the first line of the example below we define a list of the available bills in our currency. Now that that’s done we read the amount we want to represent with a minimum number of bills from the standard input and assign it to the “amount” variable.

In order to find the representation of the amount with minimum number of bills we need to start with the bills of highest amount and use as many as we can. When we can’t use them anymore we should move to the bill with second highest amount and so on until we reach the bill of minimal amount.

In order to do that we need to iterate through the array starting from the last element (the highest value bill) and then move our way to the first element of the array. That is why we initialize our iterator with the last element of the array (bills.Length – 1) and for iteration statement we decrement "i".

In the body of the loop we check how many of bills[i] we can use to fulfill the amount. To do that we divide the amount by the bill's value. If we get something different from 0 then we can use "num" number of bills of this amount. Based on how many bills we're using we output a message (in singular or plural) stating the number of bills and the bill amount.

The rest of the amount is calculated (amount % bills[i]) which is the remainder after dividing the amount by the bill's amount. If the amount is zero then we have successfully solved our problem and should break the loop. If not we proceed to the next iteration.

```
using System;

namespace AmountByMinimumBills
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] bills = new int[] { 1, 2, 5, 10, 20, 50, 100 };

            int amount;
            Console.Write("Enter payment amount: ");
            amount = int.Parse(Console.ReadLine());

            int num;
            Console.WriteLine("We may pay the amount with");

            for (int i = (bills.Length - 1); i >= 0; i--)
            {
                num = amount / bills[i];
                if (num == 1)
                    Console.WriteLine("1 bill of {0}", bills[i]);
                else if (num > 1)
                    Console.WriteLine("{0} bills of {1}", num, bills[i]);

                amount = amount % bills[i];
                if (amount == 0)
                    break;
            }

            Console.ReadKey(true);
        }
    }
}
```

Populating and looping through arrays

In order to work with arrays we need to be able to populate them appropriately. This is what we're going to do in our next example. Let's populate a one-dimensional array and output all its elements with their index in the array:

```
using System;

namespace PopulatingOneDimensionalArrays
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("How many elements will you enter ...");
            int n = int.Parse(Console.ReadLine());

            int[] arr = new int[n];
            for (int i = 0; i < n; i++)
            {
                Console.Write("[{0}] = ", i);
                arr[i] = int.Parse(Console.ReadLine());
            }

            Console.Clear();
            Console.WriteLine("Outputing elements in reverse:");
            for (int i = n - 1; i >= 0; i--)
            {
                Console.WriteLine("[{0}] = {1}", i, arr[i]);
            }

            Console.ReadKey(true);
        }
    }
}
```

Here we ask the user for the number of elements he'd like to enter. After that we create an array of appropriate size and read the elements in a loop.

On the next line we clear the console and loop through the array in reverse (the iterator is initialized with (n-1) – the last index in the array and the iteration statement decrements "i".

Let's take a look at a more complex example. Let's read a matrix (two-dimensional array) from the standard input and output the elements of the row with the greatest total amount.

In the example below we first need to specify the size of the two dimensions, declare the array and populate it with data. We urge the user to give us the size of the "x" and "y" dimension and declare the array.

Note that in order to populate the array we need to go through all its elements. In order to do that we need two nested loop (two because we're currently working with a two-dimensional array the number of nested loops depends on the dimensions count). In our case the iterator of the first loop will hold the index of the row we're currently visiting. The iterator of the second (nested) loop will hold the index of the column we're currently visiting. The combination of the two indexes will give us access to the array element at that position.

In the body of the second loop we're asking the user for the value of the element at position i, j , read a value from the standard input and assign it to the corresponding array element.

```
using System;

namespace MaxValueRow
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter matrix dimension [x] size:");
            int dimensionX = int.Parse(Console.ReadLine());

            Console.Write("Enter matrix dimension [y] size:");
            int dimensionY = int.Parse(Console.ReadLine());

            int[,] arr = new int[dimensionX, dimensionY];
            for (int i = 0; i < dimensionX; i++)
            {
                for (int j = 0; j < dimensionY; j++)
                {
                    Console.Write("[{0}, {1}]:", i, j);
                    arr[i, j] = int.Parse(Console.ReadLine());
                }
            }

            Console.Clear();
            int maxSum = int.MinValue;
            int maxSumRowIndex = -1;
            for (int i = 0; i < dimensionX; i++)
            {
                int tempSum = 0;
                for (int j = 0; j < dimensionY; j++)
                {
                    tempSum += arr[i, j];
                }

                if (tempSum > maxSum)
                {
                    maxSum = tempSum;
                    maxSumRowIndex = i;
                }
            }

            Console.WriteLine(
                    "The row with maximum summed values is {0} with a total sum of {1}",
                    maxSumRowIndex, maxSum);

            for (int j = 0; j < dimensionY; j++)
            {
                Console.Write("{0}\t", arr[maxSumRowIndex, j]);
            }

            Console.ReadKey(true);
        }
    }
}
```

After the array is properly populated we clear the console and start looking for the row with a maximum total value. In order to do that we need two helper variables – one for holding the

index of the row with max value that has been found so far and one for holding the maximum value itself. That's why we declare the two integer variables "maxSum" and "maxSumRowIndex". Since the "maxSum" variable will be used for comparison we need to initialize it properly in order to make sure that the first comparison will not fail.

Let's consider the case where we initialize the "maxSum" variable with 100 and the sum of the elements of the row with a total maximum valiu is 50. Then the algorithm will never find a row in the array that has a total value greater than the value of the "maxSum" value and the algorithm will fail.

To prevent this case we initialize the "maxSum" with the minimum value for the integer type. All numeric types expose the minimum and maximum values that a variable of this type can hold through the properties "MinValue" and "MaxValue". This way we make sure that the first comparison will be properly executed. In case the total sum of the elements of the first row is equal to int.MinValue the algorithm will still work since we have already initialized the "maxSumRowIndex" with the index of the first row.

In the first loop we declare a local variable "tempSum" whose scope is within the body of the first loop. This variable will aggregate the sum of the elements of the row we're currently visiting. The second loop takes care of the aggregation of the values of the row elements. Right after the nested loop ends the "tempSum" variable holds the aggregated sum. We compare it with the maximum that we've found so far (it is stored in the "maxSum" variable) and if the "tempSum" is greater we store the index and the sum of the elements of the current row.

At the end we output the index of the row with maximum total value and then list its elements. Note how we fix the row index and only iterate through the column indexes in order to display the items of the row.

Foreach statements

The "foreach" statement is a special kind of loop. Let's consider the case where we have a one-dimension array and we need to find the elements with minimum and maximum value.

In the example below we first declare our array and initialize it on declaration. In order to find the min and max values we declare two helper functions and initialize them with the maximum and minimum values for their type (as per the previous example).

On the next line we use the "foreach" iteration statement to go through the elements of the array. The "element" variable's scope is limited to the execution of the iterations. It is consequently assigned with the value of each element of the array. We check its value and adjust the "maxValue" and "minValue" variables using "if" statements. At the end of the loop the two helper variables hold the maximum and minimum values of the array.

At the end we output that information to the standard output.

```
using System;

namespace MinMaxValueElement
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] arr = new double[] { 3, 7, 1, 2, 3.5, 1, 12, 7.3, 12, 4.7 };
            double maxValue = double.MinValue;
            double minValue = double.MaxValue;

            foreach (double element in arr)
            {
                if (maxValue < element)
                    maxValue = element;
                if (minValue > element)
                    minValue = element;
            }

            Console.WriteLine("The minimum value element is {0}", minValue);
            Console.WriteLine("The maximum value element is {0}", maxValue);

            Console.ReadKey(true);
        }
    }
}
```

Value types and reference types

As mentioned before there are two main data type branches in C# – the value types and the reference types. We've talked about how they manage their values, but until now we have not considered working with a reference type.

Arrays are a great example how assignment differs between value and reference types. In order to illustrate let's have a simple example:

```
using System;
namespace ValuesAndReferences
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 1, b = a;
            Console.WriteLine(a);
            Console.WriteLine(b);
            b = 2;
            Console.WriteLine(a);
            Console.WriteLine(b);
            Console.ReadKey(true);
        }
    }
}
```

On the first line we declare two integer variables (integer is a value type) – “a” and “b”. We assign “a” with 1 and we assign “b” with the value of “a” (since “int” is a value type the value will be copied).

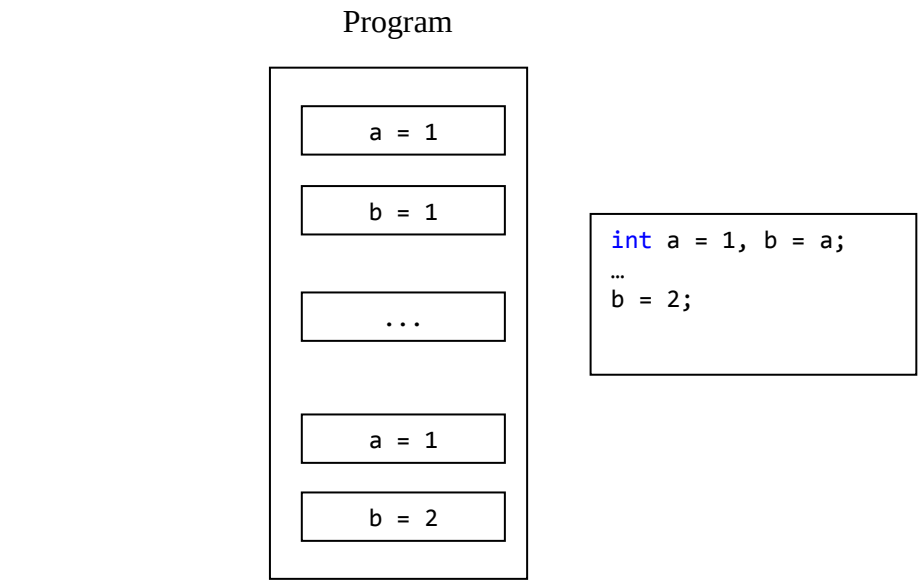
On the next two lines we output the values of “a” and “b”. On the standard output we’ll get the values of “a” and “b” on new lines. As expected we get:

```
1
1
```

On the next line we assign “b” with the value 2 and we output both variables again. As expected the value of “b” is 2 and the value of “a” is 1:

```
1
2
```

Let’s see how this is represented in-memory:

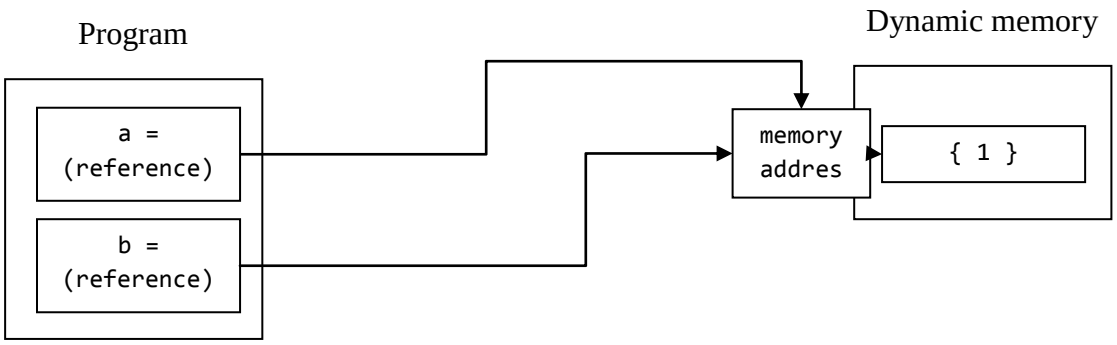


When we assign the value of “a” to “b” we create a copy of the value of “a” and then assign it to “b”. Then later when we change the value of “b” to 2 “a” is not affected.

This works a bit different for reference types. Let’s make “a” and “b” arrays of integer values (arrays are reference types).

On the first line we declare a one dimentional array “a” of one element and we assign it on declaration with the value 1. Then we declare “b” and assign it with the value of “a”.

The big difference here is that we’re working with reference types thus we woun’t make a copy of the values but copy the reference to the instance. Now both “a” and “b” point to the same instance. Let’s illustrate this with a diagram:

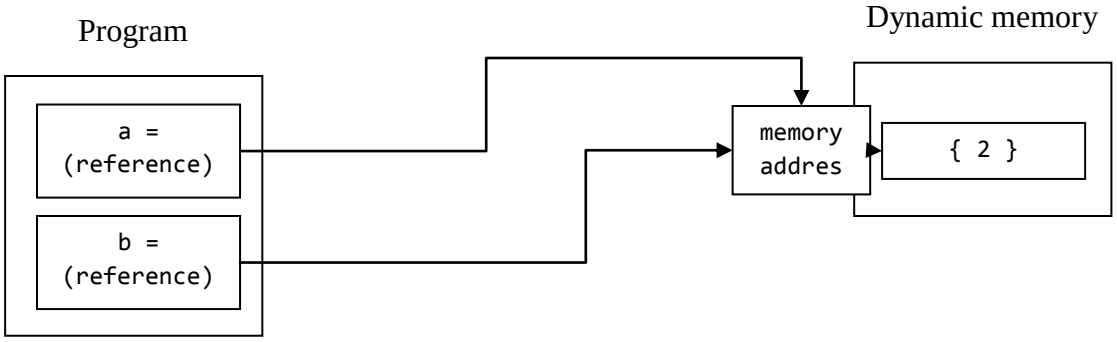


When we declare the “a” variable we create an instance of an array of one element of type integer and then we assign its element with the value 1. When we declare “b” we declare a new reference that can point to instances of array of type integer. We assign it with the reference that “a” currently holds (it currently references our 1 element array of type integer).

The output from the next two lines will be as expected:

```
1
1
```

The next line changes the value of the first element of the array referenced by “b” to 2. Note that “a” and “b” both point to the same instance of array of integers. Let’s see how this changes our diagram:



When we execute the next two lines we will get the following output:

```
2
2
```

That is because the first line will try to output the value of “a”. Since “a” references the instance on the right of the diagram we will get 2 as output. The next line outputs the value of “b”. The reference “b” points to the same instance in the memory thus the output will be the same – 2.

```
using System;
namespace ValuesAndReferences
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] a = { 1 }, b = a;
            Console.WriteLine(a[0]);
            Console.WriteLine(b[0]);
            b[0] = 2;
            Console.WriteLine(a[0]);
            Console.WriteLine(b[0]);
            Console.ReadKey(true);
        }
    }
}
```