Arrays

# Arrays in General

Array is an inbuilt data structure in the .NET framework which will store **homogenous elements** (an array of Int32s, An array of strings, an array of double etc). This way, Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Example :

1. Int32[] mynumbers = {22, 44, 24, 56, 563,108, 345, 107, 259}
2. string[] names = {“ganesh”, “purushottham ”, “vijay”, “sanjay”, “rahul”, “Latha”, “Edwina”, “Mathura”, “Bengaluru”}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Array is an inbuilt data structure in the .NET framework which will store homogenous elements (an array of Int32s, An array of strings, an array of double etc).

C# arrays are zero indexed; that is, the array indexes start at zero. Arrays in C# work similarly to how arrays work in most other popular languages There are, however, a few differences that you should be aware of.

When declaring an array, the square brackets ([ ]) must come after the type, not the identifier. Placing the brackets after the identifier is not legal syntax in C#.

int[] table; // not int table[];

Another detail is that the size of the array is not part of its type as it is in the C language. This allows you to declare an array and assign any array of **int** objects to it, regardless of the array's length.

int[] numbers; // declare numbers as an int array of any size

numbers = new int[10]; // numbers is a 10-element array

numbers = new int[20]; // now it's a 20-element array

## Declaring Arrays

C# supports single-dimensional arrays, multidimensional arrays (rectangular arrays), and array-of-arrays (jagged arrays). The following examples show how to declare different kinds of arrays:

**Single-dimensional arrays:**

int[] numbers;

**Multidimensional arrays:**

string[ , ] names;

**Array-of-arrays (jagged):**

byte[][] scores;

Declaring them (as shown above) does not actually create the arrays. In C#, arrays are objects (discussed later in this tutorial) and must be instantiated. The following examples show how to create arrays:

**Single-dimensional arrays:**

int[] numbers = new int[5];

**Multidimensional arrays:**

string[,] names = new string[5,4];

[ [3,4,4,5], [3,6,3,4], [], [], [] ]

**Array-of-arrays (jagged):**

byte[][] scores = new byte[5][];

for (int x = 0; x < scores.Length; x++)

{

scores[x] = new byte[4];

}

You can also have larger arrays. For example, you can have a three-dimensional rectangular array:

int[,,] buttons = new int[4,5,3];

You can even mix rectangular and jagged arrays. For example, the following code declares a single-dimensional array of three-dimensional arrays of two-dimensional arrays of type **int**:

int[][,,][,] numbers;

**Example**

The following is a complete C# program that declares and instantiates arrays as discussed above.

// arrays.cs

using System;

class DeclareArraysSample

{

public static void Main()

{

// Single-dimensional array

int[] numbers = new int[5];

// Multidimensional array

string[,] names = new string[5,4];

// Array-of-arrays (jagged array)

byte[][] scores = new byte[5][];

// Create the jagged array

for (int i = 0; i < scores.Length; i++)

{

scores[i] = new byte[i+3];

}

// Print length of each row

for (int i = 0; i < scores.Length; i++)

{

Console.WriteLine("Length of row {0} is {1}", i, scores[i].Length);

}

}

}

**Output**

Length of row 0 is 3

Length of row 1 is 4

Length of row 2 is 5

Length of row 3 is 6

Length of row 4 is 7

**Initializing Arrays**

C# provides simple and straightforward ways to initialize arrays at declaration time by enclosing the initial values in curly braces ({}). The following examples show different ways to initialize different kinds of arrays.

**Note**If you do not initialize an array at the time of declaration, the array members are automatically initialized to the default initial value for the array type. Also, if you declare the array as a field of a type, it will be set to the default value null when you instantiate the type.

**Single-Dimensional Array**

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

string[] names = new string[3] {"Matt", "Joanne", "Robert"};

You can omit the size of the array, like this:

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

string[] names = new string[] {"Matt", "Joanne", "Robert"};

You can also omit the **new**operator if an initializer is provided, like this:

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

string[] names = {"Matt", "Joanne", "Robert"};

**Multidimensional Array**

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

string[,] siblings = new string[2, 2] { {"Mike","Amy"}, {"Mary","Albert"} };

You can omit the size of the array, like this:

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

string[,] siblings = new string[,] { {"Mike","Amy"}, {"Mary","Albert"} };

You can also omit the **new**operator if an initializer is provided, like this:

int[ , ] numbers = { {1, 2}, {3, 4}, {5, 6} };

string[,] siblings = { {"Mike", "Amy"}, {"Mary", "Albert"} };

**Jagged Array (Array-of-Arrays)**

You can initialize jagged arrays like this example:

int[][] numbers = new int[3][2]

{

new int[] {2,3},

new int[] {5,6},

new int[] {9,6}

};

You can also omit the size of the first array, like this:

int[][] numbers = new int[][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };

-or-

int[][] numbers = { new int[] {2,3,4}, new int[] {5,6,7,8,9} };

Notice that there is no initialization syntax for the elements of a jagged array.

**Accessing Array Members**

Accessing array members is straightforward and similar to how you access array members in C/C++. For example, the following code creates an array called numbers and then assigns a 5 to the fifth element of the array:

int[] numbers = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

numbers[5] = 5;

The following code declares a multidimensional array and assigns 5 to the member located at [1, 1]:

int[,] numbers = { {1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10} };

numbers[1, 1] = 5;

The following is a declaration of a single-dimension jagged array that contains two elements. The first element is an array of two integers, and the second is an array of three integers:

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

};

The following statements assign 58 to the first element of the first array and 667 to the second element of the second array:

numbers[0][0] = 58;

numbers[1][1] = 667;

**Arrays are Objects**

In C#, arrays are actually objects. **System.Array** is the abstract base type of all array types. You can use the properties, and other class members, that **System.Array** has. An example of this would be using the Lengthproperty to get the length of an array. The following code assigns the length of the numbers array, which is 5, to a variable called LengthOfNumbers:

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

int LengthOfNumbers = numbers.Length;

The **System.Array** class provides many other useful methods/properties, such as methods for sorting, searching, and copying arrays.

**Using foreach on Arrays**

C# also provides the **foreach** statement. This statement provides a simple, clean way to iterate through the elements of an array. For example, the following code creates an array called numbers and iterates through it with the **foreach** statement:

int[] numbers = {4, 5, 6, 1, 2, 3, -2, -1, 0};

foreach (int i in numbers)

{

System.Console.WriteLine(i);

}

With multidimensional arrays, you can use the same method to iterate through the elements, for example:

int[,] numbers = new int[3, 2] {{9, 99}, {3, 33}, {5, 55}};

foreach(int i in numbers)

{

Console.Write("{0} ", i);

}

The output of this example is:

9 99 3 33 5 55

However, with multidimensional arrays, using a nested **for** loop gives you more control over the array elements.

# Bit Arrays

The BitArray class manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on i.e, 1 and false indicates the bit is off i.e, 0. This class is contained in System.Collections namespace.

**Properties of BitArray Class:**

* The BitArray class is a collection class in which the capacity is always the same as the count.
* Elements are added to a BitArray by increasing the Length property.
* Elements are deleted by decreasing the Length property.
* Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.