Contents

Arrays

- Arrays as Objects
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays
- Using foreach with Arrays
- Passing Arrays as Arguments
- Implicitly Typed Arrays

Arrays (C# Programming Guide)

1/24/2019 • 2 minutes to read • Edit Online

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.

```
type[] arrayName;
```

The following example creates single-dimensional, multidimensional, and jagged arrays:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Array Overview

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Related Sections

- Arrays as Objects
- Using foreach with Arrays
- Passing Arrays as Arguments

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Collections

Arrays as Objects (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++. Array is the abstract base type of all array types. You can use the properties, and other class members, that Array has. An example of this would be using the Length property 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 Array class provides many other useful methods and properties for sorting, searching, and copying arrays.

Example

This example uses the Rank property to display the number of dimensions of an array.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array:
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

See also

- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays

Single-Dimensional Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can declare a single-dimensional array of five integers as shown in the following example:

```
int[] array = new int[5];
```

This array contains the elements from `array[0]` to `array[4]`. The new operator is used to create the array and initialize the array elements to their default values. In this example, all the array elements are initialized to zero.

An array that stores string elements can be declared in the same way. For example:

```
string[] stringArray = new string[6];
```

Array Initialization

It is possible to initialize an array upon declaration, in which case, the length specifier is not needed because it is already supplied by the number of elements in the initialization list. For example:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

A string array can be initialized in the same way. The following is a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

When you initialize an array upon declaration, you can use the following shortcuts:

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

It is possible to declare an array variable without initialization, but you must use the `new` operator when you assign an array to this variable. For example:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

C# 3.0 introduces implicitly typed arrays. For more information, see Implicitly Typed Arrays.

Value Type and Reference Type Arrays

Consider the following array declaration:

```
SomeType[] array4 = new SomeType[10];
```

The result of this statement depends on whether `SomeType` is a value type or a reference type. If it is a value type, the statement creates an array of 10 elements, each of which has the type `SomeType`. If `SomeType` is a reference type, the statement creates an array of 10 elements, each of which is initialized to a null reference.

For more information about value types and reference types, see Types.

See also

- Array
- C# Programming Guide
- Arrays
- Multidimensional Arrays
- Jagged Arrays

Multidimensional Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns.

```
int[,] array = new int[4, 2];
```

The following declaration creates an array of three dimensions, 4, 2, and 3.

```
int[, ,] array1 = new int[4, 2, 3];
```

Array Initialization

You can initialize the array upon declaration, as is shown in the following example.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[, ,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                 { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[, ,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                       { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++) {
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

You also can initialize the array without specifying the rank.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

If you choose to declare an array variable without initialization, you must use the `new` operator to assign an array to the variable. The use of `new` is shown in the following example.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

The following example assigns a value to a particular array element.

```
array5[2, 1] = 25;
```

Similarly, the following example gets the value of a particular array element and assigns it to variable

`elementValue`.

```
int elementValue = array5[2, 1];
```

The following code example initializes the array elements to default values (except for jagged arrays).

```
int[,] array6 = new int[10, 10];
```

See also

- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Jagged Arrays

Jagged Arrays (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[3][];
```

Before you can use `jaggedArray`, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

You can also initialize the array upon declaration like this:

```
int[][] jaggedArray2 = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

You can use the following shorthand form. Notice that you cannot omit the `new` operator from the elements initialization because there is no default initialization for the elements:

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.

You can access individual array elements like these examples:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

It is possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes. For more information about two-dimensional arrays, see Multidimensional Arrays.

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

You can access individual elements as shown in this example, which displays the value of the element `[1,0]` of the first array (value `5`):

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

The method `Length` returns the number of arrays contained in the jagged array. For example, assuming you have declared the previous array, this line:

```
System.Console.WriteLine(jaggedArray4.Length);
```

returns a value of 3.

Example

This example builds an array whose elements are themselves arrays. Each one of the array elements has a different size.

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements:
        int[][] arr = new int[2][];

        // Initialize the elements:
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements:
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

See also

- Array
- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays

Using foreach with arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The foreach statement provides a simple, clean way to iterate through the elements of an array.

For single-dimensional arrays, the `foreach` statement processes elements in increasing index order, starting with index 0 and ending with index `Length - 1`:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

However, with multidimensional arrays, using a nested for loop gives you more control over the order in which to process the array elements.

See also

- Array
- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays

Passing arrays as arguments (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

Arrays can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

Passing single-dimensional arrays as arguments

You can pass an initialized single-dimensional array to a method. For example, the following statement sends an array to a print method.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

The following code shows a partial implementation of the print method.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Example

In the following example, an array of strings is initialized and passed as an argument to a `DisplayArray` method for strings. The method displays the elements of the array. Next, the `ChangeArray` method reverses the array elements, and then the `ChangeArrayElements` method modifies the first three elements of the array. After each method returns, the `DisplayArray` method shows that passing an array by value doesn't prevent changes to the array elements.

```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Passing multidimensional arrays as arguments

You pass an initialized multidimensional array to a method in the same way that you pass a one-dimensional array.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

The following code shows a partial declaration of a print method that accepts a two-dimensional array as its argument.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

You can initialize and pass a new array in one step, as is shown in the following example:

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Example

In the following example, a two-dimensional array of integers is initialized and passed to the `Print2DArray` method. The method displays the elements of the array.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Element(0,0)=1
        Element(0,1)=2
        Element(1,0)=3
        Element(1,1)=4
        Element(2,0)=5
        Element(2,1)=6
        Element(3,0)=7
        Element(3,1)=8
    */
```

See also

- C# Programming Guide
- Arrays
- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays

Implicitly Typed Arrays (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly-typed variable also apply to implicitly-typed arrays. For more information, see Implicitly Typed Local Variables.

Implicitly-typed arrays are usually used in query expressions together with anonymous types and object and collection initializers.

The following examples show how to create an implicitly-typed array:

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

In the previous example, notice that with implicitly-typed arrays, no square brackets are used on the left side of the initialization statement. Note also that jagged arrays are initialized by using `new []` just like single-dimension arrays.

Implicitly-typed Arrays in Object Initializers

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, `contacts` is an implicitly-typed array of anonymous types, each of which contains an array named `PhoneNumbers`. Note that the `var` keyword is not used inside the object initializers.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

See also

- C# Programming Guide
- Implicitly Typed Local Variables
- Arrays
- Anonymous Types
- Object and Collection Initializers
- var
- LINQ Query Expressions