

.NET/C# Syntax.

Modern .NET/C# Course

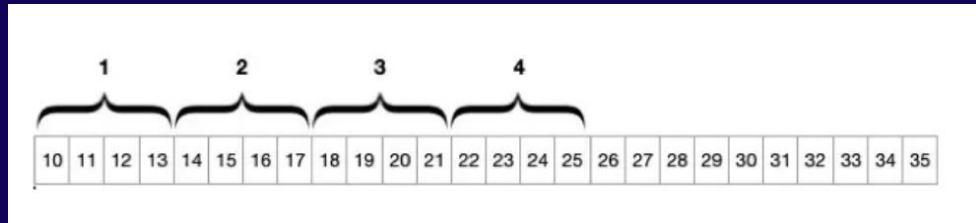
.NET Academy

Content

- Arrays.
 - Inline arrays.
- Loops.
 - For.
 - Foreach.
 - While.
 - Do-While.

Array.

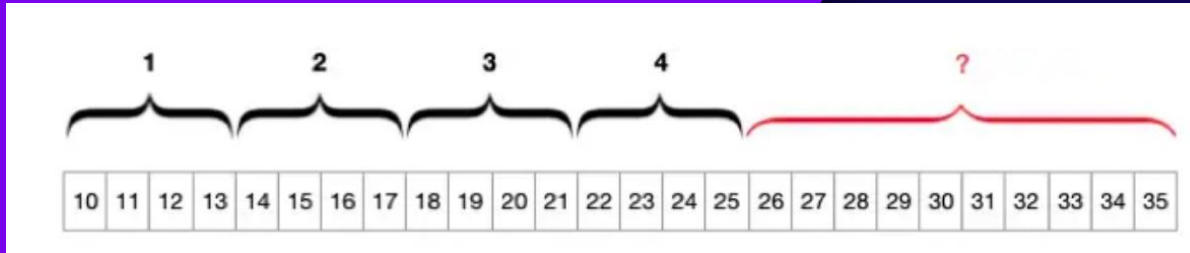
With this line of code, we created a static array that contains the four integers, but what exactly happened? The compiler will see that we need to create an array of integers and for this it needs to ask for memory, but how much? If we check integer size `sizeof(array[0]);` we will get 4 bytes, so to store our array we need 16 bytes $4 * 4 = 16$. Computer memory is organized into memory cells, each storing 8 bits and has an index number. One byte equals 8 bits, so each item will use 4 memory cells or 32 bits. We can see the full picture on the image.



Array.

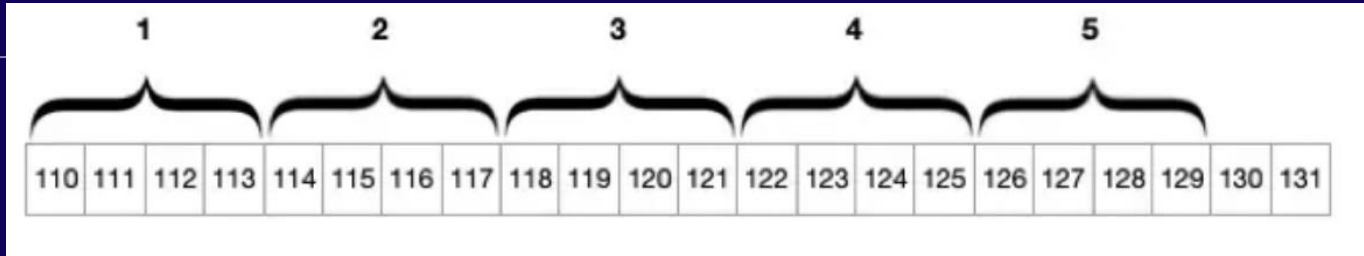
We see on the image some computer memory with a 25 memory cells. Now we can see that our array took 16 memory cells or 128 bits.

A push operation means to add a new element to the end of an array. The question is, is it possible to add an element to the array at all? And the answer, No. What if I want to add another integer 5? Of course, you might think “what’s the problem?, I can just add another integer and it will take cells 26–29”. But the answer will still be No you can’t. To understand why not, we need to go back to our array and memory cells.



Array.

The problem with adding additional values to an array is that our array was allocated 128 bits or 16 memory cells and because we can't manage where the memory object will be saved, we can't be sure that cells 26–29 are free. Maybe earlier, another array was stored starting from cell 26, or a 1 byte char was stored in that cell. The solution for this problem, is to take the length of the current array (in our case 4) and increase it by 1. An array of length 5 requires 20 bytes. We must ask for 20 bytes for a new array, copy the values from the old array and insert our new value 5 into the last cell. Now our array occupies 20 memory cells or 160 bits, and now the memory may look like this:



Array.

In the programming world, this kind of array is called a dynamic array, and it involves a concept known as Growth Factor. For example, let's use a growth factor of 1.5. If we have an array of 4 items (128 bits) and we push a new element to the end, the new array would be 5 items long (192 bits). $4 * 1.5 = 6$ $6 * 4 * 8 = 192$ 24 memory cells or 192 bits.



.NET Academy

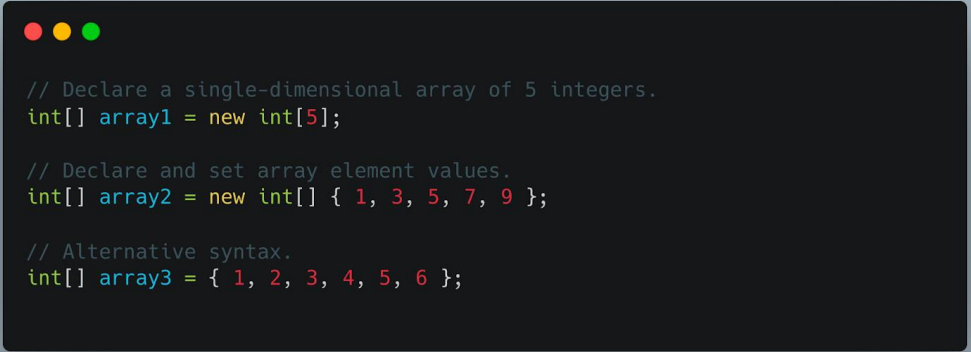
Array.

A pop operation removes the last element from an array. Let's back to our array `int array[4] = {1,2,3,4};`, now we want to pop the last item, in this case 4. For popping elements, you can find many solutions. For example, you can allocate memory for a new array (1 shorter than the original) and copy all elements except the last. But let's imagine popping from an array with 1000 elements. We would need to allocate memory for a new array and copy 999 elements, a costly task. To simplify this task we can create a buffer zone.

Okay, now we have talked about array, and how does it work in memory. Let's see how we can define arrays in C#.

Array.

We can store multiple variables of the same type in an array data structure. We declare an array by specifying the type of its elements. If we want the array to store elements of any type, we can specify object as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object. Let's see an example:



```
// Declare a single-dimensional array of 5 integers.  
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 };
```


Inline Arrays.

Inline arrays enable you to create an array of fixed size in a struct type. Such a struct, with an inline buffer, should provide performance comparable to an unsafe fixed size buffer.

Inline arrays are mosly to be used by the runtime team and some library authors to improve performance in certain scenarios. You likely won't declare your own inline arrays, but you will use them transparently when they are exposed as `Span<T>` or `ReadOnlySpan<T>` objects by the runtime.

Inline Arrays.

You can declare an inline array by creating a struct and wrapping it with the InlineArray attribute, which takes in the array length as a parameter in the constructor.


```
[System.Runtime.CompilerServices.InlineArray(10)]  
public struct MyInlineArray  
{  
    private int _element;  
}
```

Note: the name of the private member is irrelevant. You can use private int `_abracadabra`; if you wish. What matters is the type, as that decides the type of your array.

Inline Arrays.

You can use an inline array similar to any other array, but with some small differences. Let's take an example:

First thing to note is that during the initialization, we do not specify the size. Inline arrays are fixed size and their length is defined through the `InlineArray` attribute that's applied to the struct. Besides that, everything looks as it would if you were using a normal array, but there's actually more.



```
MyInlineArray arr = new();


for (int i = 0; i < 10; i++)
{
    arr[i] = i;
}

foreach (int item in arr)
{
    Console.WriteLine(item);
}
```

Inline Arrays.

Some of you might have noticed that in the for loop above we iterated until 10 instead of `arr.Length` – and that is because inline arrays don't have a `Length` property exposed like normal arrays do.

In a similar fashion, you can also use the spread operator in combination with inline arrays.



```
int[] m = [1, 2, 3, ..arr];
```

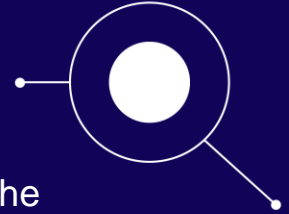
Loops.



Looping in a programming language is a way to execute a statement or a set of statements multiple times depending on the result of the condition to be evaluated to execute statements. The result condition should be true to execute statements within loops.

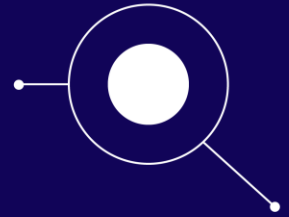
- **while** loop The test condition is given in the beginning of the loop and all statements are executed till the given boolean condition satisfies when the condition becomes false, the control will be out from the while loop.
- **for** loop has similar functionality as while loop but with different syntax. for loops are preferred when the number of times loop statements are to be executed is known beforehand. The loop variable initialization, condition to be tested, and increment/decrement of the loop variable is done in one line in for loop thereby providing a shorter, easy to debug structure of looping.
- **do-while** loop is similar to while loop with the only difference that it checks the condition after executing the statements, i.e it will execute the loop body one time for sure because it checks the condition after executing the statements.
- The loops in which the test condition does not evaluate false ever tend to execute statements forever until an external force is used to end it and thus they are known as **infinite loops**.

Questions for the study of the topic



1. How does memory allocation work when you create an array in C#? Explain the concept of memory cells and their role in storing array elements.
2. When you push a new element to the end of a dynamic array, what happens in terms of memory allocation, and how does the growth factor affect the new array's size?
3. What is the purpose of a buffer zone when popping elements from an array, and how does it simplify the process?
4. What is an inline array in C# and what purpose does it serve in certain scenarios?
5. How does the initialization of an inline array differ from a regular array, and why is there no Length property exposed for inline arrays?
6. What is the purpose of looping in a programming language, and under what condition are statements executed within loops?
7. What distinguishes the do-while loop from the while loop, and how does the order of condition evaluation affect its behavior?

Homework



1. What is the difference between a static array and a dynamic array, and how does the concept of a growth factor relate to dynamic arrays?
2. Describe the process of extracting an element from an array and explain why it can be complicated when working with large arrays.
3. How is an inline array declared and what attribute is used to determine its length?
4. Compare and contrast the while loop and the for loop in terms of syntax and typical use cases.
5. What is an infinite loop in programming and how to get out of it?

Thank you for your attention.

.NET Academy