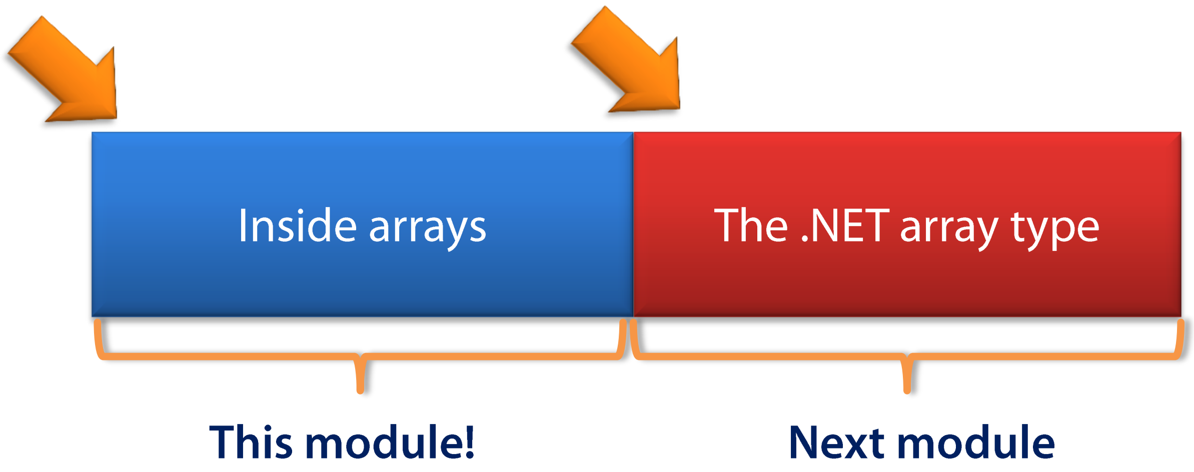
Inside Arrays

Simon Robinson
http://TechieSimon.com
@TechieSimon

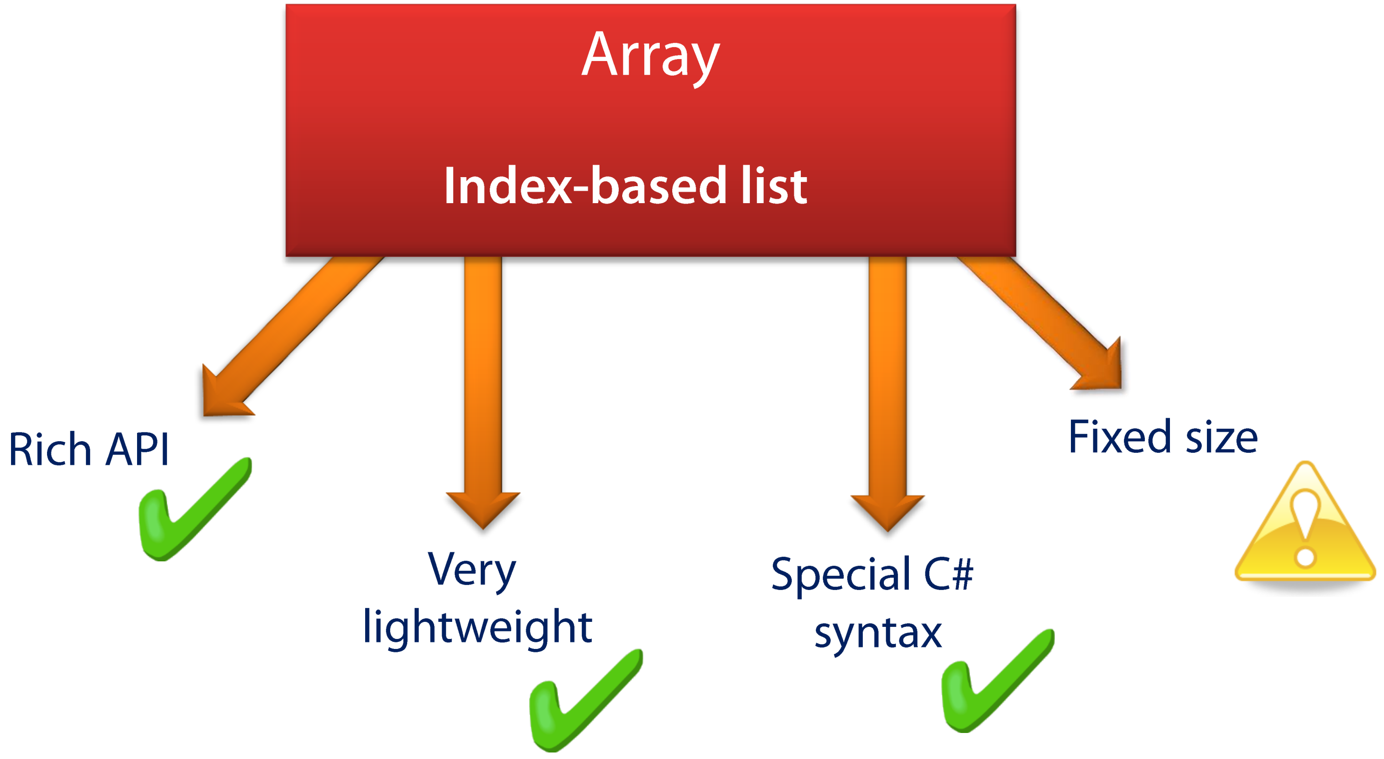
Arrays



Module Overview

- **What is an array**
  - Basic syntax
- **Arrays under the hood**
  - Element access is very efficient
- **Declaring and initializing arrays**
- **Enumerating (iterating) array contents**
  - foreach loop
  - for loop

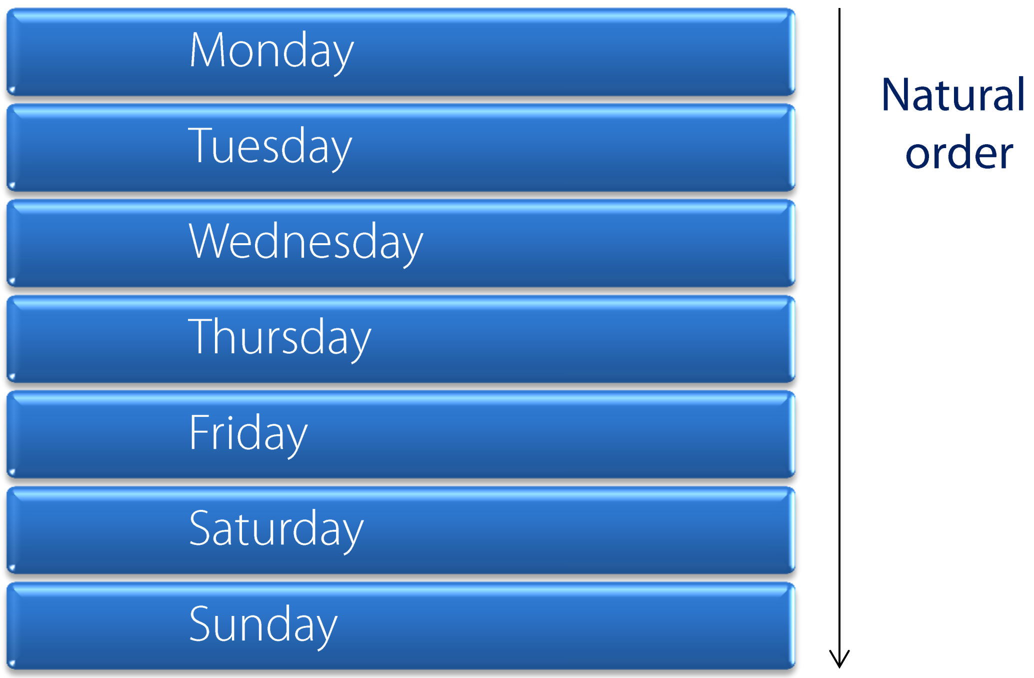
What are Arrays



Array

To store the names of the days of the week….

Fixed size = 7



Array

To store the names of the days of the week….

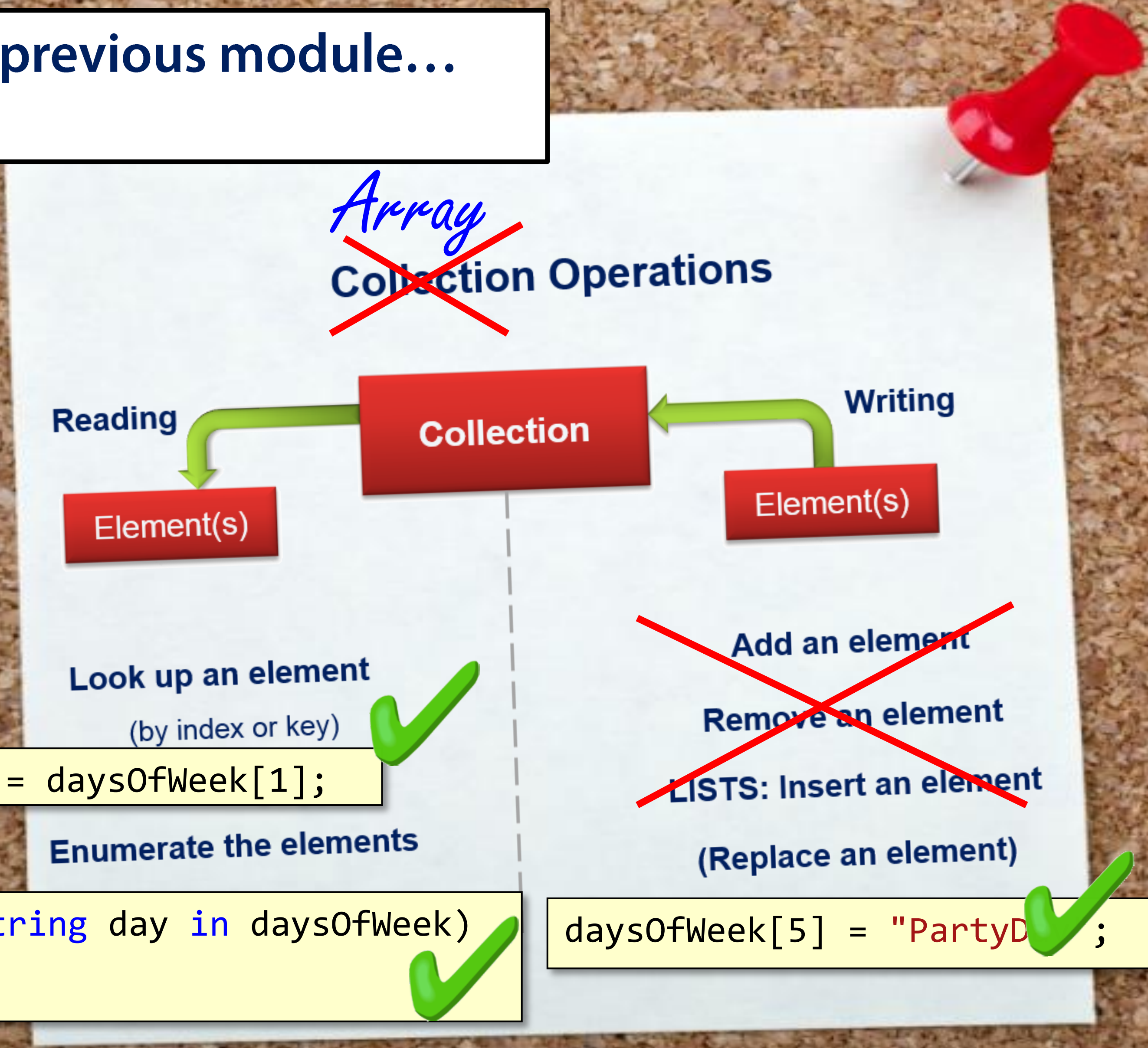
Fixed size = 7

- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- Saturday
- Sunday

Natural order

This is the kind of data arrays are great for

From the previous module…



Array

~~Collection~~ Operations

Look up an element

(by index or key) ✓

```
string day = daysOfWeek[1];
```

Enumerate the elements

```
foreach (string day in daysOfWeek)
{
```

✓

~~Add an element~~

~~Remove an element~~

~~LISTS: Insert an element~~

(Replace an element) ✓

```
daysOfWeek[5] = "PartyD    ;
```

Arrays under the Hood

Arrays

Special syntax in C#:

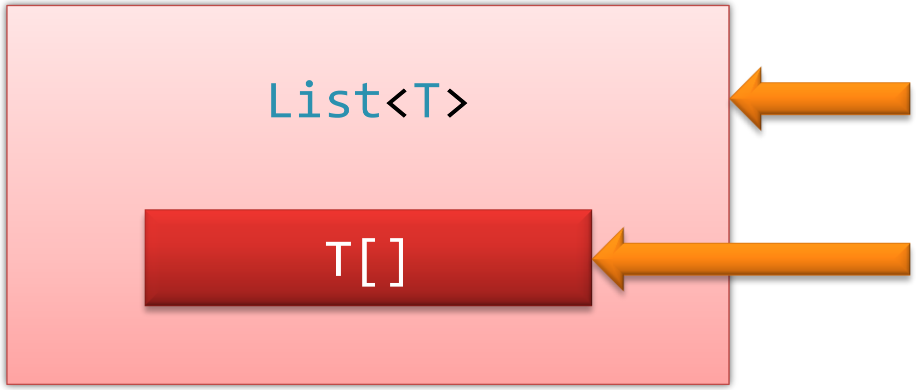
```
int[] iHaveSquareBrackets;
```

Implemented inside the CLR itself

Other Collections

Implemented using generics (eg. in C#)

Mostly implemented using arrays!



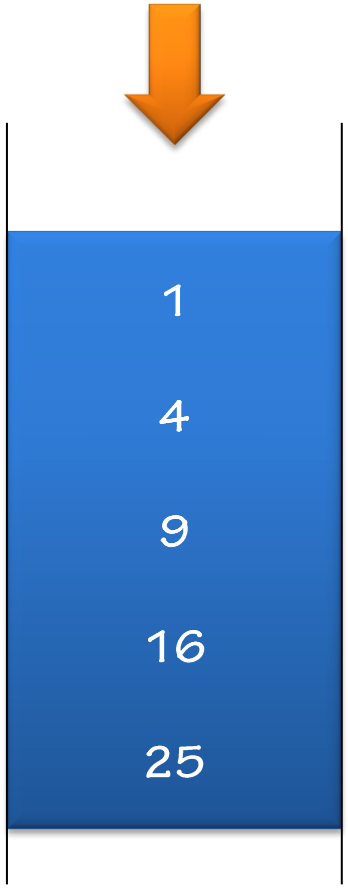
Can implement in managed code (eg. C#)

Implemented by .NET execution engine

Array of 5 integers:

```
int[] squares = {1, 4, 9, 16, 25};
```

Single memory block

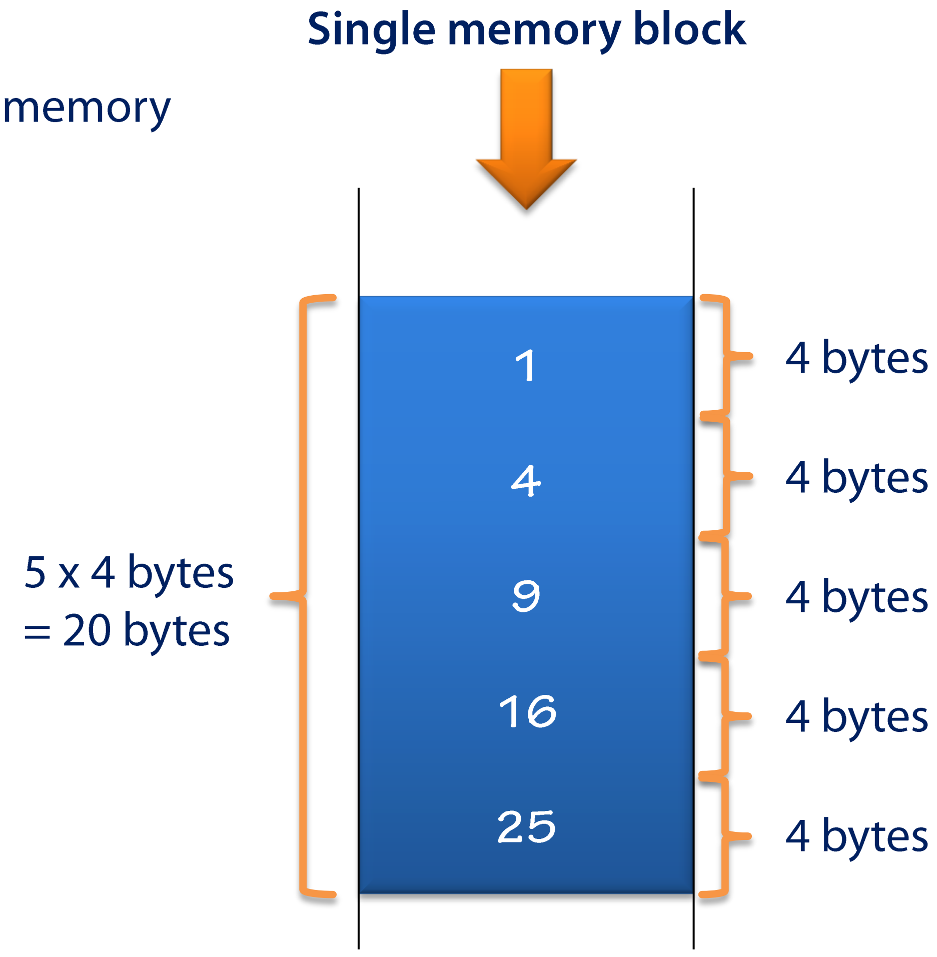


Elements go
one after the other
in the block of memory

Array of 5 integers:

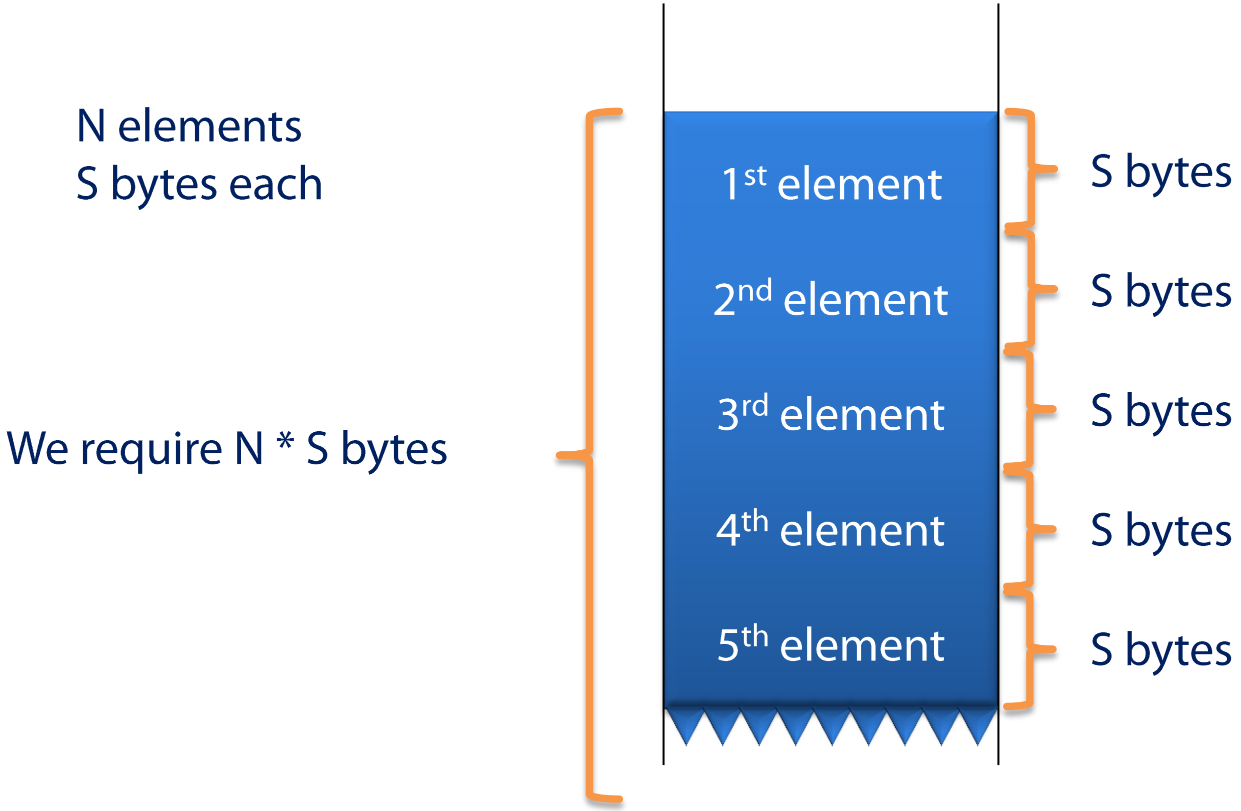
```
int[] squares = {1, 4, 9, 16, 25};
```

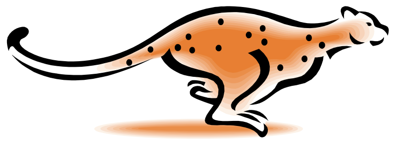
One int requires 4 bytes of memory



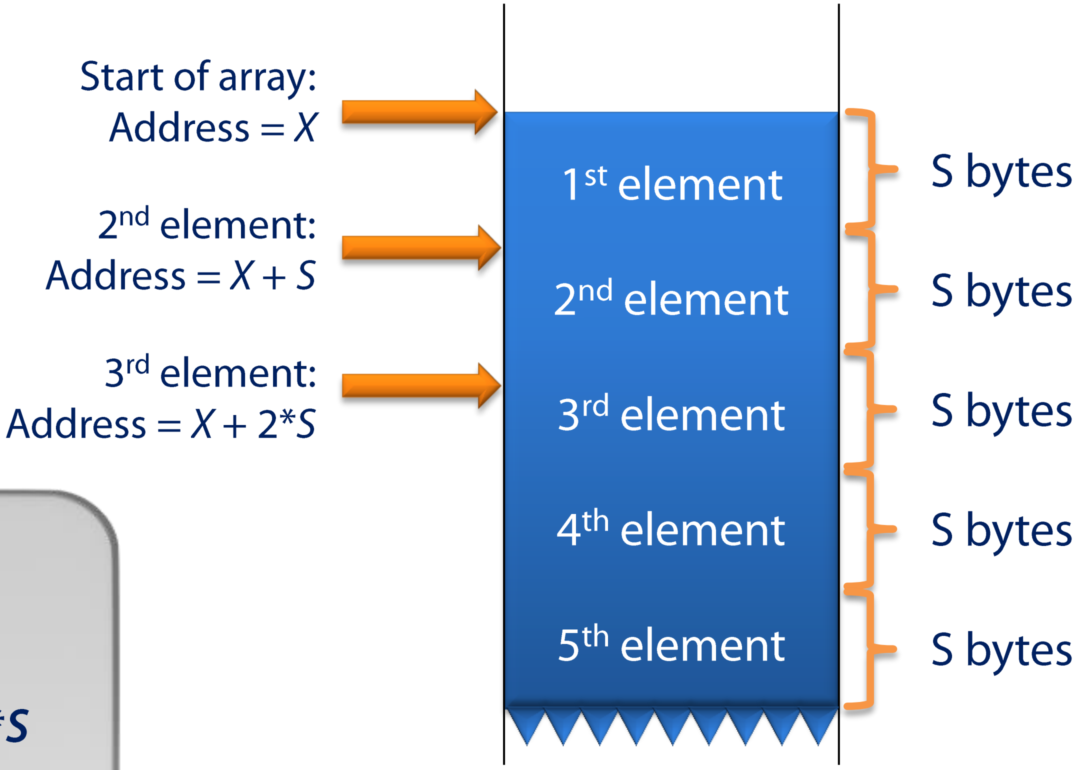
More generally:

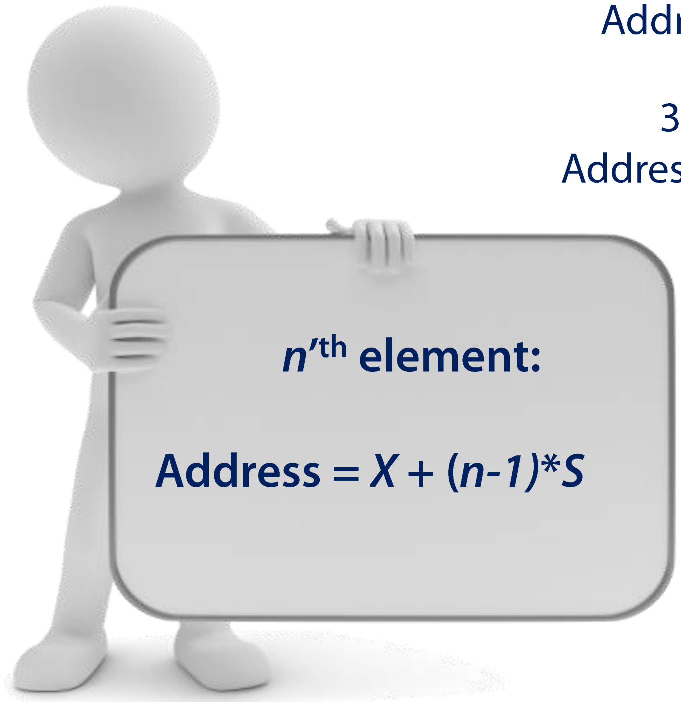
Array of some type that occupies S bytes:





Looking up elements is fast!





***n*'th element:**

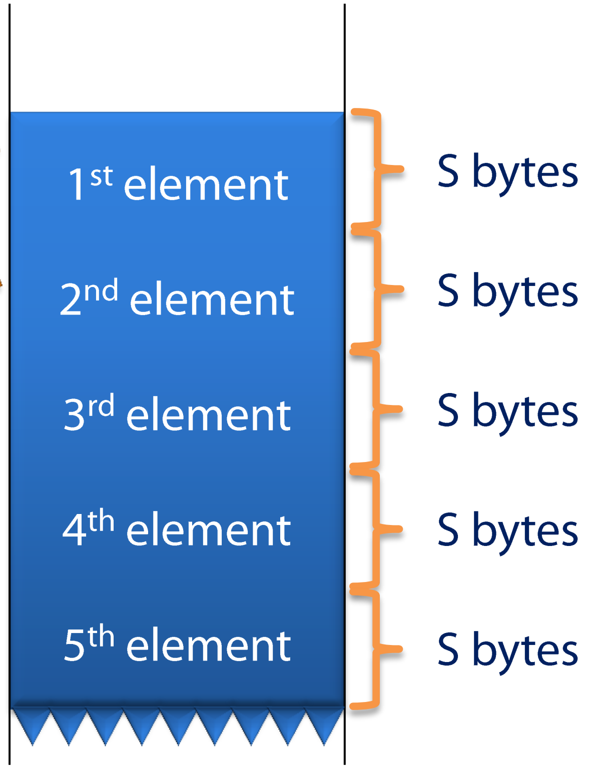
Address = $X + (n-1)*S$

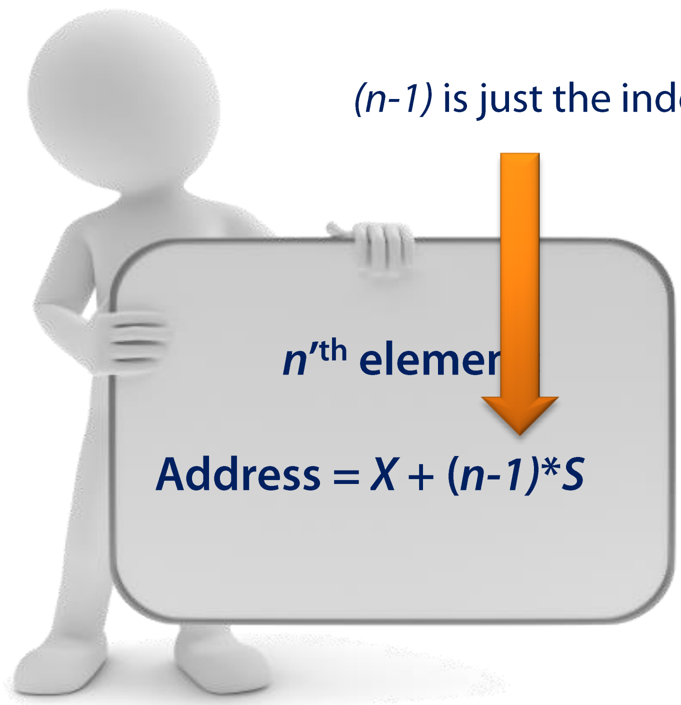
The index = the number of elements to jump over!

1st element is `array[0]`
- don't jump over any elements

2nd element is `array[1]`
- jump over 1 element

(n-1) is just the index!



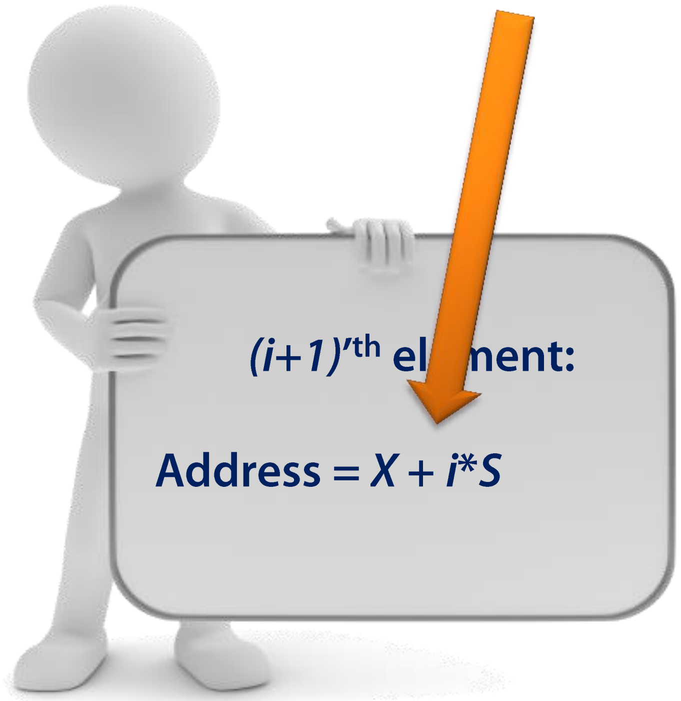


***n*'th eleme**

Address = *X* + (*n*-1)*S

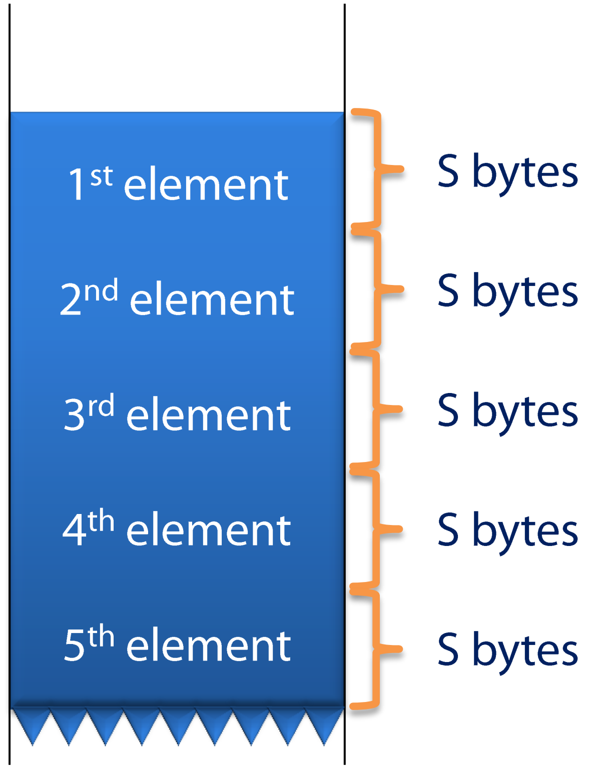
The index = the number of elements to jump over!

So to look up `array[i]`:



(i+1)'th el ment:

Address = $X + i*S$



In C++:

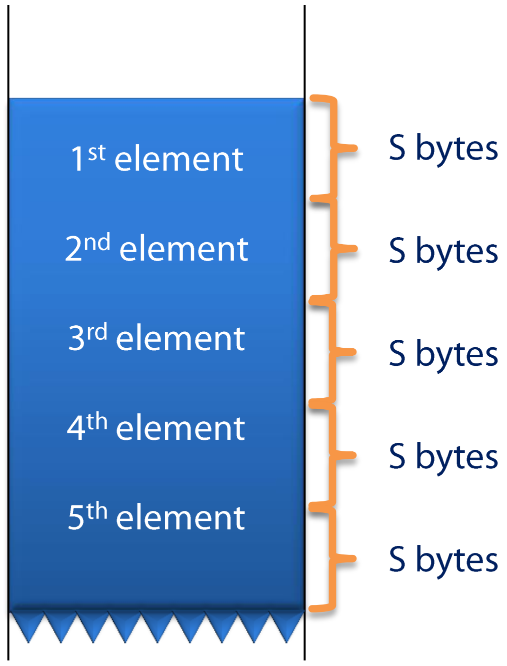
```
myArray[i]
```

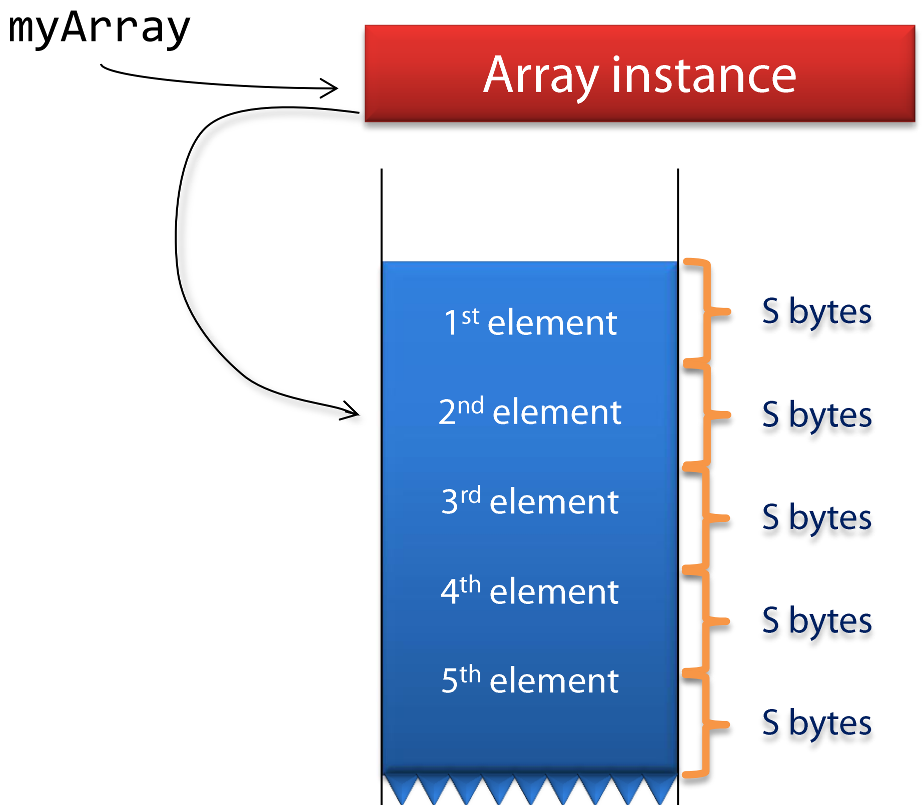
Literally means
(address of `myArray`) + $i * S$

In C#:

```
myArray[i]
```

`myArray` refers to a managed array object.
Extra indirection through that object

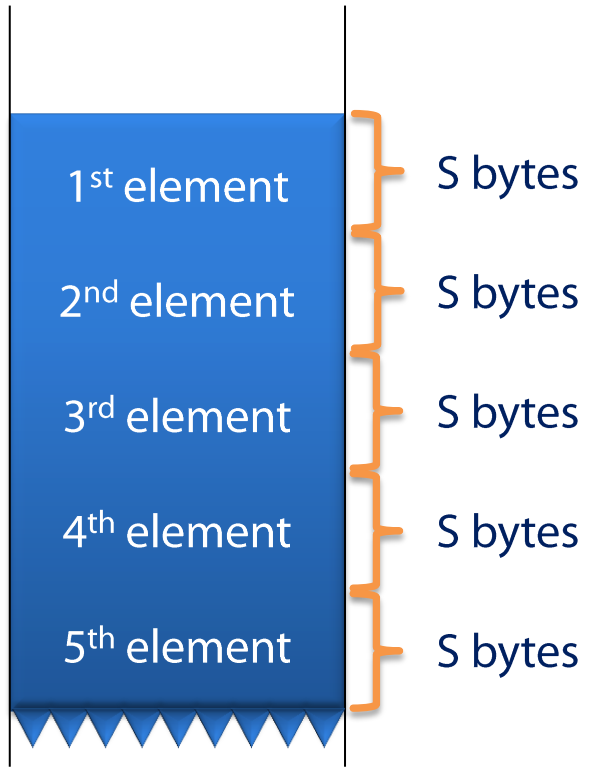




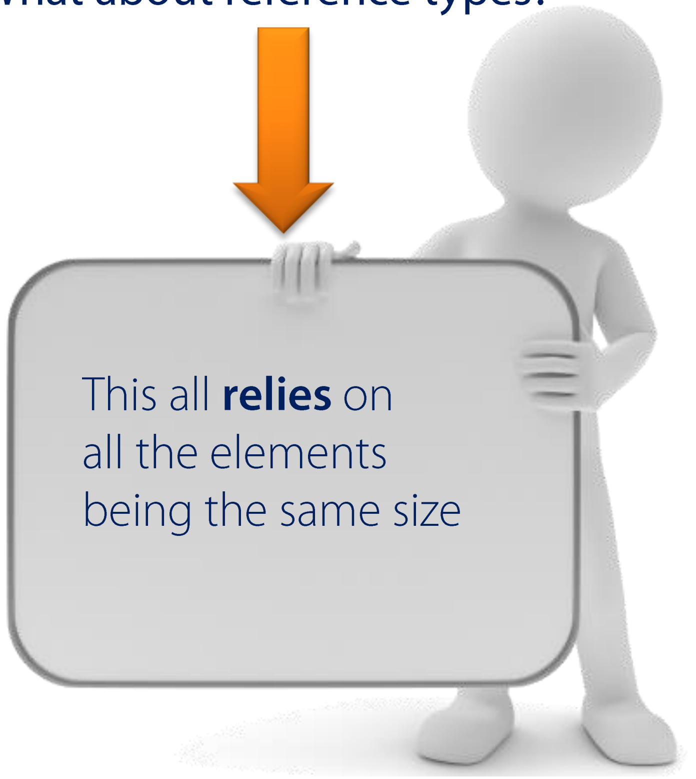
Arrays are:

Very simple to implement ✓

Very efficient for looking up elements ✓



This is true for a value type.
What about reference types?



Arrays of Reference Types

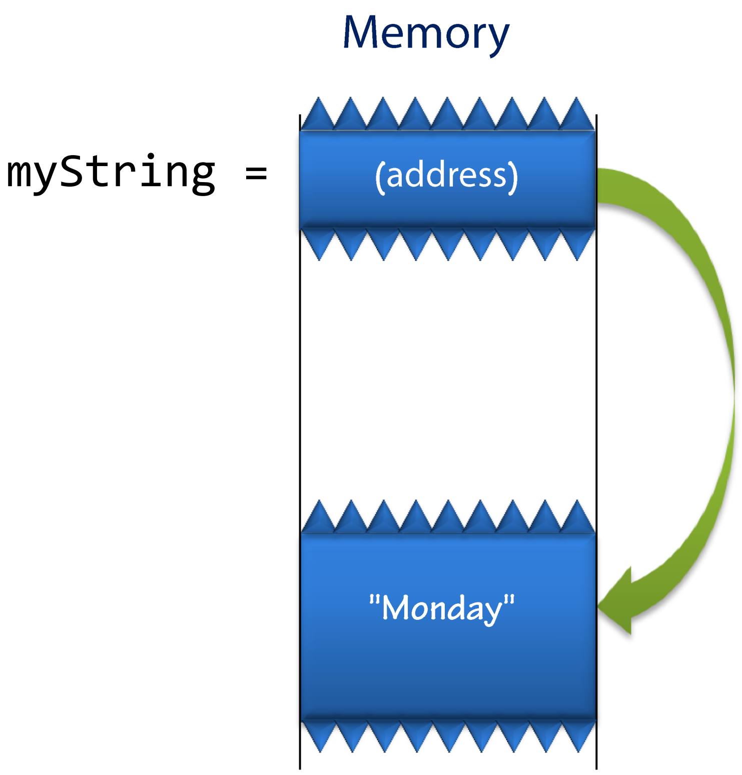
Example: Days of the week

```
string[] daysOfWeek = {
                   "Monday",
                   "Tuesday",
                   "Wednesday",
                   "Thursday",
                   "Friday",
                   "Saturday",
                   "Sunday" };
```

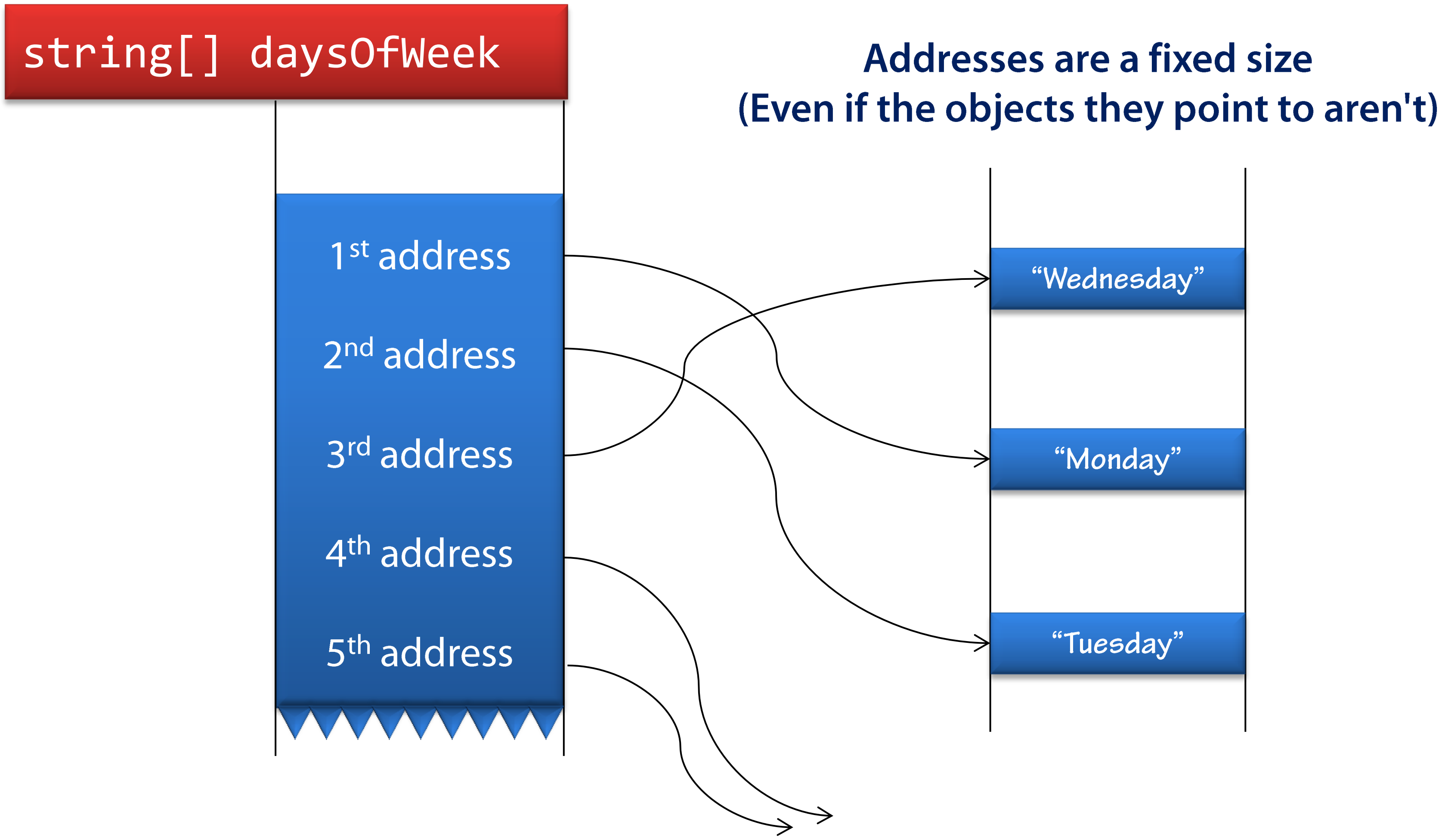
`string` is a reference type

Variable stores address of actual data

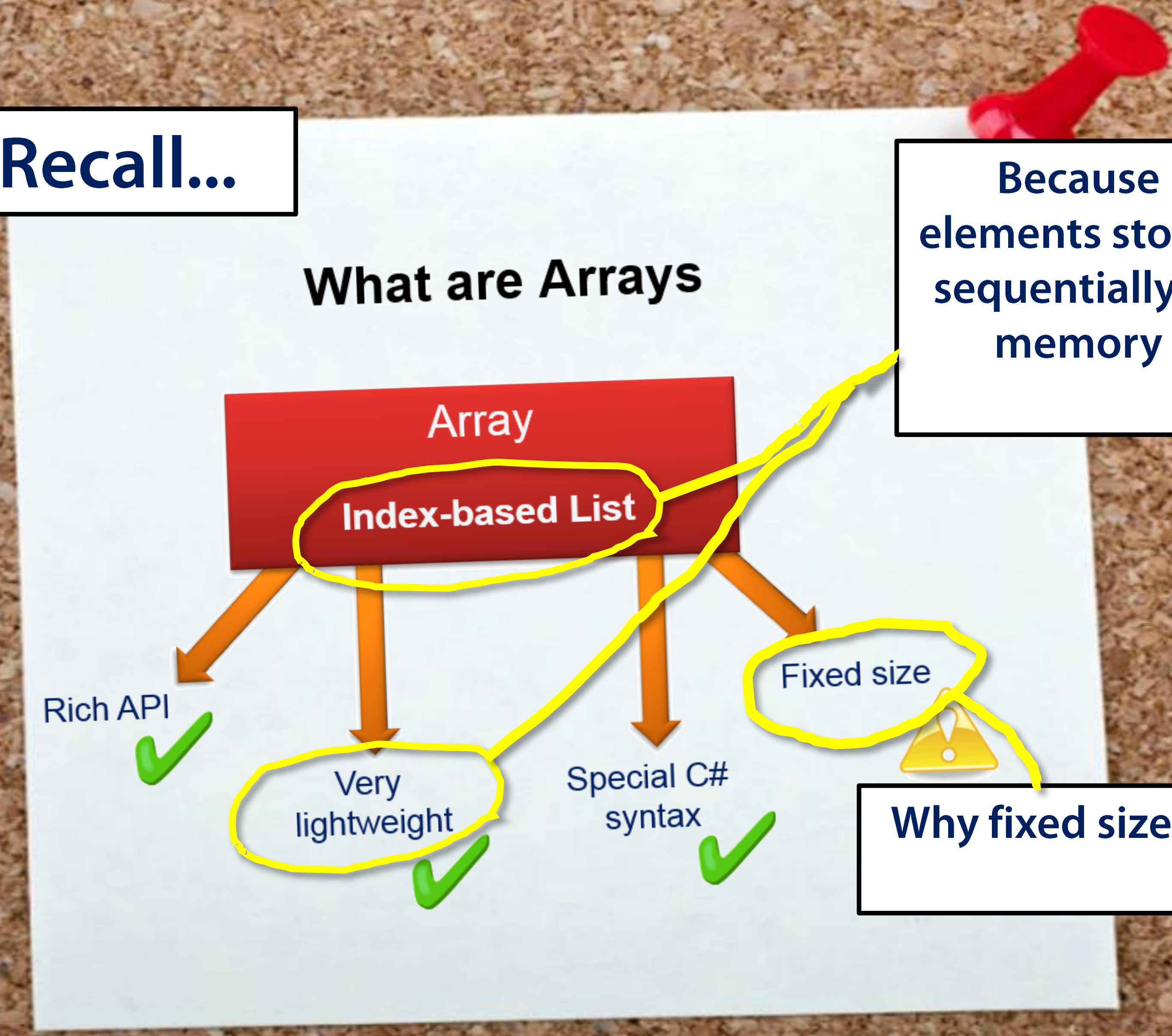
```
string myString = "Monday";
```



Array of strings



Recall...

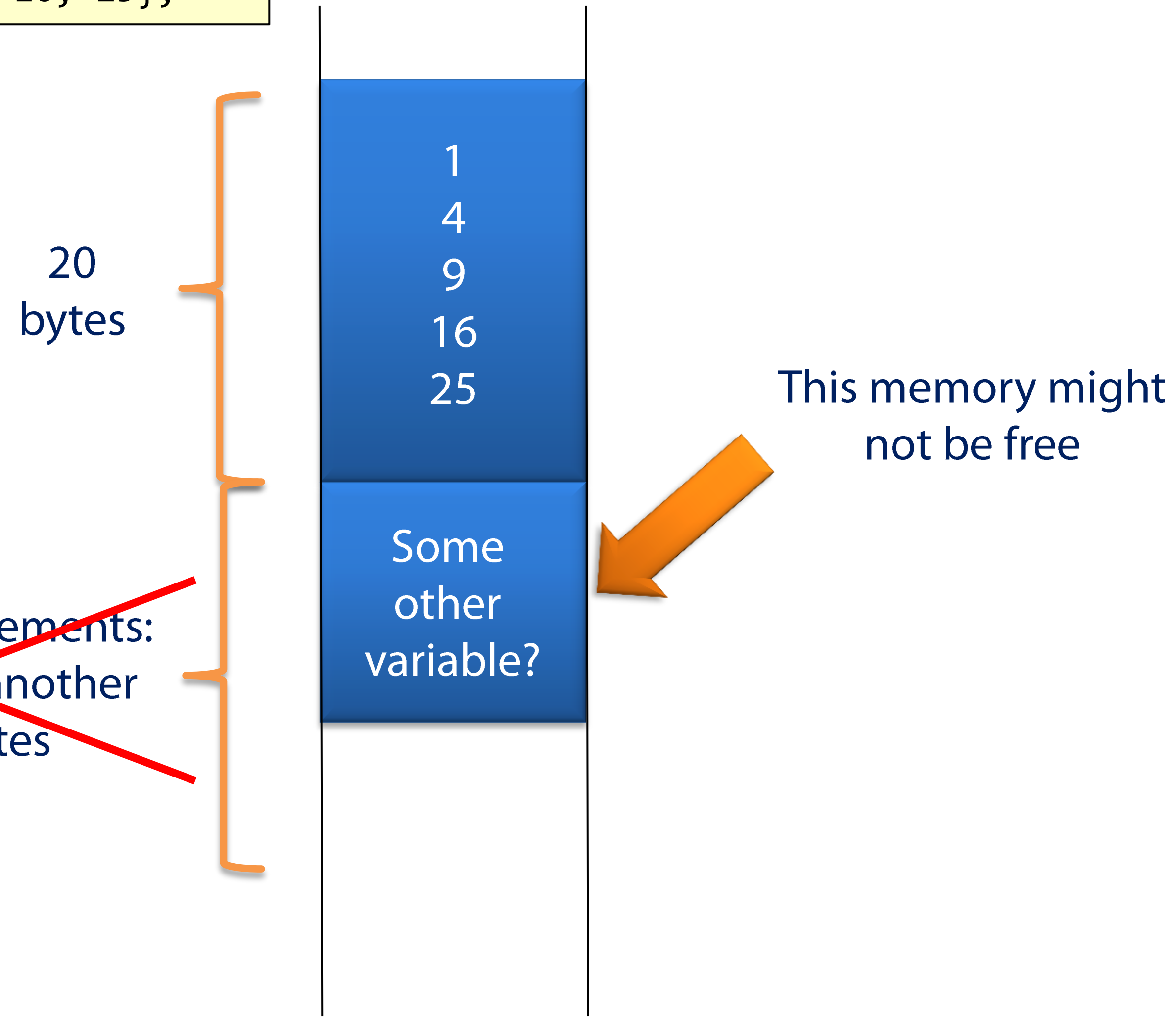


Because elements stored sequentially in memory

Why fixed size?

Suppose you wanted to add elements to an array

```
int[] squares = {1, 4, 9, 16, 25};
```



This memory might not be free

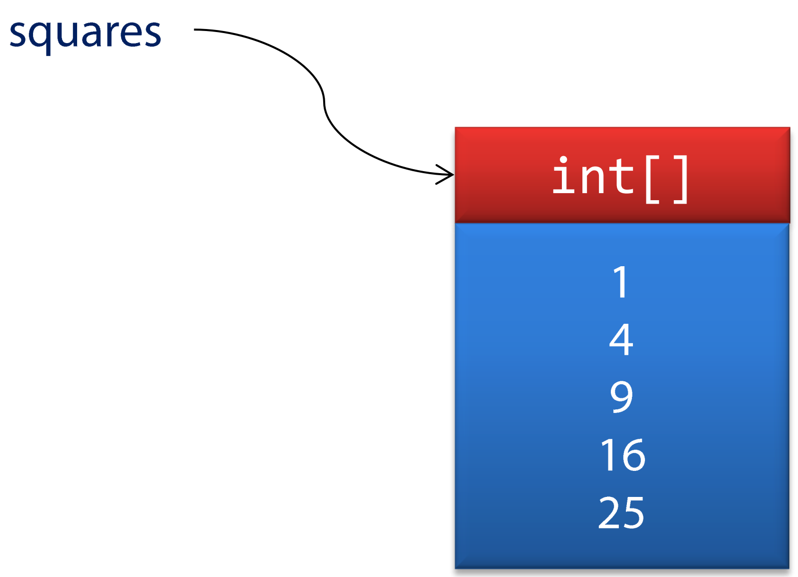
Can't extend the memory block

~~To add 5 elements: We need another 20 bytes~~

Array Initializers

You can use any expression that can be evaluated at run-time!

```
int eight = 8;
int[] squares = new int[] {
    1,
    2 * 2,
    eight + 1,
    int.Parse("16"),
    (int)Math.Sqrt(625)
};
```



Array Initializers

```
int eight = 8;
int[] squares = new int[] {
    1,
    2 * 2,
    eight + 1,
    int.Parse("16"),
    (int)Math.Sqrt(625)
};
```

The initializer is not a constructor!

Array Initializers

Compiler turns this...

```
int eight = 8;
int[] squares = new int[] {
    1,
    2 * 2,
    eight + 1,
    int.Parse("16"),
    (int)Math.Sqrt(625)
};
```

...into (roughly) this

```
int eight = 8;
int[] x5 = new int[5];
x5[0] = 1;
x5[1] = 2*2;
x5[2] = eight + 1;
x5[3] = int.Parse("16");
x5[4] = (int)Math.Sqrt(625);
```

These values are known at compile time…

```
int[] x5 = new int[5] { 1, 4, 9, 16, 25 };
```

… so the compiler can do (roughly) this

```
int[] x5 = new int[5];
// Some magic to set up handle - commented out
System.Runtime.CompilerServices.RuntimeHelpers.
   InitializeArray(x5, handle);
```

Very efficient!

Enumerating an Array

foreach **loop**

```
foreach (string day in daysOfWeek)
{
```

Explicitly request each element

for **loop**

Code Demo

Summary

- **Arrays are fixed size and ordered**
- **Elements are stored sequentially in memory**
  - Element access is very efficient
- **Syntaxes to construct array:**
  - Array initializers to initialize elements
- **To enumerate (iterate) array contents**
  - foreach loop
  - direct element access (typically in a for loop)
- **foreach loops give read-only access**