Arrays, Objects and Methods

Table Of Contents

Arrays

An array is a contiguous block of memory containing multiple values of the same type (usually!).

When an array is initialised the array will be allocated and will return the address of where the array is stored.

These data types are also useful for avoiding issues such as this.

```
int variable01 = 32;
int variable02 = 98;
// ...
int variable87 = 23;
```

Arrays

Within the C# language we are able to intiailise an array in a few different ways. Most commonly and what is typically used is allocate and specify size.

```
int[] numbers = new int[16];
```

Although we can also initialise it with values defined.

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

The above segment will define an array of length 4 containing the values 1, 2, 3 and 4

Arrays

In addition, there is some other more specific quirks with arrays with C#.

```
int[] numbers = [1, 2, 3, 4];
```

Similar to the static initialisation from before but now with **square brackets**.

Arrays - Layout

Let's consider the layout of an array and visualise it. Since it is a **contiguous block** of memory, we are able retrieve values from the following array using the indexes 0 to 4.

```
double[] amounts = [1.50, 2.75, 10.59, 8.99, 7.65];
```

Let's visualise this array as a table.

Index	**0**	**1**	**2**	**3**	**4**
Value	1.50	2.75	10.59	8.99	7.65

Given the five values, we are able to access the first values using **index** 0 which is at the base address of the array.

Arrays - Layout

A more simplified version, is to just have blocks containing values. However, we will make the assumption that the base address of the array `numbs` is at `0x1000` (4096 as a decimal value).

```
int[] numbers = [23, 40, 56, 68, 21, 99];
```

We will have the offset from the base address above. We need to factor in one more concept which is the size of the datatype. For an `int`, the size is 4 bytes, although this assumption **can be** broken in javascript and python.

+0	+4	+8	+12	+16	+20
23	40	56	68	21	99

Arrays - Layout

Changing the datatype will impact the offset as well. Assuming our base address is still `0x1000` but the data type is a `long`, the amount we need to shift to get to the next number is now doubled.

```
long[] numbers = [23, 40, 56, 68, 21, 99];
```

Since `long`, is 8 bytes instead of `int` which is 4 bytes, the offsets are going to utilise this size multiplied by the index.

+0	+8	+16	+24	+28	+32
23	40	56	68	21	99

Demo - Arrays Demo

Multi-Dimensional Arrays

We are not limited just creating single dimensional arrays. We are able to create multi-dimensional arrays.

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

There are two types, one adheres to a matrix-like structure and the other is commonly referred to as a jagged array.

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

We need to be mindful of what this looks like though when assigning and using these objects.

Multi-Dimensional Arrays

With the following code:

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

It will initialise the 2 elements within the array to null. This allows us to set each element to a separate array.

```
array[0] = new int[4];
array[1] = new int[8];
```

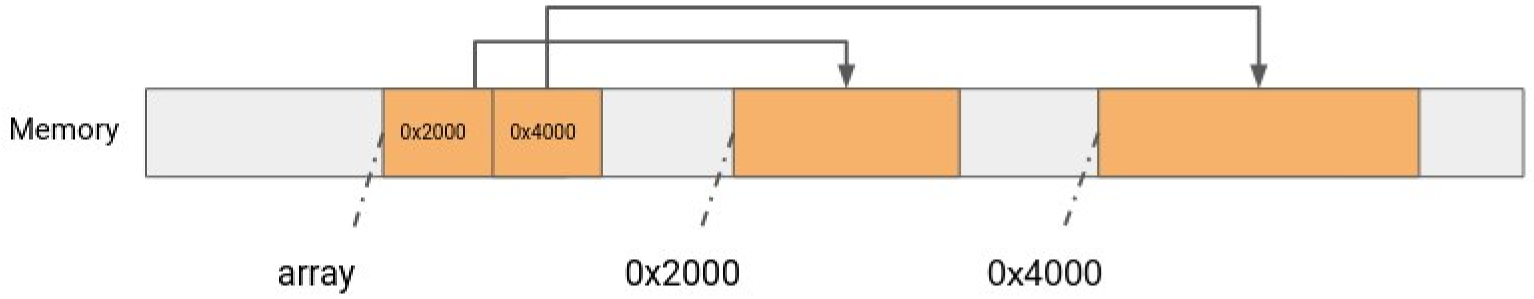
Now, this can seem a little heavy because we will link back to how arrays are calculated and traversed.

Multi-Dimensional Arrays - Layout

Given the `array` initialisation below, we will visualise how it can be mapped in memory.

```java
int[][] array = new int[2][];
array[0] = new int[4];
array[1] = new int[8];
```

Given the new allocations as outlined, these can be at different offsets as determined by the allocator. We have just used some arbitrary values here for simplicity.



Values 0x2000 and 0x4000 are used for demonstration purposes and are not the actual addresses.

Multi-Dimensional Arrays - Traversal

Traversing a single array is a fairly straight forward process. Multi-Dimensional ones usually require some more specifics around the domain in which they are used in but also conform to certain patterns.

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

int counter = 1;

for(int i = 0; i < array.Length; i++) {
    for(int j = 0; j < array[i].Length; j++) {
        array[i][j] = counter;
        counter++;
    }
}
```

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

int counter = 1;

for(int i = 0; i < array.Length; i++) {
    for(int j = 0; j < array[i].Length; j++) {
        array[i][j] = counter;
        counter++;
    }
}
```

- Counter is increment every time we iterate in the inner loop.
- We should have in total 9 positions
- Outer loop is the first dimension and the inner loop is for the second dimension, these are both 5 for the sake of convenience however they can be different.

Multi-Dimensional Arrays - Traversal

Let's look at just printing out the values and seeing what we are able to get from this.

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

// Assuming we have assigned it using counter

for(int i = 0; i < array.Length; i++) {
    for(int j = 0; j < array[i].Length; j++) {
        Console.WriteLine(array[i][j]);
    }
    Console.WriteLine("");
}
```

Multi-Dimenional Arrays - Traversal

However we don't have to assign each element like we have done before. We can change this up a bit. Consider what the order will be if we were to assign it like so.

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

int counter = 1;

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

    int innerLength = array[i].Length;

    for(int j = 0; j < innerLength; j++) {
        array[i][innerLenght - j - 1] = counter;
        counter++;
    }
}
```

Demo - Multi-Dimensional Arrays

Arrays - Use cases

For both single dimensional arrays or mutli-dimensional arrays there is a lot of different scenarios in where you will use this.

For example:

- Image Resizing
- Image Filters (Blurring, Re-colouring, Indexing)
- Edge Detection in Images
- Animating Entities
- Nearest Pair Of Points

All have different schemes and you will need keep in mind how a domain specific formula will dictate accesses.

Take a break!

Static Methods

A method is a stored set of instructions bound to an object. In the case of a static method, the object is the class which it is defined in.

You have seen a method already which would be the `Main` method but we will be able to start creating our own.

```
public static int addThree(int a, int b, int c)
{
    return a + b + c;
}
```

Static Methods - Breakdown

You can use the following pattern to understand how methods are written. Within the software-industry, these get called `functions`, however in the context of C# and other programming languages, there is a difference.

Declaration pattern.

```
static return_type name ([parameters])
```

Static Methods - Breakdown

```
static return_type name ([parameters])
```

- `static` - Indicates that the method is associated with the class and not an **instance**. We will look into **instance methods** later on.
- `return_type` - Refers to the datatype that is returned or `void` if it does not return anything.
- `name` - Name of the method itself, like `addThree` aka as the identifier.
- `paramters` - There can be 0 to many paramters specified here. You will need to specify the datatype and the name of the parameter.

Static Methods - Return Types

C# an use any primitive or reference type as a return type. The compiler will check and ensure that any assignment to the return value of a method is correct.

There is a special return type that you may or may not have encountered `void`.

`void` does not return any value and any void method is typically used for manipulating passed data or output.

Returning data from a method is generally used for querying or object creation.

Demo - Static Methods

Methods and Internals

As with most commonly used programming language, the runtime of these programs uses what is called a **stack machine**. In C#, this is called `VES` (Virtual Execution System) which also what we call our **virtual machine**.

The way a stack machine works is that we have the following components.

- Stack Frame - Holds data for executing the method, this will be your **variables that are declared inside the method**.
    - What is not seen are the **stack pointer** and **frame pointer**, a **stack pointer** is the base address of the stack frame and a **frame pointer** is the where the **return address**
    - **A return address** is the address of where the program should go after it has finished executing a method. In short **return** is an end of method construct but also will be used as a jumping point to the previously called method.

Methods and Internals

- Call Stack - Holds all the stack frames, when we execute a program, we are able to call a number of methods (or functions) which will be executed. When a method calls another, the **caller** still needs have **alive** and not replaced by the **callee**.

Call Stacks are fixed sized, like an array and roughly **4-8MB** in size.

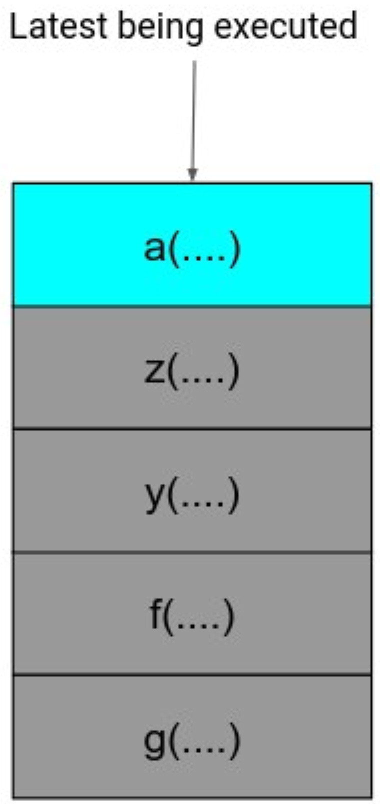
Running out of stack space will result in **StackOverflow** (You can probably see why the website called itself that now).

Methods and Internals

Let's visualise what a call-stack looks like now.

The method being executed at the top of the stack is the most recently called method. A method finishes executing once it has reached a return state or for void method, once it has reached the end of method scope.

- Each method that is executed gets their own **Stack Frame** which is then put on top of the stack. Once finished, it is popped off.



Demo - Method Calls, Call-Stack and Tasks

If we have some time, I will go through **Class** tonight as well.

Classes

There is a clear distinction between a primitive type and a reference type but how is the distinction made?

Reference Types are Classes

We have already used classes within our programs since the start of semester. However now we are able to define our own classes.

Most programming languages have some mechanism of structuring data for reuse.

Classes - But What are they?

> "A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviours. When the program is run, each object can act alone or interact with other objects to accomplish the program’s purpose."

Sometimes it is simply conveyed as a blueprint/template/concept of an **object**.

C# as an OOP language, primarily focuses on this way to structure data and code.

Classes

You have used classes in some form already and also with some basic usage of methods.

What we have missed is instantiating something that is more explicitly or not simply derived to be a class (like an array).

You have used classes like Console, String and Array.

Classes - Objects

The **type** of an object variable is its **class**.

```
Point p;
```

The class is `Point` which we will also call its **type** as well.

We can instantiate a class by using the `new` keyword and like so.

- `Point topleft = new Point(-1, -1);`
- `Point right = new Point(1, 0);`
- `Point home = new Point(-3388797, 15119390);`

Classes - `new` keyword

The `new` keyword does some amount of heavy lifting here. Whenever we use `new`, we are being explicit with allocating memory to hold this data.

- Since we have a point that holds at least two ints, we need to store that somewhere.

You will observe it is used in conjunction with the class name itself. Which can seem confusing but this relates to the **constructor** which outlines how it is built.

Classes - Can I make my own?

Yes!

However let’s start off with a basic class definition.

```
class Cupcake {
  bool delicious = false;
  string name = "Chocolate Cupcake";
}
```

We can instantiate this class with the following line of code.

```
Cupcake c = new Cupcake();
```

Classes - Constructors

In the previous example, what was happening is that we have fields already initialised and usable, we are then able to use a **default constructor**.

Classes - Constructors

Extending our Cupcake class we can write our own constructor.

```
public class Cupcake {
    bool delicious = false;
    string name = "Chocolate Cupcake";

    public Cupcake() { /* NO OP */ }
}
```

While not much as changed, we now have some control over how it is constructed and we can start parameterising this **constructor**.

Demo - Constructors and we will expand more on this next week

References and Primitive Types

Arrays and Objects are reference types. As outlined in Index Calcuation, Arrays are assigned to a memory address, Objects are also assigned to a similar value or modelled where each field is an offset from the base address.

References and Primitive Types

As a refresher, some of the common C# primitive types are.

- int - Integer datatype, will represent numbers like 1, 2, -3, 50.
- bool - Represents true or false
- float - Floating point value, represents values like 20.5, 1.22, 9.67
- double - Similar to float but with more precision.
- byte - Integer value between -128 to 127.

However, `Array` and instances/objects are **reference types**.