MODULE 1: INTRODUCTION TO PROGRAMMING IN C#

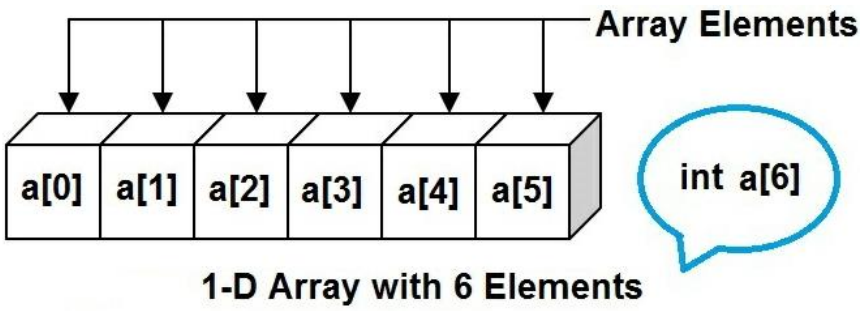
TUTORIAL 4: FURTHER DATA TYPES & PROGRAMMING TECHNIQUES

Introduction

In this tutorial you will learn about arrays, lists, and associated programming techniques. Importantly, you should gain some practical experience involving these concepts in typical computing applications. This topic forms an introduction to the broader field of 'data structures' which not only describes the data types available in a programming language, but introduces abstract concepts including lists, queues, stacks, trees and so on – you will meet these later in this course. For now, we shall look at a widely-used structure – the list – and see that in C#, it can be implemented as an array or as a designated list. Typical operations include 'search' and 'sort'. We shall look at methods for searching lists for a particular item and sorting data into numerical or alphabetical order. While there are built-in methods to perform some of these actions, it is important that we can implement our own versions as these facilities are not available in all programming languages. We shall also look at two-dimensional arrays that represent a grid of values and give an overview of typical operations.

Study Guidelines

An array in its simplest form is a collection of similar data with a common name to identify it. A typical example is a list of numbers each of type 'int', for example:



Note that we start numbering array elements from 0.

Within our program, we would declare and, as an option, intialise its contents e.g.

```
private void Form1_Load(object sender, EventArgs e)
{
    int[] Nums = { 7, 3, 67, 12, 2, 29 };
    for(int i=0; i<6; i++)
    {
        listBox1.Items.Add(Convert.ToString(Nums[i]));
    }
}
```

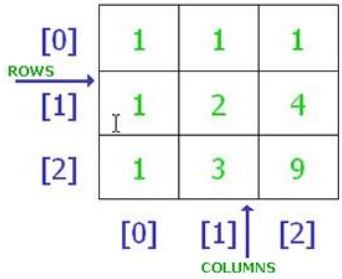
Note that if we were not going to set the initial vales of the array elements, then we declare the array size e.g.

```
String [] Names = new String [10];
```

Similarly, if we want to create an array of integers then we could use something like;

```
int[] Vals = new int[4];
```

We now introduce two-dimensional arrays which represent a grid containing data. Each element of the array is identified by its row and column position, as illustrated below:



Important Note: once again, the row and columns are numbered from 0 upwards. Therefore row 2, column 3 is indexed by (1,2) – the so-called 'N-1 rule'.

A situation where you might use a 2-D array is a simple '*noughts and crosses*' program where the user plays against the computer. Many other programs need to use 2-D arrays for a variety of reasons including mathematical utilities where these arrays correspond directly to matrices which are widely used e.g. for solving simultaneous equations. They can also be used in image processing to hold individual pixel values in a grid representing a rectangular picture.

To create a 2-D array in C#, you should use a declaration such as:

```
int[,] myArray = new int[2,3];
```

A 2-D array can be initialized as follows:

```
int[,] myArray = new int[,] { {1,2,3} {4,5,6} }; // { {first row} {second row} }
```

Additional Notes

2-D arrays typically involve nested for loops in a program e.g. to input the array values at run time and output the stored values, we might use the following code:

```
int nRows = 2, nCols = 3;
int[,] myArray = new int[nRows,nCols];
int r, c;

// Enter the values for the array
for (r = 0; r < nRows; r++)
{
    for(c=0; c < nCols; c++)
    {
        Console.Write("Enter a value for row {0}, col {1}: ", r+1, c+1);
        myArray[r, c] = Convert.ToInt32(Console.ReadLine());
    }
}
```

```
Console.WriteLine("The stored array is:");

// Output the values for the array
for (r = 0; r < nRows; r++)
{
    for (c = 0; c < nCols; c++)
    {
        Console.Write(myArray[r, c]+" ");
    }
    Console.WriteLine();
}
Console.ReadLine();
```

Note that we are not restricted to 1-D and 2-D arrays. We can create three-dimensional arrays which can represent a physical cube made up of blocks, each containing an item of data. C# also allows us to create arrays with even more dimensions e.g. a 5-dimensional array, although this hurts the mind a bit! Another facility is provided by the *jagged array* but we do not need this at the moment.

Operations on Lists

In previous tutorials we have used one-dimensional arrays to represent lists of items. We now want to take a further look at lists and we start with the foreach statement. An example is:

```
char[] text = { 'H', 'e', 'l', 'l', 'o' };
foreach (char i in text)
    Console.Write(i);
Console.ReadLine();
```

Two common operations associated with lists are *searching* and *sorting*:

Searching a list

We introduced searching in Tutorial 2, but we give a few more details here. The simplest search method is a 'sequential search' in which we compare each listed item to the search item starting with the first item in the list and moving to the next item repeatedly until we find a match. If we are looking for a single occurrence of the search item, we can terminate the search (e.g. by a 'break' statement in a for loop). Otherwise we can continue the search and report any further occurrences.

The following code is an example of searching a string array with N elements containing a list of names (previously defined):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleSearch
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            String[] Names = { "Fred", "Barry", "Gill", "Harry", "Mary", "Julie" };
            String searchName = "Harry";
            for (int i = 0; i < 6; i++)
            {
                if (Names[i] == searchName)
                {
                    Console.WriteLine(Names[i]);
                }
            }
        }
    }
}
```

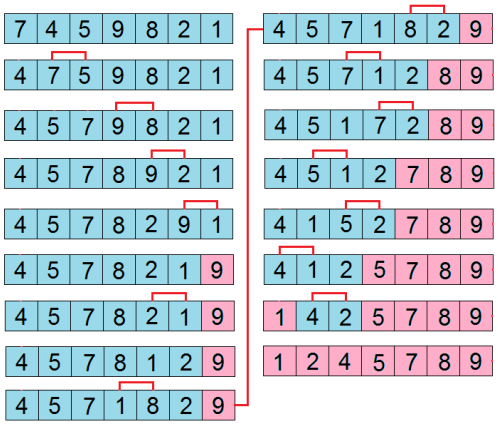
If we wish to rapidly search a very large list, we should consider using a more sophisticated search technique or a different way of structuring the data – this will be considered in a later module.

Sorting a list (lowest to highest values):

The Bubble Sort:

This method involves sorting a range of values into numeric order. The values will be stored in a 1-D array (typically integers).

The method involves repeatedly scanning the list and moving the largest value to the next biggest position at the top of the list. The routine is illustrated below:



The simplest coding for this would be:

```
namespace BubbleSort
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] vals = new int[] { 9, 7, 5, 2, 3, 1 };
            int numVals = 6;

            // begin simple bubble sort
            for (int i = 0; i < numVals - 1; i++)
```

```
            {
                for (int j = 0; j <= numVals - 2; j++)
                {
                    if (vals[j] > vals[j + 1])
                    { // swap the values
                        int temp = vals[j];
                        vals[j] = vals[j + 1];
                        vals[j + 1] = temp;
                    }
                }

            }
            // output the sorted list
            for (int i = 0; i <= numVals - 1; i++)
            {
                Console.Write(vals[i] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Can you identify any inefficiencies in this programming or the method of sorting?

From the above code, we can see that, given N items in a list, we have an outer loop that requires N-1 iterations each of which requires an inner loop to perform N-1 iterations. So the total number of comparisons is (N-1) x (N-1) i.e. $N^2 – 2N +1$. For applications involving the sorting of a large data set, this could have important implications regarding the time taken e.g. 1 thousand data items requires nearly 1 million comparisons and possible swaps. When considering the efficiency of a sort routine, we refer to the order of a search and use the so-called 'O notation' to describe the number of operations relative to N, the number of data items. The bubble sort is said to be of $O(N^2)$ where we refer to the highest power in the calculation of the number of comparisons or iterations.

Notes on Good Practice

We are using 'static' arrays which have a fixed size, declared at the outset. There are situations where the actual number of stored data values will vary at run-time, so it is important to ensure that the array is big enough to accommodate all likely eventualities otherwise the program will crash with an 'out of bounds' error. Some programming languages will not produce an error and simply access the next physical item of storage in the computer's memory. This can be even worse because the program appears to work properly but might generate strange results which may not be spotted – the consequences could be very serious!

An alternative to *static arrays* is to use an ArrayList object that allows you to create *dynamic arrays* that don't require you to specify a size in your program. You don't need to use these here but simply remember that there is this alternative for possible situations when it is impossible to predict data storage requirements.

Another situation that can arise, particularly involving array sizes, is that we write our program using fixed values for example:

```
int[,] myArray = new int[2,3];
```

We then proceed to use the same values in subsequent code e.g.

```
for (r = 0; r < 2; r++)
     {
         for(c=0; c < 3; c++)
         {
             Console.Write("Enter a value for row {0}, col {1}: ", r+1, c+1);
             myArray[r, c] = Convert.ToInt32(Console.ReadLine());
         }
     }
```

The problem with this is that if, at a later stage we decide that our array needs to be bigger, we have to go through the whole code changing the values – and possibly missing some!

A more sensible alternative is to use the following code (as we saw in an earlier example):

```
int nRows = 2, nCols = 3;
int[,] myArray = new int[nRows,nCols];
```

On-Line Resources

For an introduction to arrays, you should start off by looking at the following website which provides a good overview and examples:

http://zetcode.com/lang/csharp/arrays/

If you wish to look at a video for arrays, try:

https://www.youtube.com/watch?v=RQ0JHMGiobo

More details about arrays can be found at:

https://msdn.microsoft.com/en-us/library/ttw7t8t6.aspx

Sorting a list: *the Bubble Sort:*

https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm

Note that the bubble sort is not very efficient and we shall discuss this in a Module 3 and look at ways of improving it along with alternative methods.

Searching a list:

Linear (sequential) search:

https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm

This is not very efficient for large lists, so we might use an alternative method such as:

Binary search:

https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm

Core Practical Exercises

1. **Console Programs**

1.1 Write a program that will search a list of names for items with a given name and report how many, if any, have been found.

1.2 Write a program to implement the 'binary search' technique mentioned in the notes. The program should feature a stored list of at least 30 numbers in ascending order and invite the user to search for a given number.

1.3 One problem with the Bubble Sort is that the basic underlying algorithm extends to values that have already been sorted. Can you make a simple adjustment to make this code more efficient?

2. **Windows Form Applications**

2.1 Write a program featuring a text box in which the user can enter a reasonably large amount of text. The program will allow the user to then search that text for occurrences of a given letter and report how many times it occurs.

2.2 Extend the program in 1.2 to give a visual progress report to the user indicating the current search interval. The program should pause at each stage to allow the user to view the results and press a button to continue.

Further Exercises

1. Change the program in 2.1 to count the occurrences of each letter in the alphabet for the given text and display the results in a ListView box.
2. Create a program to generate a 'Magic Square' of order NxN where N is an odd number between 3 and 11. An example of a 3x3 magic square is shown below:

8	1	6
3	5	7
4	9	2

Each row, column and main diagonal adds up to the same amount (in this case 15).

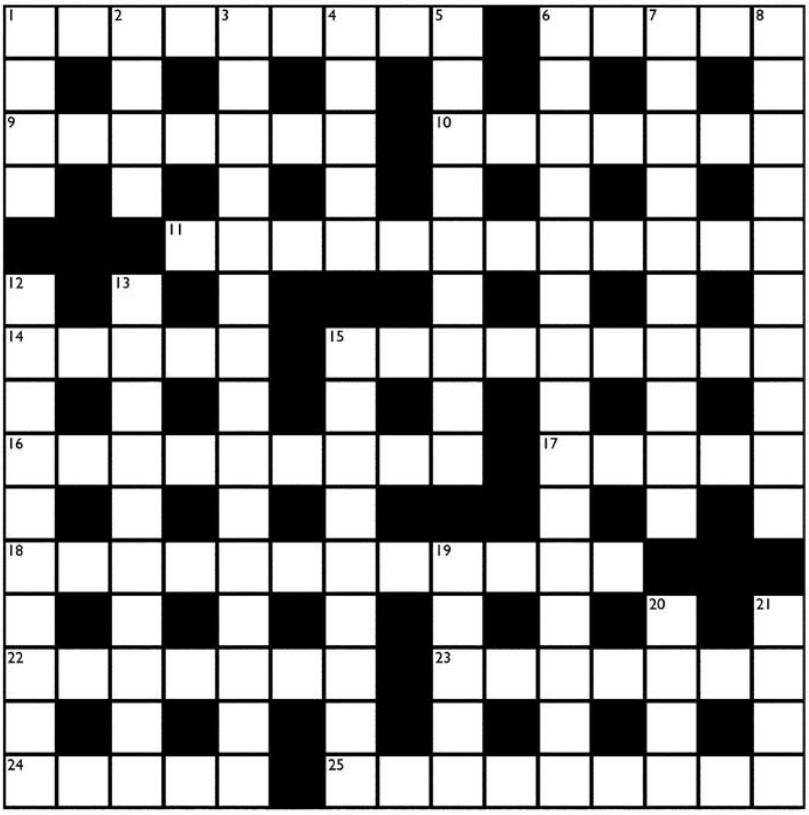
The user should input the size of the square (N) and the program will generate the result according to the following rules:

Imagine a rolled-up grid in which the top is connected to the bottom, but also the right-hand side flows over to the left-hand side. Then start off by placing a 1 in the middle of the first row. Subsequent numbers are placed according to:

Attempt to move diagonally upwards to the right, reappearing in the bottom row or left-hand column if necessary. Place the next number in this square unless it is already filled, in which case drop down one row from the current position and place the number in the current column position.

3. Crossword Numbering Program

   Most newspapers contain a daily crossword which usually feature a 15 x 15 grid such as:



   By examining the above grid, write a program that will define a grid pattern in some way (e.g. 0 represents a black square and 1 represents a white square) and then number the squares accordingly. You may require a second array for the clue numbers.