WEEK 7: PROGRAMMING WITH ARRAYS

Objectives

Following the completion of this topic, students should be able to:

- Understand the basic concept of arrays
- Identify common programming errors when using arrays
- Know various ways to initialise arrays
- Understand how to pass an array to a function.
- Understand multidimensional arrays and how to handle them
- Understand the basics of C-strings

TASK 1. SOME BASICS ABOUT ARRAYS

An array is a data structure, which can store multiple data of a particular type in contiguous memory locations. The advantage of using one array to store *N* items rather than *N* individual variables is that only one identifier is required. This is particularly beneficial when passing data to functions. The syntax to declare an array is:

```
element-type array-name[capacity];
```

where `element-type` is any valid data `type`, `array-name` is the name of the array and `capacity` is the number of items contained in the array - the size of the array.

- `capacity` must be a *constant* value. I.e. you **cannot** use a *variable* to set the size of the array.
- Each individual item in the array is accessed using a unique reference, which is constructed of the array identifier/name along with the relevant index. The index sequencing begins at **0** for the first item and ends at ***N*-1** for the last. The syntax for accessing an individual array element is:

  ```
  array-name[index];
  ```

  For example:
  - When `index` equals 0, `array-name[index]` will enable access to the first item.
  - When `index` equals 1, `array-name[index]` will enable access to the second item, and so on.
- The collection of data in the array is most often related in some particular way, though this is not a requirement. For example;
  - Data for an individual student's marks for several assessment items can be stored in an *array* such as:

    ```
    float marks[5]; // 1 student, 5 marks
    ```
  - Data for multiple students' marks for one assessment item can be stored in an *array* such as:

    ```
    float studentsMarks[50]; // 50 students, 1 mark
    ```

- The most common programming error relating to arrays is referred to as an "*index out of bounds*" error. That is; attempting to access memory through the array indexing that is not allocated to the array.

For example:

```
int array[5];
```

declares an integer array with 5 items. To access the items, you need to provide indexing of 0, 1, 2, 3 or 4. If, for example, you attempt to access `array[5]` you will be accessing memory not allocated to the array and also most likely not allocated to your program.

- Depending on the compiler used and how '*bad*' the memory violation, this will often result in the program *crashing* during execution providing a '*segmentation fault*' error.
- However, sometimes the program will not crash in spite of the memory violation. Hence, you can't rely on the '*segmentation fault*' error during testing to alert you to the error.
- Hence, you have to be ***very*** careful when using arrays to avoid this.

Do the following:

a. Now type the following program and execute it.

```
#include<iostream>
using namespace std;

const int SIZE = 5;

int main()
{
        int score[SIZE], max;

        cout << "Enter " << SIZE << " scores: \n";

        for (int i = 0; i < SIZE; i++)
              cin >> score[i];

        max = score[0];
        for (int i = 1; i < SIZE; i++)
        {
              if (score[i] > max)
                    max = score[i];
        }
        cout << "Max score is: " << max << endl;

        return 0;
}
```

Note that SIZE is a **constant** value!
NEVER use a variable to determine the size.

b. What is the overall task of the program?

c. What is the output of the above program when the following values are entered: 2, 8, 12, 21, and 4?

d. What will happen if the following values are entered, 2 8 12 21 4 35? Answer this question *before* testing the program, and then compare the result with your answer. If it is not the same, can you figure out why? If not, ask your tutor for assistance.

e. Modify the above program so that the user can enter however many scores they want. Your program should ask the user prior to entering the values, how many

will be entered and use this value to determine the number of times the loop will iterate. Once all scores have been entered, determine the max score and output as before.

f. Compile and run the program testing it with any 5 values to ensure it works accordingly.

g. Now suppose that the user has input 4 for the number of scores he wants to input and enters 10 20 30. What will happen? What will happen if the user enters 10 20 30 40 50?

h. Run the program entering 20 scores. Did you encounter any problems during execution of your program? If so, find and correct the logic errors. If you cannot fix the problem, ask your tutor for assistance.

TASK 2. PROGRAMMING WITH FUNCTIONS AND ARRAYS

a. Type, compile and run the following program.

```
#include <iostream>
using namespace std;

int main ()
{
	int n = 5;
	char symbol = '^';

// The following code prints a diamond pattern
	for (int r = 0; r < n; r++)
	{
		for (int c = r; c < n; c++)
		    cout << symbol;
		for (int c = n-r; c < n; c++)
		    cout << ' ';
		for (int c = n-r; c < n; c++)
		    cout << ' ';
		for (int c = r; c < n; c++)
		    cout << symbol;
		cout << endl;
	}
	for (int r = 1; r <= n; r++)
	{
		for (int c = 0; c < r; c++)
		    cout << symbol;
		for (int c = r; c < n; c++)
		    cout << ' ';
		for (int c = r; c < n; c++)
		    cout << ' ';
		for (int c = 0; c < r; c++)
		    cout << symbol;
		cout << endl;
	}
	cout << endl;

	return 0;
}
```

b. What is the output?

c. Add the *prototype* and *definition* (**header only - no code yet**) for a function named `printPattern`, which takes 2 parameters: one to represent the size of the pattern and the second to represent the character to be displayed in the pattern. The two parameters are to take the default values of the current two variables in main.

d. Compile your program and fix any errors/warnings.

e. Cut and paste the code that prints the pattern from main to the function, `printPattern`.

f. Compile your program and fix any errors/warnings.

g. Add a function *call* in main to `printPattern`, using the default parameters only, then compile and run your program. Is the output the same as in b. above? If not, find and fix the errors.

h. Add three function calls in `main` to `printPattern`: one overriding the default character with any other character you'd like, another overriding the default size with any other size you'd like and the third overriding both the size and character.

i. Compile and run the program to ensure it works as expected.

j. Modify main, so that when you call `printPattern`, you will pass data stored in two arrays. One array will hold the size of the pattern and the other the character for the pattern. *Don't forget to declare and use a* ***constant*** *for the maximum size of the arrays.* Initialise the arrays with whichever sizes and characters you think best.

k. Using a `for` loop, *call* `printPattern`, each time passing the corresponding size and character variables from the arrays. If you have trouble determining how to do so, do not spend too much time on it. Ask for assistance after making a *reasonable* attempt.

l. Compile and run the program to ensure it works as expected.

TASK 3. CHALLENGE TASK

a. Your task is to implement a program that does the following:

- Create an array of 15 integers, which may contain the same values.
- Read each element of this array and wherever the array has two or more consecutive numbers that are identical, replace them by one copy of the number so that no two consecutive numbers in the array are the same.
- Any unused elements at the end of the array are set to -1. For example, if the array you created in Step 1 is [0 0 0 0 1 1 0 0 0 7 7 7 1 1 0], your program should output [0 1 0 7 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1].

TASK 4. INITIALISING AN ARRAY

In C++, you can declare and *initialise* an array at the same time. The syntax for initialising an array during declaration is:

```
array-type array-name[capacity] = {val1, val2, val3, etc};
```

where `array-name` is the name of the array, `array-type` is the type of the array and `capacity` is the size of the array.

- Initial values of `val1`, `val2`, `val3, etc.` are assigned to the elements of the array consecutively. I.e.
  - `array-name [0]` = `val1`
  - `array-name [1]` = `val2` and so on.
- If an array is declared with size *N* and less than *N* initial values are given, the remaining elements in the array will be initialised with the default value of 0.
- However, this can only occur during an *initialisation* statement. For example:

```
array-type array-name[N] = {val1};
```

will create an array of size *N* and set the first element to `val1` and the remaining *N*-1 elements to 0.

- When providing initial values during the declaration, as above, it is possible to omit the *capacity* of the array and by default where *N* values are given, the size will default to *N*. For example:

```
float myArray[ ] = {22.2, 43.4, 66.1, 7};
```

will declare and initialise an array of size 4.

Now answer the following questions:

a. In the following array declarations,
   - Identify those with errors and explain the error
   - In those without errors, what is the size of the array and what values will be stored in each of the array elements.

```
i.  int array[4] = {2, 3, 56, 7, 8};
ii. int array [] = {8, 7, 6, 0};
iii.int array [5] = {8, 7, 6, 0};
iv. float array [5] = {8.5, 7.33, 6, 0};
v.  int array [] = {8, 7, 6.0, 0};
```

b. Identify any error(s) in the following code. If there are no errors, what will be the values stored in the array?

```
const int SZ = 10;
int iArray[SZ];
for (int index = 1; index <= SZ; index++)
        iArray [index] = 3*index;
```

c. If you found no error(s) in the code above, look again. While there are no syntax errors, there are logic errors. What are they?

d. What do you think is the output of the following code? ***Do not code it yet***.

```
#include<iostream>
using namespace std;

const int SZ = 4;

int main()
{
```

```
        int myArray[SZ] = {8, 7};
        for (int index = 0; index < SZ; index++)
              cout << myArray[index] << " ";

        return 0;
}
```

e. Implement the above code and execute it. Is the output the same as you determined in step (d) above. If not, work through the code, step by step, explaining the logic as you go.

TASK 5. PASSING AN ARRAY TO A FUNCTION

A function can have an array as a formal parameter. The syntax of a function *prototype* with an array as a parameter is:

```
return-type function-name (type array-name[]);
```

- Declaring an array as a parameter is almost the same as that of any other parameter, except that in the *prototype* and *definition* the operator, [ ], is added after the name, indicating that the parameter is an array.
- Arrays are passed by *reference* by default and there is no way to override this.
- During the function *call*, only the array name is supplied. I.e. do *not* include the brackets [ ].

Do the following:

a. Type the following code and execute it.

```
#include<iostream>
using namespace std;

const int SZ = 5;

void FillArray(int a[], int size);

int main()
{
      int array[SZ];

      FillArray (array, SZ);

      return 0;
}

void FillArray(int a[], int size)
{
      for (int i = 0; i < size; i++)
      {
            cout << "Enter element number `" << i+1 << "': ";
            cin >> a[i];
      }
}
```

b. What does the function `FillArray` do?

c. Normally, we use '&' to pass a variable by reference to a function. Identify which of the parameters of the `FillArray` function have been passed by reference? Why there is no '&'?

d. Write a function named `DisplayArray`, which displays all elements of a given array. The prototype of this function is as follows.

```
void DisplayArray(int a[], int size);
```

e. Compile the program and test this new function.

f. Write a function named `FindLargest`, which returns the largest of all elements of a given array. The prototype of this function is as follows. *Note: This function does* ***not*** *produce output.*

```
int FindLargest(const int a[], int size);
```

g. Compile the program and test this new function.

h. Notice the keyword **`const`** in the prototype above?
   i. What does it mean?
   ii. Which parameter(s) is it linked with?
   iii. Why has it been included?

TASK 6. MULTI-DIMENSIONAL ARRAYS

So far, we have looked at what is referred to as single dimension arrays. That is, they have one dimension only. An array, however, can have more than one dimension, which is referred to as a multi-dimensional array. They are, in effect, arrays of arrays.

- The number of dimensions is not restricted beyond what one can effectively grasp and code.
- They are similar to standard arrays with the exception that they have multiple sets of square brackets [ ] after the array identifier: one for each dimension. For example, the following declarations declare three arrays: a one-dimensional array, a two-dimensional array and a three-dimensional array.

```
int array1D[DIM1];
int array2D[DIM1][DIM2];
int array3D[DIM1][DIM2][DIM3]
```

- Again, the capacity of the arrays in all dimensions must be a *constant* value.
- Most often, the type of multi-dimensional array we use is a two dimensional array. This can be thought of as a grid of rows and columns.
- When accessing each individual element of the array, you need to specify the coordinate for each of the dimensions. For example:

```
int array3D[5][3];
```

Declares a two-dimensional integer array with dimensions, 5 and 3. This can be viewed as 5 rows by 3 columns. That is, there are 15 integers in total. The illustration below demonstrates the indexing of each of the 15 elements in the array.

		Columns		
	index	0	1	2
Rows	0	[0][0]	[0][1]	[0][2]
	1	[1][0]	[1][1]	[1][2]
	2	[2][0]	[2][1]	[2][2]
	3	[3][0]	[3][1]	[3][2]
	4	[4][0]	[4][1]	[4][2]

So, `array3D[0][0]` enables access to the element at row 0, column 0, while `array3D[2][1]` accesses the element at row 2, column 1.

a. Examine the code below to determine the expected output. What is the output? ***Do not code it***.

```
const int DIM = 4;

int myArray[DIM][DIM];

for (int i = 0; i < DIM; i++)
    for (int j = 0; j < DIM; j++)
            myArray[i][j] = j;

for (int i = 0; i < DIM; i++)
{
        for (int j = 0; j < DIM; j++)
            cout << myArray[i][j] << " ";
        cout << endl;
}
```

b. Implement the code and compile and run it to verify your answer in (a).

TASK 7. MORE MULTI-DIMENSIONAL ARRAYS

a. Write a program that declares and initialises a two-dimensional array named `magicSquare`, which consists of 3 rows and 3 columns, whose components are shown below. Assume that the array is of type `int`. *Hint: remember how to initialise a one-dimensional array?*

8 1 6
3 5 7
4 9 2

b. Calculate the sum of each of the rows and output the result as: *`The sum of row <r> is <sum>.`*

c. Calculate the sum of each of the columns and output the result as: *`The sum of column <c> is <sum>.`*

d. Calculate the sum of the diagonal from index [0][0] to [2][2] and output the result as: *The sum of the diagonal from* `[0][0] to [2][2]` *is <sum>.*

e. Calculate the sum of the shaded numbers and output the result. *Note: You must **not** hard code the result, e.g. by putting* `cout << 8+1+6+7+2`. *You must use nested loops to solve the problem.*

8	**1**	**6**
3	**5**	**7**
4	**9**	**2**