Arrays

In this lecture we are building on our knowledge of data types by looking at a structured type called an array.

As per usual there are some other concepts and ideas to grapple with first.

Lists

A list is simply a sequence of values with a sense of position. The values are generally referred to as elements, with this in mind there is a definite first element and a definite last element. Each element except the first has a unique preceding element (predecessor) and each element except the last has a unique succeeding element (successor).

It is very common for a problem to require storing and manipulating lists of data values.
We may have a list of players or a list of scores. This could need also ordering, according to rank.
Although lists can be quite complicated, when they start to store large amounts of data, we will be focusing on lists of simple values and data types.

We may manipulate lists in many ways; we may insert new elements, delete old elements. We may wish to search the list for a particular element. We may wish to sort the list into some order, or replace one value with another.

Array: Definition

There is a definite need to store information in lists; C++ provides the most basic structured type, the array as one method of implementing a list.

Definition: Array = (Computer Science) An array, also known as a vector or list, is one of the simplest data structures in computer programming. Arrays hold a fixed number of equally-sized data elements, generally of the same data type. Individual elements are accessed by index using a consecutive range of integers

An array is therefore defined by the following properties:

- Array:
    - A sequential list of data values that are all of the same type (homogeneous)
    - Individual elements are accessed by their position (1^{st}, 2^{nd}, 3^{rd}, 25^{th})
    - Has a fixed capacity called the dimension, and can only hold up to that many elements at once
    - Fundamental C++ mechanism for storing lists of data

Examples of array uses for games including DirectX using a 1D array to describe the screen surface.

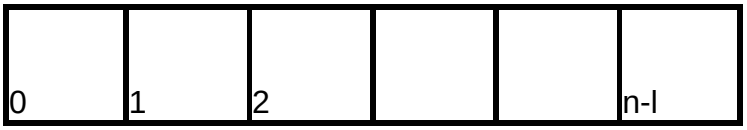
Mathematically Terminology and Notation

This is the bit where we look at the terminology, they way arrays are expressed.

In mathematics lists are often denoted by using a name (for the list) and by attaching a sub-script to indicate a particular position being discussed:
X_1, X_2, X_3

As subscripts are not supported by text editors you often see the following notation:
X[0]. X[1], X[2]

In C++ positions is an array are numbered sequentially, starting with 0:



This known as row-major order: It indicates that the elements of each row are stored in order.

Notice that the elements (or cells or positions) are numbered 0 through N-1 When we use N like this, we mean in a statistical sense: In statistics, N is the size of a sample, or in our case the size of the array. N can be of any number.

Array Declarations

An array is a variable, it has parts and is structured, but it is still just a variable. As with variables every array must have a name and that identifier must be declared before it is used. An array is declared by giving the type of data, the array identifier, and the number of items to be stored, in brackets.

Syntax
The syntax for storing single dimension arrays is:

<data_type> ***<array_name>*** ***[array_size];***

<data_type> = C++ data type such as char, int, string or user defined. This indicates the data type the array will store

<array_name> = The identifier of our array – the name we give it

[array_size] = The dimensions of the array – how many elements and therefore instances of data it will hold

Although the elements maybe of any type at all the dimension must be a positive integer constant or expression that evaluates to a positive integer constant.

Declaration

As we should be able to imagine from the syntax above array declaration is very similar to variable declaration.

```
// Declare an Array of 25 integers called Student_Marks
int Student_Marks[25];
```

Remember though, that the dimension must be a positive integer constant or expression that evaluates to a positive integer constant.

```
const int BUFFER_SIZE = 1000;
const int DICE_ROLLS = 347;
// constant integer dimension
char Buffer[BUFFER_SIZE];
// constant integer expression
// used as a dimension
int DiceFrequency[DICE_ROLLS+1];
// Literal constant used a dimension
string TheWords[100];

int numItems = 100;
// Not valid as numItems variable
// is not a constant
string TheItems[numItems];
```

Accessing the Elements

As mentioned we refer to the individual positions in an array as the elements or cells.

To access an element with the array we use the array name followed by an integer, enclosed in square brackets, that indicates a position with in the array. An integer used in this fashion is often called the index of the element.

If this seems a little like alien to you then lets look at array index's a little bit more:

In this example:

```
// Declare an Array of 25 integers called Student_Marks
int Student_Marks[25];
```

We declare an array, called Student_Marks, of 25 **int** values, indexed from 0 to 24.

Note that there is not an element with index 25. The 25 in the definition specifies the total number of elements to be allocated, not the upper limit of the index value. Since the index values start at 0, the last index value is always one less than the number in the definition—24 in this case. Forgetting this is one of the most common C++ errors, especially for beginners.

The simplest types of array and the type we are mainly concerned with during this are 1D arrays and it is best to view these as a row of pigeon or cubby holes, where each box stores a variable or type that was used in the array declaration.

The numbers in the cubby holes represent the data values we are storing, the name at the side represents the name of the array and the name and number above represent the index that corresponds to the relevant cubby hole.

X[0] == 12

What we are indicating here is that the index is equals, in a mathematic sense, the number on the right.

Each of the 13 array elements,

X[0], X[1],..., X[24]

Behaves in every respect as an ordinary integer variable, we can use the individual elements of an array in any way a simple variable of that type may be used.
A value may be assigned to an element:

X[5] = 79;

or its value may be used in an expression:

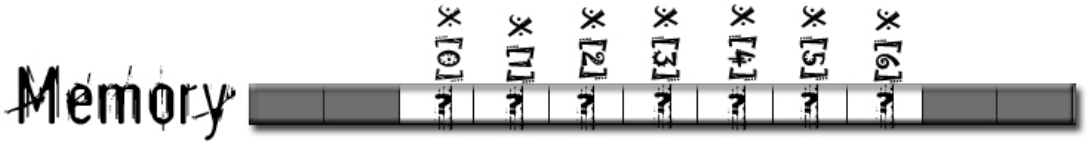
cout << X[13];

```
if(X[0] > X[1])         // compares element 0 and element 1
{
        // do something…
}
```

Arrays Declaration and Memory

```
// Declare an Array of 25 characters called Student_Marks
char X[13];
```

Looking at the above declaration: At runtime a block of memory will be allocated for our array variable **X**.
X has 13 elements of type char, remembering that char is stored in one byte of memory; X will therefore require 13 bytes of memory.
The blocks of memory are allocated contiguously (as a single chunk).
It looks something like this:



Logically the valid index ranges from 0-12, the dimension minus one.

As with any variable declaring it does not initialise it. That is why in the diagram the elements are filled with question marks.

Failing to initialise variables is often a source of errors, as we have seen during the tutorials; this is the same for arrays.

Arrays and sizeof

The expression

```
sizeof(X);
```

Returns the size, in bytes, of the whole array. This is calculated at compile-time.
No run-time record of the size of the array is maintained by C++.
Remembering the size of the array is the responsibility of the programmer.
Forgetting this is a common source of runtime problems.
For example:

```
int Student_Marks[7];
```

```
// no Student_Marks[7] -- causes a run-time error.
Student_Marks[7] = 99;
```

Will not be recognised as an error by the compiler, but will probably cause an unexplained run-time crash or incorrect calculations.

If you want to dynamically calculate the number of elements in an array at runtime, you can divide the number of bytes by the size of the type of the array elements:

```
sizeof(Student_Marks) / sizeof(int)
```

This only works with the original array name. You cannot do this with an array parameter name in a function.

Initializing an Array

The act of declaring our array does not initialise it; the array has no values and is empty. Just like variables arrays are initialised when they have values assigned to them.

Loops

One of the most common ways to initialise an array is to traverse the array, using a loop, assigning values.
A for loop is generally used to initialise an array, as we set the size of the array we know how many elements the array contains and therefore iterating by counting using a for loop is best.

```
const int MAX_SIZE = 30;            // Array Size.
int Player_Scores [MAX_SIZE];

int idx;
for(idx = 0, i < MAX_SIZE, i++)
{
      Player_Scores [i] = 0;
}
```

The for loop counter **`idx`** is used as the array index within the loop body.
`idx` starts at 0 and is incremented by one after each loop pass.
The loop terminates when **`idx`** equals the size of our array.

The loop design guarantees that each element of the array will be accessed, in turn, and that the loop will terminate before an invalid index access is attempted.

It is important to understand this as you will use it during your assessment and is standard processing of arrays throughout much of programming.

Instead of using loops we can use some short cuts to initialise our arrays; these short cuts are listed below. They should generally be avoided in favour of using loops, although in certain cases such as small arrays or where we wish the values to all equal a certain value such as 0 or NULL a loop would be unnecessary.

Curly Brackets
Arrays be they global or local may also be explicitly initialised by following the definition by an = sign and a list of initialising values, of the correct type, enclosed in braces and separated by commas.

```
int Student_Marks[5] = {68, 45, 32, 75, 57};
int Student_Marks[] = {68, 45, 32, 75, 57};
```

We have this opportunity when we declare an Array, to assign initial values to each one of its elements using curly brackets { }

If we try to assign values in this fashion after array declaration we will generate syntax errors.

Using curly brackets enables us to initialise the array with increased ease. If we initialise only a few of the values, or even just one, then the then rest of the array is initialised to a default value, generally 0.

Take the following examples:

Set ALL elements to 0:
```
int Player_Order [5] = {0};
```

Set first element to 1 and rest to 0:
```
int Player_Order [5] = {1};
```

Set first element to 1, the second to 2, and the rest to 0.
```
int Player_Order [5] = {1,2};
```

Set elements to 5, 4, 3, 2, and 1 respectively.

```
int Player_Order [5] = {5,4,3,2,1};
```

Declare an array and set elements to 4 and 3:

```
/* C++ sets size to 2 automatically */
int Player_Order [] = {4,3};
```

It must be noted that if some, but not all, of the items in an array are initialised explicitly, then the rest will be initialised to zero, even for local declarations in function bodies.

When assigning the initial values using curly brackets it is possible for us to miss off the array size, the array size is then of a size equal to the number if initial values. Although possible to do this, it is should generally be avoided in favour of indicating the array size.

If you are wondering about chars and strings, then they to are initialised to 0 in the above cases. Remember default conversions that are part of C++, integers and characters can be used interchangeably.

Global

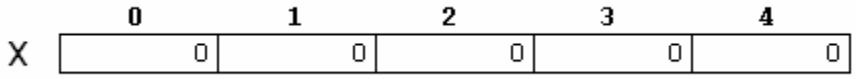
It is important to note the following applies only to global arrays; this technique of automatic initialisation, when used for local arrays can result in runtime errors.

An array declared with global scope will be initialised to zeros, just like any other global variable. An array declared locally, inside a function body, will not be initialised, just like any other local variable.

Take the following examples:

Set ALL elements to O:

```
int X[5];
```



The initialisation options using curly brackets apply to global arrays, but initialisation using for loops if attempted globally will result in syntax errors.
For loops must appear in a block of code.
Remember we should try to avoid the use of global variables, this also applies to arrays.

Const

Arrays, as they effectively store data types they can also be declared constant the values they hold not changing.
To do this we use the keyword const as normal.

```
const int X[5];
```

Once we have initialised our array, which is only possible during declaration via curly brackets ({ }) we can no longer alter the values stored in the array.

Usage of an Array

Sometimes only some of the elements in an array actually store meaningful data, the elements store initial values such as 0 or NULL.

The number of elements in the array that hold meaningful data are known as its usage.

Note: Arrays and other data types

A quick word about arrays and other data types: Arrays can be of other data types, char, string, float, pointers, user defined types.
When other data types are used, we've mainly used int and char in the examples they follow the same rules declaration, initialisation and traversing. Just remember that arrays can only hold the data types they are declared to hold.

Arrays and Functions

There are different ways to pass arrays to functions, there is passing individual array elements and passing the whole array.

If we have an array:

```
int X[13];
```

Then `X`, the name of the array refers to the whole array and `X[/*some value*/]` refers to an element in the array at the index specified by the value.

Array Elements as Parameters

It is quite easy to pass individual array elements to a function.

These individual array elements can be *Value* or *Reference* parameters of the function, depending on whether you use an & (ampersand) or not.
Individual Array elements are Value Parameters by default.

A single array element can be treated just like any simple variable of its type so the receiving function treats the receiving function just as a simple variable.

When passing a single array element the actual parameter is the array name with the index attached.

```
int X[13];

// read some values in to X[]

swapInts(X[0], X[5]);

// the function definition
void swapInts(int x, int y)
{
      int temp = x;
      x = y;
      y = temp;
}
```

Because in this example we are passing by value, we are making a copy of the values in memory we will not have swapped the actual values around in the array.
To do that we need to pass the array elements by reference:

```
int X[13];

// read some values in to X[]

swapInts(X[0], X[5]);

// the function definition
void swapInts(int& x, int& y)
{
      int temp = x;
      x = y;
      y = temp;
}
```

This will swap the actual values around inside the array.
Note though that the functions does not know or care that the actual parameters are elements of an array.

Aggregate Operations
First of all, what is an aggregate operation?
An aggregate operation is an operation that is performed on a data structure, for example an array, as a whole rather than performed on individual elements.

Examples of aggregate operation include:
Returning an entire array from a function
Assigning one array to another

Comparing two arrays
Adding two numeric arrays
Passing an array as a parameter

Many of these operations are not supported as default – such as comparing to arrays, it is however possible for you to implement them your self.

Whole Arrays as Parameters
Entire arrays can also be passed as function parameters.

When we pass an array as a parameter:

- The actual parameter is just the array name
- The formal parameter is an identifier followed by an empty pair of square brackets

```
// Function that takes two whole arrays
// as parameters
bool AreEqual(int arrayOne[], int arrayTwo[]);

// Two arrays
int valuesOne[5] = {0};
int valuesTwo[5] = {0};
// calling the function AreEqual
AreEqual(valuesOne, valuesTwo);
```

The example above shows how to define function parameters that accept arrays and how to pass the whole array.

The function has no way of knowing the size (or dimension) of the array, therefore we define no size for the array in the parameters list.

By default if the array name is used, so that the whole array is passed, it is passed by reference. Any operations that modify the array are reflected back in the calling function.

By adding the word const in our functions parameter list causes the array to be passed by constant reference, therefore no modification of the array can occur.

Arrays are passed by reference to reduce memory use and improve speed, copying lots of data uses memory and causes slow down.

**Note: As you pass arrays by reference they can altered in functions, affecting there values elsewhere, if you are passing an entire array to a function, and you don't want values to be changed there are ways to avoid it:*

- *Declare another array inside the function (i.e. local to the function), and copy across the values from your main array and then use the other array inside the function, or,*

- *Use the const keyword before the array parameter to ensure that no values in the array can be changed.*

The above methods can be useful in different circumstances, although generally pass by constant reference is the quickest and less memory intensive method.

Cautions and Errors

When we attempt to access parts of the array that are out of bounds, or have not been initialised then we get errors, usually runtime errors that will usually crash our program.

Consider the following example. We need to include the **iCount** variable so the functions can know the size of our array otherwise we could end up accessing the array out of bounds.

```
// initialise array
int Student_Marks[] = {68, 45, 32, 75, 57};
// call SumOf function
int iTotalMarks = SumOf(Student_Marks, 5);

int SumOf(int Number_Values[], int iCount)
{
      int i, iSum = 0;

      for(i = 0; i < iCount; i++)
            iSum += aiValues[i];

      return iSum;
}
```

Note that you cannot do away with the **iCount** parameter and calculate the number of items within the function itself. That would require ***run-time information*** about array sizes, which is not maintained by C++.
However, the way we have passed the **iCount** value is not very good, since the size of the array might be changed in the definition without the **iCount** value being changed in the function call - this is particularly true if the definition is separated from the function call by other definitions and statements.

A better way would be to define a symbolic name for the size of the array and use that in both the array definition and the function call. For example:

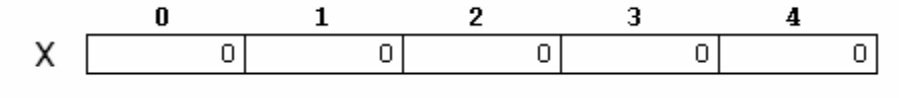
```
// define array size
const int MARKS_SIZE = 5;
// declare and initialise array
int Student_Marks[MARKS_SIZE] = {68, 45, 32, 75, 57};
// invoke the SumOf function
int iTotalMarks = SumOf(Student_Marks, MARKS_SIZE);
```

A method of calculating the size of the array at runtime is by using the **sizeof** operator in the function call to calculate the number of elements:

```
iTotalMarks = SumOf(Student_Marks, sizeof(Student_Marks) /
sizeof(int));
```

Array Index
It is important to remember that the array index is the dimension (or size) minus one:

```
int X[5] = {0};
```



The valid indexes, and therefore memory locations, for the above array is X[0] to X[4].

The problem with out of bounds statements in C++ is that there is no error checking at compile-time or run-time.

I have mentioned that usually out of bounds arrays indices crash the program, but that is not always the case. On machines such as windows 95/98 it could cause other problems by accessing a memory location that holds a value belonging to another program. If this is value modified corruption of the other program could take place.

Assert

Luckily C++ does include a useful tool that allows us to check at run time whether an array is being accessed out of bounds, is called the `assert()` function. The `assert()` function is actually part of the C library of functions, but it is something we can have access to.

To obtain access to the procedure is to put the include directive

```
#include <cassert>
```

At the top of your program

The syntax of `assert()` is:

```
assert(logical-expression);
```

It checks that certain conditions which you think should be true (include it at those points where we need to check these conditions) do indeed hold. If the program executes the above statement and the expression "logical-expression" is false, the program will terminate at that point with an appropriate error message. "assert" statements can be especially useful to check maximum/minimum value conditions, e.g.:

```
assert(my_var >= 0 && my_var <= MAXIMUM_VALUE);
```

They can be used to check arrays and cause the program to exit before any memory out of bounds is accessed.
They have limited usefulness though as we still need to know the length of the array to check that we are not going to be accessing out of bounds.
Most modern OS, and some robust old ones such as unix, will terminate any program that causes a memory access violation.

There are other issues with `assert()` as well:

- Often it is only possible to use `assert()` calls in the debug builds of a program. In many cases debug builds are disabled as the standard of the compiler. With Visual Studio debug builds are default
- Produces very limited diagnostic information, such as line number and function where it was triggered

Instrumenting Code

This is basically coding in our own error checking. By doing this we can print out longer and more detailed information, using the `exit()` function we can even quit the program if we encounter an error.

```
const int ARRAY_SIZE = 15;
int valuesOne[ARRAY_SIZE] = {0};

int iArrayAccess = 6;
// check we are not accessing out of bounds
// if we are exit
if(iArrayAccess >= ARRAY_SIZE)
{
      cout << "illegal access of array valuesOne attempted\n"
            << "see line and the variable iArrayAccess\n";

      exit(1);
}
```

This technique has the benefit that it is not disabled in non-debug builds.