# LAB 10

**Summary**

|  |  |
| --- | --- |
| **Items** | **Description** |
| Course Title | Programming Fundamentals |
| Lab Title | Arrays in C++ |
| Duration | 3 Hours |
| Operating  System/Tool/ Language | Ubuntu/ g++/ C++ |
| Objective | To get familiar with use of Arrays in C++ |

# Arrays:

**Syntax:**

**type name [elements];**

**for accessing : name[index]**

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

This means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be represented as:



where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

**type name [elements];** where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.

Therefore, the foo array, with five elements of type int, can be declared as:

*int* foo [5];

NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

## Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

*int* foo [5] = { 16, 2, 77, 40, 12071 };

This statement declares an array that can be represented like this:



The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values,

one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

*int* bar [5] = { 10, 20, 30 };

Will create an array like this:



The initializer can even have no values, just the braces:

*int* baz [5] = { }

This creates an array of five int values, each initialized with a value of zero:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

*int* foo [] = { 16, 2, 77, 40, 12071 };

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

1 *int* foo[] = { 10, 20, 30 };

2 *int* foo[] { 10, 20, 30 };

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

## Accessing the values of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

name[index]

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



For example, the following statement stores the value 75 in the thirdelement of foo: foo [2] = 75;

and, for example, the following copies the value of the third element of foo to a variable called x: x = foo[2];

Therefore, the expression foo[2] is itself a variable of type int.

Notice that the third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2]. By this same reason, its last element is foo[4]. Therefore, if we write foo[5], we would be accessing the sixth element of foo, and therefore actually exceeding the size of the array.

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.

At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets [] with arrays.

1. *int* foo[5]; *// declaration of a new array*
2. foo[2] = 75; *// access to an element of the array.*

The main difference is that the declaration is preceded by the type of the elements, while the access is not. Some other valid operations with arrays:

1 foo[0] = a;

2 foo[a] = 75;

1. b = foo [a+2];
2. foo[foo[a]] = foo[2] + 5;

//Reference website: <http://www.cplusplus.com/doc/tutorial/arrays/>

Sample program #01 :

#include<iostream> using namespace std; int main()

{

cout<<"EXAMPLE OF ARRAYS"<<endl;

int a[5];

for(int i=0;i<5;i++)

{

cout<<"enter the value # "<<i+1<<"\t"; cin>>a[i];

}

cout<<"you have enterd follwing values"<<endl;

for(int j=0;j<5;j++)

{cout<<a[j]<<endl;}

}

Sample Program # 02

#include<iostream> using namespace std; int main( )

{

int avg, sum = 0 ; int i ;

int marks[10] ; /\* array declaration \*/ for ( i = 0 ; i <= 9 ; i++ )

{

cout<<"enter marks";

cin>>marks[i]; /\* store data in array \*/

}

for ( i = 0 ; i <= 9 ; i++ )

sum = sum + marks[i] ; /\* read data from an array\*/ avg = sum / 10 ;

cout<<"average ="<<avg ;

}

**Lab Tasks**

Task#01

Write a C++ Program, which asks user for five integer values. Store the values in Integer Array. Calculate Average, Maximum Value, Minimum Value from Array.

Task#02

my\_array= [13, 99, 6, 76, 11, 83, 8, 67, 66, 22, 96, 46, 21, 65, 48, 22, 28, 11, 83, 87,10]

Write a c++ code to print my\_array in reverse order.

Task#03

Declare an array by the name of ‘arr’ and initialize it with the following values ‘={32.20,41.88,16.12, 23.88, 7.21}’. You can hard code the array just don’t specify the array size as for this problem.

Understand and calculate the size of the array in a variable called ‘size’.

Traverse the array using any loop and keep storing the elements at the same index after dividing it by two.

Display the array being halved.

Display the sum of the same array being halved.



Task#04

Take an array of length n where all the numbers are nonnegative and unique. Find the element in the array possessing the highest value. Split the element into two parts where first part contains the next highest value in the array and second part hold the required additive entity to get the highest value. Return the array index value using a function named ‘splitBig( )’.

Input: 4 8 6 3 2

Output: 4 6 2 6 3 2

Note: use index number to return a value from an array.

Task#05

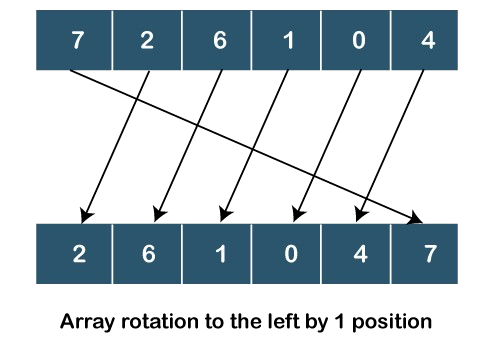
Find missing elements in an array of 100 integers, which contains numbers between 1 and 100.

Task#06

Let the user enter 50 values.

* Rotate the array entered by the user (by 25 positions)
* Search a number entered by user

Hint: In left rotation, the array elements rotated to the left with the specified number of positions. It rotates the array in the clockwise direction. The lowest index element moves to the highest index in rotation.





Task#07

Azam is a property dealer. He wants to buy and sell property in Bahria Town. He knows the prices that the rate for the first day of the week is 1500000, 140000 for 2nd day,130000 for 3rd day, 120000 for 4th

day, 110000 for 5th day, 100000 for 6th day and 90000 for 7th day. Help him find the best day he should

buy the property. Then find the best day to sale this property.

Task#08

A traveller wants to travel from point A to point B and he has two path options (path1, path2). Find a new path with minimum cost from the beginning of any path to the end of any of the two paths. We can switch from one path to another path only at the common elements.

The path costs are given:

path1= [2,3,7,10,12]

path2=[1,5,7,8]

**Output:** 20

**Explanation:**

first path will be: 2+3+7+10+12 = 34

Second path will be: 2+3+7+8= 20

As second path is smallest so output will be 20.

**Submission Instructions:**

**Submit a zip folder** containing cpp files

Naming convention for cpp files: q1.cpp, q2.cpp etc

Naming convention for zip folder: **Rollnum\_lab9\_secA.zip**

**Note: Lab task will not be marked if above mentioned instructions are not followed.**