
**School of Computing and Informatics**

COMP315 – Advanced Programming

**Chapter 5.1.2 – Arrays (Using class template $< array >$)**

# 1 Introduction

C++ provides a class template $< array >$ for creating and managing arrays in memory. To declare an array you use a declaration of the form

```
#include<array>
array<type, size> arrayName;
```

Listing 1: How to declare an array using class template $< array >$

The notation $< type, size >$ indicate that array is a class template. The compiler reserves appropriate amount of memory based on the type and size of elements. For instance, an array that can store 10 elements of type int is declared as

```
array<int, 10> marks;
```

Listing 2: An array that can store 10 integers

The indexing of the array elements starts from 0 to size-1 where size is the total number of elements the array can hold.

## 1.1 Looping over an array

We can loop over an array and perform some specific tasks like searching, summing among other using a loop. Consider the following code

```
#include <iostream>
#include <array>
using namespace std;
int main()
{
  array<int, 10> marks;
  for (int i = 0; i < marks.size(); i++)
  {
    marks.at(i) = i * 10;
  }
  for (int i = 0; i < marks.size(); i++)
  {
    cout << marks.at(i) <<" ";
  }
  return 0;
```

```
16 }
```

Listing 3: Looping over an array

Line 6 of the code in Listing 3 creates an array that can store 10 integers using a class template $< array >$.

On line 7, we define a loop that can traverse our array and initialize every element in that array with $i \times 10$ where $i$ is the position of the respective element in the array.

On line 11, we define another loop that traverse through the array, displaying elements in every location as shown in Figure 1.

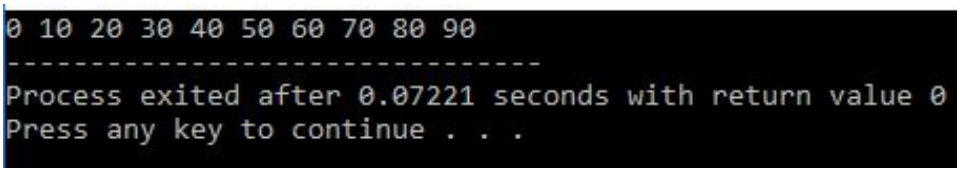


Figure 1: Output of code in Listing 3

As you can notice from code in Listing 3, class template array provides some functions that we can use while manipulating our array

1. size() – Returns the number of elements an array can hold. Size of an array.
2. at(index) – Given an index, it returns the content of an array at a particular index. It has one overload which points to the location of the index given
3. empty() – Check if an array is empty
4. begin() – Return iterator for beginning of array
5. end() – Return an iterator for end of an array
6. fill(value) – Assign all elements in the array a given value
7. max_size() – Return maximum possible length of array

## 1.2 Initialize array at declaration using an initializer list

An initializer list as

```
1 array<int, 5> marks{67, 89, 34, 56, 76};
```

Listing 4: Initializing an array at declaration with initializer list

can be used to initialize every element of an array. An initialization like

```
1 array<int, 5> marks{};//initialize all elements to 0
```

Listing 5: Initializing an array at declaration with initializer list

will initialize all elements of the array to zero.

### 1.3 Static local arrays and Automatic local arrays

A static local variable in a function definition exists for program's duration but is visible only in the functions' body.

A C++ program initializes static local arrays when their declarations are first encountered. If a static array is not explicitly initialized by a programmer, each element of the array is initialized to zero by the compiler when the array is created. Other local (non static) variable are not initialized in the same manner. Consider the code

```
1  #include <iostream>
2  #include <array>
3  void staticArray();
4  void automaticArray();
5  using namespace std;
6  int main()
7  {
8    cout << "First call to each function:\n";
9    staticArray();
10   automaticArray();
11   cout << "\n\nSecond call to each function:\n";
12   staticArray();
13   automaticArray();
14   cout << endl;
15
16   return 0;
17 }
18 void staticArray()
19 {
20   static array < int, 3> array1;
21   cout << "Values on Entering Static Array\n";
22   for (int i = 0; i < array1.size(); i++)
23   {
24     cout << "array1[" << i << "]" <<"\t" << array1.at(i) << "   ";
25   }
26   cout << "\nValues on Exiting Static Array\n";
27   for (int i = 0; i < array1.size(); i++)
28   {
29     cout << "array1[" << i << "]" <<"\t" << (array1.at(i) += 5)<< "   ";
30   }
31 }
32 void automaticArray()
33 {
34   array < int, 3> array2{1, 2, 3};
35   cout << "\n\nValues on Entering Automatic Array\n";
36   for (int i = 0; i < array2.size(); i++)
37   {
38     cout << "array2[" << i << "]" <<"\t" << array2.at(i) << "   ";
39   }
40   cout << "\nValues on Exiting Automatic Array\n";
41   for (int i = 0; i < array2.size(); i++)
42   {
43     cout << "array2[" << i << "]" <<"\t" << (array2.at(i) += 5)<< "   ";
44   }
45 }
```

Listing 6: Static and Automatic Arrays

Static local array on line 20 of Listing 6 initializes every element to zero. When called for first time (line 9), it displays zeros. Line 29 adds five for each element of the static array and display them. On calling the function for second time (line 12), the initial values in the static array is 5 for all the three elements. And after updating it becomes 10 for every element (line 29). If we were to call staticArray() for third time,

initial values will be 10, 10, 10 and values after exiting will be 15, 15, 15.

Automatic array on line 34 initializes array elements to 1, 2, and 3. When called for the first time, it displays 1, 2, 3. Line 43 updates these elements by adding 5 and display them. When called for the second time (line 13), initial values are 1, 2, 3 and final value are 6, 7, 8. Figure 2 shows the output for the code in Listing 6.

```
First call to each function:
Values on Entering Static Array
array1[0]       0   array1[1]   0   array1[2]   0
Values on Exiting Static Array
array1[0]       5   array1[1]   5   array1[2]   5

Values on Entering Automatic Array
array2[0]       1   array2[1]   2   array2[2]   3
Values on Exiting Automatic Array
array2[0]       6   array2[1]   7   array2[2]   8

Second call to each function:
Values on Entering Static Array
array1[0]       5   array1[1]   5   array1[2]   5
Values on Exiting Static Array
array1[0]       10   array1[1]  10   array1[2]  10

Values on Entering Automatic Array
array2[0]       1   array2[1]   2   array2[2]   3
Values on Exiting Automatic Array
array2[0]       6   array2[1]   7   array2[2]   8
```

Figure 2: Output of code in Listing 6

Notice that, automatic array is re-created and re-initialised during each call, while static array is not re-initialised during every call, instead it contains values of the most recent update.

### 1.4 Range based for statement

C++11 range-based for statement allows you to traverse an array without using a counter, hence avoiding the possibility of "stepping out of the array". Thats nice neh? You don't have to implement your own bound checking code. So how do we do this range based for loop. The loop as general format of

```
1 for (type varName : arrayName) //read as for every var in the array
2 {
3   //do somethin
4 }
```

Listing 7: Range based for loop

The code in Listing 10 uses range based for loop to display contents of an array

```
1 #include <iostream>
2 #include <array>
3
4 using namespace std;
5 int main()
```

```
{
  array <int, 5> marks{89, 56, 72, 59, 73};
  for(int m : marks) //range based for loop
  {
    cout << m << "\t ";
  }
  cout << endl;
}
```

Listing 8: Range based for loop

### 1.5 Sorting and Searching arrays

C++11 provides < *algorithm* > library that has functions, that can be used to performing sorting and searching. Listing 9 shows how to sort elements of the marks array and search if a value (72) exist in the array.

```
#include <iostream>
#include <array>
#include <algorithm>
using namespace std;
int main()
{
  array <int, 5> marks{89, 56, 72, 59, 73};
  cout << "Before Sorting\n";
  for(int m : marks) //range based for loop
  {
    cout << m << "\t ";
  }
  //sort
  sort(marks.begin(), marks.end());
  cout << endl;
  cout << "After Sorting\n";
  for(int m : marks) //range based for loop
  {
    cout << m << "\t ";
  }
  cout << "\n\nLets search if a value 72 exist in array marks" << endl;
  //Lets perform some binary search
  bool found = binary_search(marks.begin(), marks.end(), 72); //takes start iterator, stop iterator and a constant value to
      be searched
  cout << 72 << (found ? " was" : "was not") <<" found in marks\n";

}
```

Listing 9: Search and Sort an Array

### 1.6 Multidimensional arrays – using < *array* > class template

Multidimensional array declaration takes the form

```
array< array <type, cols>, rows> marks;
```

Listing 10: Multidimesional array

To declare a multidimensional array of 5 rows 3 columns for integers is

```
array < array <int, 3>, 5> marks;
```

Listing 11: Multidimesional array

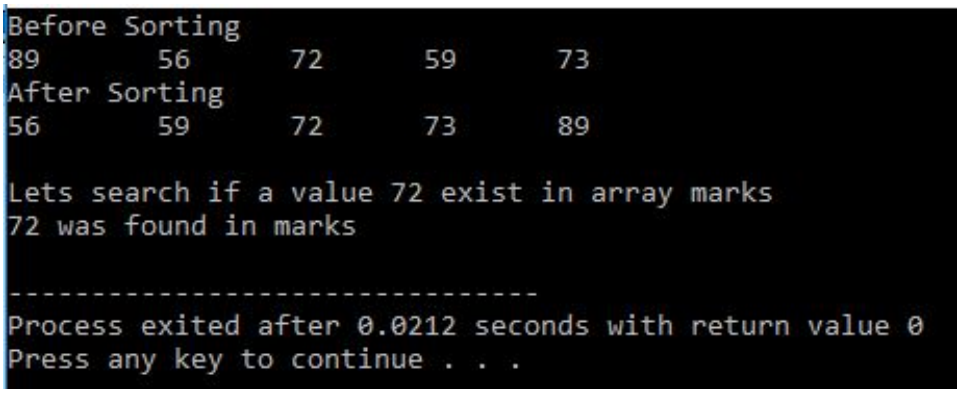


Figure 3: Output of code in Listing 9

### 1.7 Nested range based for loop

We use auto keyword - which tells the compiler to infer (determine) a variable's data type based on the variables initializer value. See the code below

```
#include <iostream>
#include <array>
using namespace std;
int main()
{
  array < array <int, 3>, 5> marks {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
  for(auto row : marks) //row can be any other valid variable name
  {
    for (auto col : row) //col can be any other valid variable name
    {
      cout << col <<"\t";
    }
    cout << endl;
  }
}
```

Listing 12: Multidimesional array – Nested range based for loop

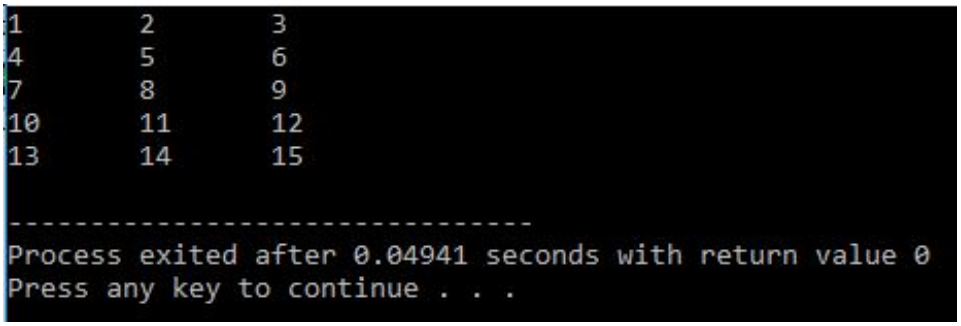


Figure 4: Output of code in Listing 12