



School of Computing and Informatics

BCS 362 – GENERIC PROGRAMMING WITH C++

Chapter 5.1 – Arrays

1 Introduction

It is assumed that the students are conversant with basic array concepts like

1. Array declaration
2. Array indexing
3. 1-D and 2-D arrays

As this has been covered in class, and in the previous modules.

This section presents an introduction to `std::array` define in array library.

Introduced in C++11, `std::array` provides fixed array functionality that won't decay when passed into a function. `std::array` is defined in the array header, inside the `std` namespace. Declaring a `std::array` variable is easy:

```
1 #include <array>
2
3 std::array<int, 3> myArray; // declare an integer array with length 3
```

Just like the native implementation of fixed arrays, the length of a `std::array` must be set at compile time.

`std::array` can be initialized using an initializer lists or uniform initialization:

```
1 std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initialization list
2 std::array<int, 5> myArray2 { 9, 7, 5, 3, 1 }; // uniform initialization
```

Unlike built-in fixed arrays, with `std::array` you can not omit the array length when providing an initializer:

```
1 std::array<int, > myArray = { 9, 7, 5, 3, 1 }; // illegal , array length must be provided
```

You can also assign values to the array using an initializer list

```
1 std::array<int, 5> myArray;
2 myArray = { 0, 1, 2, 3, 4 }; // okay
3 myArray = { 9, 8, 7 }; // okay, elements 3 and 4 are set to zero!
4 myArray = { 0, 1, 2, 3, 4, 5 }; // not allowed, too many elements in initializer list !
```

Accessing array values using the subscript operator works just like you would expect:

```
1 std::cout << myArray[1];
2 myArray[2] = 6;
```

Just like built-in fixed arrays, the subscript operator does not do any bounds-checking. If an invalid index is provided, bad things will probably happen.

`std::array` supports a second form of array element access (the `at()` function) that does bounds checking:

```
1 std::array<int, 5> myArray { 9, 7, 5, 3, 1 };
2 myArray.at(1) = 6; // array element 1 valid, sets array element 1 to value 6
3 myArray.at(9) = 10; // array element 9 is invalid, will throw error
```

In the above example, the call to `array.at(1)` checks to ensure array element 1 is valid, and because it is, it returns a reference to array element 1. We then assign the value of 6 to this. However, the call to `array.at(9)` fails because array element 9 is out of bounds for the array. Instead of returning a reference, the `at()` function throws an error that terminates the program (note: Its actually throwing an exception of type `std::out_of_range` – we cover exceptions in chapter 15). Because it does bounds checking, `at()` is slower (but safer) than `operator[]`.

`std::array` will clean up after itself when it goes out of scope, so theres no need to do any kind of cleanup.

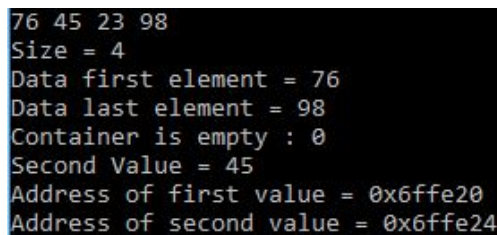
The `size()` function can be used to retrieve the length of the array:

```
1 std::array<double, 5> myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };
2 std::cout << "length: " << myArray.size();
```

Because `std::array` doesnt decay to a pointer when passed to a function, the `size()` function will work even if you call it from within a function. The code below shows some functions and their task in `std::array` manipulation.

```
1 #include<iostream>
2 #include<array>
3 using namespace std;
4
5 int main()
6 {
7     array<int, 4> data = {76, 45, 23, 98};
8     for(auto i = data.begin(); i != data.end(); i++) //we can loop over this array using an iterator
9         cout << *i << " ";
10    cout << endl;
11    cout << "Size = " << data.size() << endl;
12    cout << "Data first element = " << data.front() << endl; //give the element at the first location
13    cout << "Data last element = " << data.back() << endl; //returns last element of the array
14    bool a = data.empty(); //check if container is empty
15    cout << "Container is empty : " << a << endl;
16    //you can swap values in two arrays using swap() see example on vectors
17    cout << "Second Value = " << data.at(1) << endl; //we can access elements using at() function
18    cout << "Address of first value = " << data.begin() << endl;
19    cout << "Address of second value = " << data.begin() + 1 << endl;
20 }
```

Figure 1 shows the output of the code above.



```
76 45 23 98
Size = 4
Data first element = 76
Data last element = 98
Container is empty : 0
Second Value = 45
Address of first value = 0x6ffe20
Address of second value = 0x6ffe24
```

Figure 1: Output of code above