Arrays
(Programming Club 1)

Mihai Enache – M.I.Enache@sms.ed.ac.uk, UG4, School of Informatics
1 October 2018

1. Introduction

We will start our series of weekly sessions about data structures, algorithms and interview questions with what is arguably the simplest of the data structures – the **array**. Broadly speaking, an array is a collection of elements (**values**) identified by the position (**index**) on which they are placed in the array. Unlike the natural language, usually we count the elements of an array starting from 0. Thus, the first element of an array with n elements is placed at index 0 and the last one is found at index n - 1 (see the figure below for an example of an array with 4 elements).

Value:	4	7	-1	7
Index:	0	1	2	3

2. Implementation

As mentioned above, this is a very simple and basic data structure. Hence, it is already implemented for you in most of the programming languages that you will use and you can already use it. However, it is often useful to know how a data structure that you plan to use is implemented.

When an array is created, the number of elements it will hold must be specified. In general, an array can hold only one type of elements. For example, to create an array in C++ we write the following:

```
int myArray[10];
```

This creates an array of size 10 that can store integer values. The **size** of an array **is fixed** and it can't be changed after the array was created. The elements of the array are placed on a continuous area in memory – we will not discuss about this here as this is more advanced stuff that you will learn in your later years. However, because of this property, it is very easy to access an element given the starting position of the array and the index of the element. Hence, if we want to set the value of the 7-th element to 14 we have to type (remember, the array is 0-indexed!) :

```
myArray[6] = 14;
```

Having a fixed size is both an **advantage** and a **disadvantage**. It gives us very fast access to the value of a specified index, since we can easily determine it given the starting place of the array in memory. However, the downside is that we might end up using more memory than we actually need. For example, imagine that you build an application that holds the age of the students that signed up for a given class. The only thing you can assume is that there will be no more than 300 students in a class. Hence, if you decided to use this data structure in your application, you will end up writing something like `int age[300];` - now think about what happens if only 30 people sign up for that class? You will only use the interval [0, 30). Hence 90% of the array that you just created will be of no use.

3. Array operations

Now let’s discuss about what type of operations you can make on your array. We have seen so far that it is easy to **store** a value at a specific index or to **access** the value of that index.

Find - what if we are interested in finding whether a given value exists in the array? Formally, given x, we want to know if there is an index idx such that `myArray[idx]` is x. Unfortunately, if the array has no other properties that you can take advantage of (like being sorted), you will need to try each of its elements until you find the target value. Thus, if the size of an array is n, finding whether a given element exists or not takes n steps in the worst case (for example if the element is not in the array). But you can also be lucky and need to execute only one step (if the element you are looking for is placed just at the beginning of the array for example). However, today and in general we are interested in the worst case scenario, since we want to guarantee some performance for our application.

In contrast, **storing** a value or **accessing** a value given an index takes only one step.

Delete – if we want to delete a specific value from our array but we don’t know its position, we still have to perform a **find** first and then we can actually remove it. Hence, this operation also takes around n steps in the worst case.

See the example below (assuming that we have already created and initialized an array of size n).

```
int sum = 0;
for (int i = 0; i < n; ++i) {
     sum = sum + values[i];
}

int equalToSum = 0;
for (int i = 0; i < n; ++i) {
     if (values[i] == sum) {
          equalToSum = equalToSum + 1;
     }
}

cout << equalToSum << “ elements are equal to the total sum\n”;
```

In this example, we compute the total sum of the array and then traverse again the whole array to see how many elements are equal to the total sum.

4. Resources

1. Arrays in:
a) C++: http://www.cplusplus.com/doc/tutorial/arrays/
b) Java: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
c) Python: https://www.i-programmer.info/programming/python/3942-arrays-in-python.html

2. Array problems:
a) https://www.hackerrank.com/interview/interview-preparation-kit/arrays/challenges
b) https://codeforwin.org/2015/07/array-programming-exercises-and.html