A-Z of Arrays

Whenever we start our journey of Data Structures and Algorithms we are always welcomed by “Arrays”, but most of us do not get the concept of arrays easily. So, let’s demystify what arrays are from 1-D to 2-D arrays and everything beyond.

## What is an array?

In C/C++ array refers to a data type which is a collection of similar types of data elements. An array can have integers, characters, floats etc.

The array elements are allocated continuous memory locations inside RAM and the variable used for declaration stores the base address/starting address of the array.

For e.g.: int arr[5] will have 5 contiguous memory locations and “arr” will store the address of first location. If the address of first location, for example, is 100 then all of the locations will be 100, 104, 108, 112, 116 (addresses).

To access value at any specific location in the array we can use the index of that location. Do note the index of array starts from 0. So, for element at location 5 we’ll write “arr[4]” to access its value.

## Why does array indexing start from 0?

We’ve always learned that array indexing starts from 0 but ever wondered why so? Let’s shed some light on how actually the compiler calculates memory locations for understanding this.

Before jumping to understanding the 0 index mystery, we need to learn about 2 operators the “&” and the “\*” operators.

& => The ampersand operator, also known as “Address of” operator, gives the memory location or address of the variable. For e.g.: &a[2] will give us the address of the 3rd memory block in the array.

\* => The asterisk operator, also know as “Value at” operator, gives us the value at the provided memory location or address. For e.g.: \*104 will give value at address 104 which in our case will be value at a[1].

Now, let’s get back to understanding the mystery of index 0. So, when we write a[i], for any given index i, the compiler internally converts this into the formula of \*(a + i). Then the address is calculated as:

\*(Base address + i \* (Size of data type))

and this gives us the value at the specified index. Since the name of variable, a, stores the base address or value at starting location’s address, when we need to access that value, in the formula we need to do (a + 0). Hence, the index starts from 0.

## 2-D Arrays:

A two-dimensional array or 2-D array is an array where we have rows and columns of data. For e.g., int arr[3][4] will give us an array with 3 rows and 4 columns.

Declaration of 2-D arrays is similar to 1-D arrays, but here we specify both the rows and columns. Example: arr[rows][columns].

## Initializing 2-D arrays:

There are various methods to initialize a 2-D array. Some of the common methods are listed below:

1. Giving all values in single braces => int a[3][2] = {1, 2, 3, 4, 5, 6};
2. Giving values less than total size of array => int a[3][2] = {1, 2, 3, 4}. The last 2 positions will be filled by ‘0’.
3. Giving values more than total size of array => int a[3][2] = {1, 2, 3, 4, 5, 6, 7}. This will throw an error at compile time.
4. Specifying with internal braces => int a[3][2] = {{1, 2}, {3, 4}, {5, 6}};

## Internal architecture of 2-D arrays:

When we declare a 2-D array we expect it to be stored as rows and columns inside the RAM but, in reality, there is no 2-D array inside RAM. All the 2-D arrays are converted into a 1-D array and stored in RAM.

So, how do we get to access values like arr[2][3] if the array stored is actually in 1-D?

This is because of our compiler. The complier, as for 1-D arrays, uses a formula to access the values of the 2-D array. The formula is:

a[i][j] = Base address + i \* (Size of each row) + j \* (Size of data type)

Let’s take an example to understand this. For an array int arr[3][3] and starting index 100, when we write arr[1][2], this is what the compiler calculates:

arr[1][2] = 100 + 1 \* (2) + 2 \* (4) = 112 + 8 = 120. And then we’ll get the value at 120.

So, another notation for a[i][j] can be \*(\*(a + i) + j). Try exploring how this will be resolved to the value at a location.