DAILY DSA | DAY-1 | ARRAYS – PART1|
-GOPALKRISHNA A

Let’s begin with a few self-questions

- Do you keep a list of contacts on your Phone?
- Have you ever seen a leaderboard during competitions?
- Are saving images of particular file sizes on your digital devices?
- Do you have your favorite music playlist track on your smartphone?

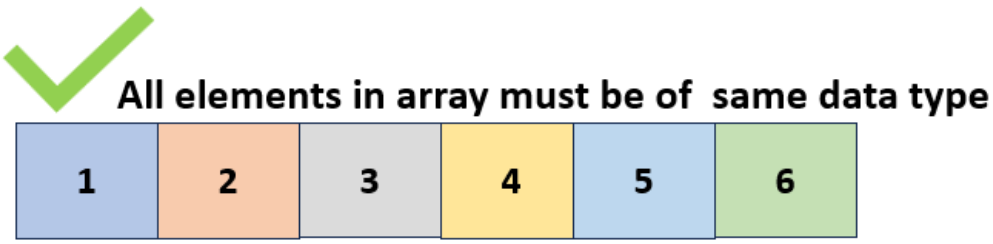
If your answer is "Yes" to any of these questions, then it's almost certain that you've used **arrays,** and knowingly/unknowingly they are part of our daily stuff!

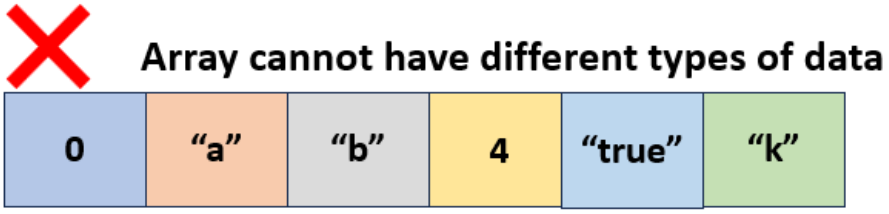
Sounds exciting! Let's dive to understand arrays...

Arrays are one of the most straightforward topics in data structures, yet they are powerful in storing and accessing data.

An array is a structured collection of data elements, each accessible through a numerical index. It stores elements of a consistent data type (Homogeneous), maintaining their order in a contiguous manner.

Arrays take advantage of this "Grid" structure to **store lists of related information in adjacent memory locations** to guarantee extreme efficiency in finding those values



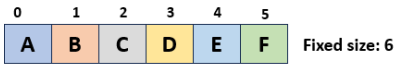


Types of Arrays:

1. **Static array:** An array is allocated upfront as a single block of memory based on the number of elements and type of data we want the array to hold. This allows you to read & write elements into the array efficiently.

STATIC ARRAY

Stores things in order (i.e. elements are indexed), has quick lookups and a fixed size



	Avg & Worst case
Space	O(N)
Index (Lookup)	O(1)
Search	O(N)
Append	O(1)
Insert	O(N)
Delete	O(N)

Pros

- Highly efficient for **quick lookups**

Cons

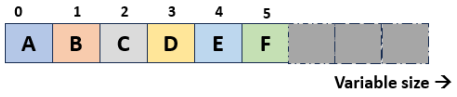
- Inefficient at adding/removing elements. **Adding/removing requires shifting all elements** to accommodate for the added or removed element.
- **Fixed size**

2. **Dynamix arrays:** In this type of array, memory is allocated at run time but does not have a fixed size. Dynamic arrays automatically increase or decrease their size by allocating a new copy of the array when it begins to run out of space.

    - Dynamic arrays guarantee **better-amortized performance** by only performing the costly operations when necessary.

DYNAMIC ARRAY

An array that dynamically changes size, i.e it expands (contracts) when its current size is too small (big)



	Avg & Worst case
Space	O(N)
Index (Lookup)	O(1)
Search	O(N)
Append	O(1) O(N) in worst case
Insert	O(N)
Delete	O(N)

Pros

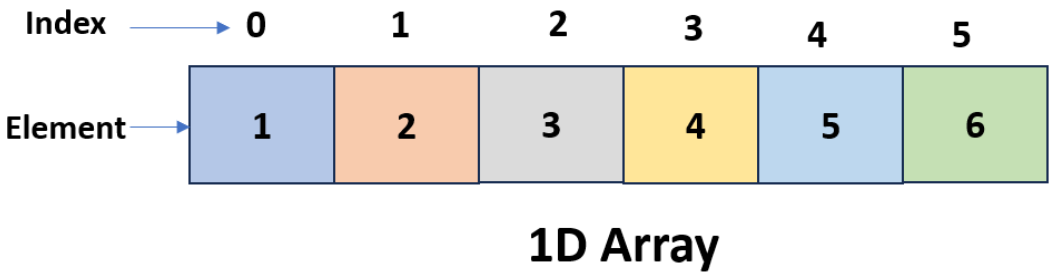
- Highly efficient for **quick lookups**
- Variable size and can accommodate a changing (dynamic) number of items

Cons

- Inefficient at adding/deleting elements
- Worst case for appending element is O(n) - cost of resizing

Representation of Array:

1. **1- Dimensional arrays (1-D):** Array where elements are stored one after another

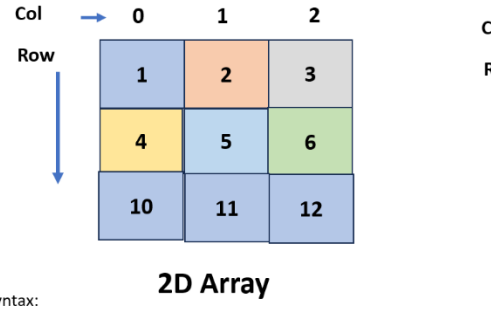


1D Array

Syntax: data_type array_name[array_size]
Where:

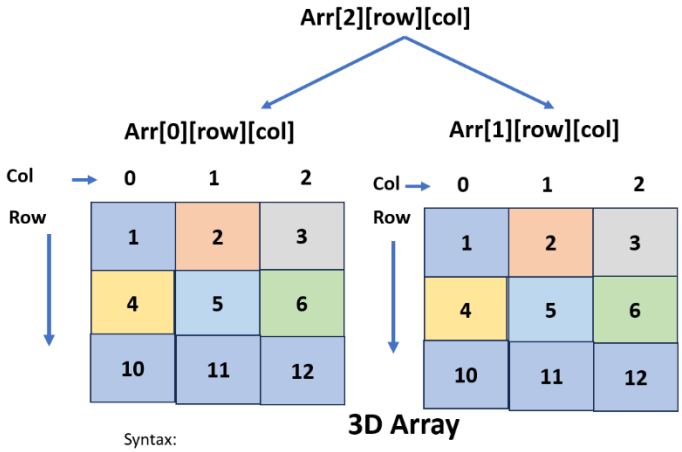
- data_type: Type of data of each array block
- array_name: Name of the array using which we can refer to it
- Array_size: Number of blocks of memory array going to have

2. **Multi-dimensional arrays:** Array of arrays or a matrix consisting of rows and columns.



2D Array

Syntax:
data_type array_name[Size of 1st dimension] [Size of 2nd dimension]



3D Array

Syntax:
data_type array_name[Size of 1st dimension] [Size of 2nd dimension] [size of 3rd dimension]

Calculating memory usage:
To calculate the memory footprint of an array, simply multiply the size of the array by the size of the data type

- Memory usage of an array that contains two thousand 32-bit integers

  2000 * 32 bits = 2000 * 4 bytes = 4Kb

- Memory usage of an array that contains one hundred 10-character strings

  100* 10 chars = 100*10*1 byte = 1Kb

Common array operations

Insert an item, remove an item, update an item, Find an item, Loop over an array, Copy an array, copy part of an array, sort an array, reverse an array, Swap two items, Filter an array