Table of contents

Arrays

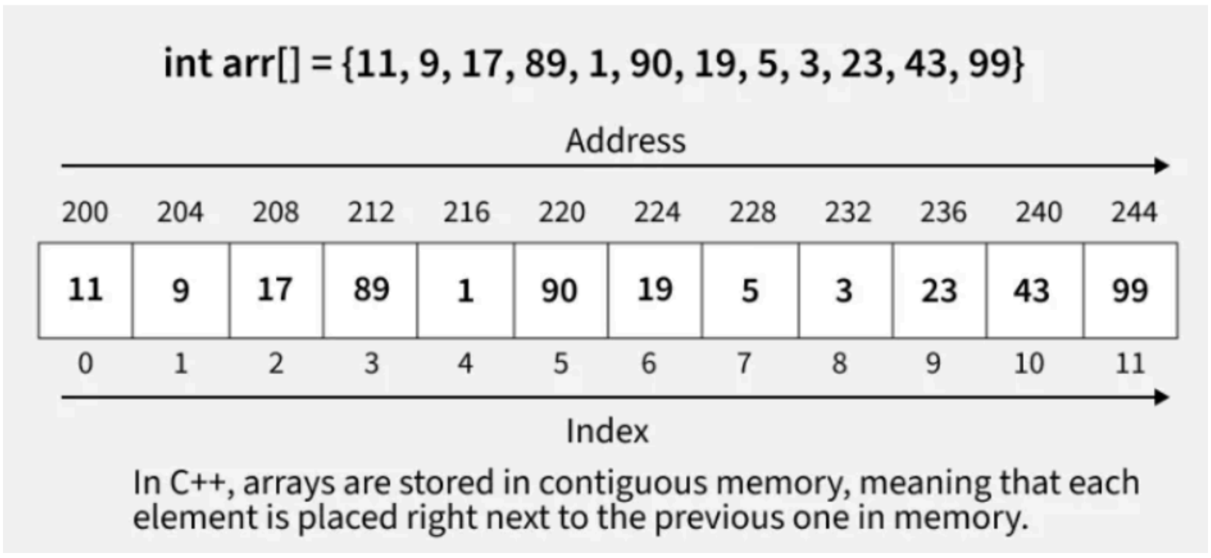
Array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

Basic Terminologies of Array

- **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
- **Array element:** Elements are items stored in an array and can be accessed by their index.
- **Array Length:** The length of an array is determined by the number of elements it can contain.

Memory representation of an Array

In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory. This allows for efficient access and manipulation of elements.



In C++, arrays are stored in contiguous memory, meaning that each element is placed right next to the previous one in memory.

Declaration of Array

Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations:

C++

```
// This array will store integer type element
int arr[5];

// This array will store char type element
char arr[10];

// This array will store float type element
float arr[20];
```

Initialization of Array

Arrays can be initialized in different ways in different languages. Below are some language-specific array initializations:

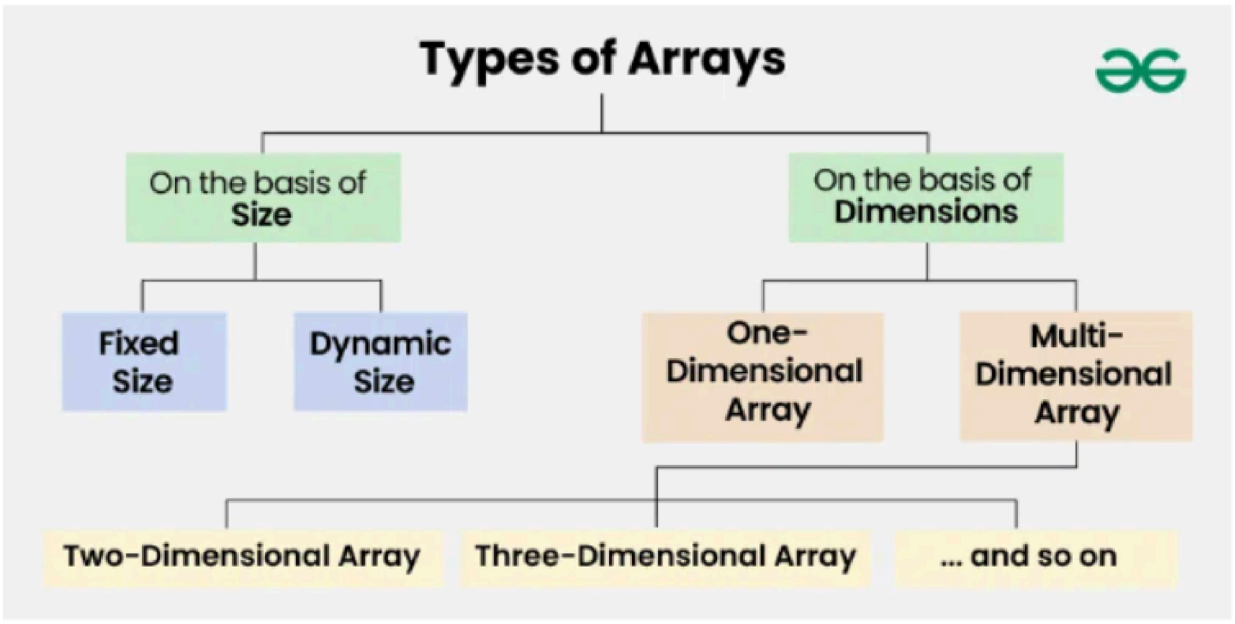
C++

```
int arr[] = { 1, 2, 3, 4, 5 };
char arr[5] = { 'a', 'b', 'c', 'd', 'e' };
float arr[10] = { 1.4, 2.0, 24, 5.0, 0.0 };
```

Types of Arrays

Arrays can be classified in two ways:

- On the basis of Size
- On the basis of Dimensions



Types of Arrays on the basis of Size

1. Fixed Sized Arrays

We cannot alter or update the size of this array. Here only a fixed size (i,e. the size that is mentioned in square brackets **[]**) of memory will be allocated for storage. In case, we don't know the size of the array then if we declare a larger size and store a lesser number of elements will result in a wastage of memory or we declare a lesser size than the number of elements then we won't get enough memory to store all the elements. In such cases, static memory allocation is not preferred.

C++

```
// Method 1 to create a fixed sized array.
// Here the memory is allocated at compile time.
int arr[5];
// Another way (creation and initialization both)
int arr2[5] = {1, 2, 3, 4, 5};

// Method 2 to create a fixed sized array
// Here memory is allocated at run time (Also
// known as dynamically allocated arrays)
int *arr = new int[5];
```

2. Dynamic Sized Arrays

The size of the array changes as per user requirements during execution of code so the coders do not have to worry about sizes. They can add and removed the elements as per the need. The memory is mostly dynamically allocated and de-allocated in these arrays.

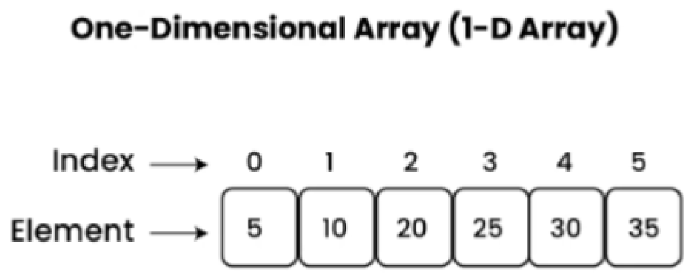
```
C++

#include<vector>

// Dynamic Integer Array
vector<int> v;
```

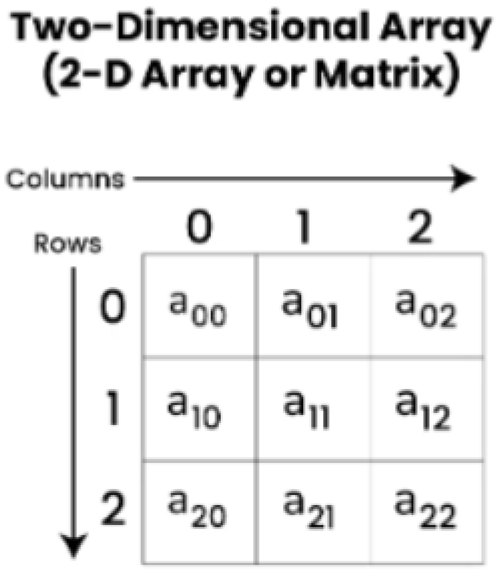
Types of Arrays on the basis of Dimensions

1. One-dimensional Array(1-D Array): You can imagine a 1d array as a row, where elements are stored one after another.



2. Multi-dimensional Array: A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.

- **Two-Dimensional Array(2-D Array or Matrix):** 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



Operations on Array

1. Array Traversal

Array traversal refers to the process of accessing and processing each element of an array sequentially. This is one of the most fundamental operations in programming, as arrays are widely used data structures for storing multiple elements in a single variable.

Types of Array Traversal

Array traversal can be done in multiple ways based on the requirement:

1. **Sequential (Linear) Traversal**
   - This is the most common way of traversing an array.
   - It involves iterating through the array one element at a time from the first index to the last.
   - Used for printing elements, searching, or performing calculations (such as sum or average).

C++

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

```
for (int i=0 ; i<5 ; i++){
        cout<< arr[i] << endl;
}
```

2. **Reverse Traversal**
   - Instead of starting from index 0, the traversal begins from the last element and moves towards the first.
   - This is useful in cases where we need to process elements from the end.

C++

```
int arr[5] = { 1, 2, 3, 4, 5 };
for (int i=4 ; i>=0 ; i--){
        cout<< arr[i] << endl;
}
```

2. Insertion in Array

Insertion in an array refers to the process of adding a new element at a specific position while maintaining the order of the existing elements. Since arrays have a fixed size in static implementations, inserting an element often requires shifting existing elements to make space.

How Insertion Works in an Array?

Arrays are stored in contiguous memory locations, meaning elements are arranged in a sequential block. When inserting a new element, the following happens:

1. **Identify the Position**: Determine where the new element should be inserted.
2. **Shift Elements**: Move the existing elements one position forward to create space for the new element.
3. **Insert the New Element**: Place the new value in the correct position.

4. **Update the Size (if applicable)**: If the array is dynamic, its size is increased.

For example, if we have the array:

arr = [10, 20, 30, 40, 50]

and we want to insert 25 at index 2, the new array will be:

arr = [10, 20, 25, 30, 40, 50]

Here, elements 30, 40, and 50 have shifted right to make space.

Types of Insertion

1. Insertion at the Beginning (Index 0)

- Every element must shift one position right.
- This is the least efficient case for large arrays as it affects all elements.

2. Insertion at a Specific Index

- Elements after the index shift right.
- If the index is in the middle, half of the array moves.

3. Insertion at the End

- The simplest case since no shifting is required.
- Used in dynamic arrays where size increases automatically.

3. Deletion in Array

Deletion in an array refers to the process of removing an element from a specific position while maintaining the order of the remaining elements. Unlike linked lists, where deletion is efficient, removing an element from an array requires shifting elements to fill the gap.

How Deletion Works in an Array?

Since arrays have contiguous memory allocation, deleting an element does not reduce the allocated memory size. Instead, it involves:

1. **Identify the Position**: Find the index of the element to be deleted.
2. **Shift Elements**: Move the elements after the deleted element one position to the left.
3. **Update the Size (if applicable)**: If using a dynamic array, the size might be reduced.

For example, consider the array:

arr = [10, 20, 30, 40, 50]

If we delete the element 30 (index 2), the new array will be:

arr = [10, 20, 40, 50]

Here, elements 40 and 50 shifted left to fill the gap.

Types of Deletion

1. Deletion at the Beginning (Index 0)

- Every element shifts left by one position.
- This is the most expensive case as it affects all elements.

2. Deletion at a Specific Index

- Only elements after the index shift left.
- If the index is in the middle, half of the array moves.

3. Deletion at the End

- The simplest case since no shifting is required.
- The size of the array is reduced (in dynamic arrays).

4. Searching in Array

Searching in an array refers to the process of finding a specific element in a given list of elements. The goal is to determine whether the element exists in the array and, if so, find its index (position).

Searching is a fundamental operation in programming, as it is used in data retrieval, filtering, and processing.

Example:
Consider an array:

arr = [10, 20, 30, 40, 50]

If we search for 30, the algorithm will:

1. Compare 10 with 30 → No match.
2. Compare 20 with 30 → No match.
3. Compare 30 with 30 → **Match found at index 2.**

C++ Built-In Vector Functions

Function	Syntax	Description
size()	nums.size(); //for vectors	Give the size of the vector.

sort()	sort(arr, arr + n); // for raw arrays
	sort(v.begin(), v.end()); // for vector
		Sorts elements in ascending order.

reverse()	reverse(arr, arr + n);
	reverse(v.begin(), v.end());
		Reverses elements of the array/vector.

push_back()	nums.push_back(9); //For vectors	Adds an element at the end.
pop_back()	nums.pop_back(); //For vectors	Removes the element at the end.

accumulate()	accumulate(arr, arr + n, 0);
	accumulate(v.begin(), v.end(), 0);
		Returns the sum of elements from start to end.

find()	find(arr, arr + n, 10);
	find(v.begin(), v.end(), 7);
		Returns an iterator to the element if found, else returns `end()`.

fill()	fill(arr, arr + n, 0);
	fill(v.begin(), v.end(), -1);
		Sets all elements to a specific value.