Arrays

Dr. Gopal Chandra Jana

Introduction to Array

*An **array** is a collection of items of the same variable type stored that are stored at contiguous memory locations. It's one of the most popular and simple data structures and is often used to implement other data structures. Each item in an array is indexed starting with 0.*

Basic terminologies of array

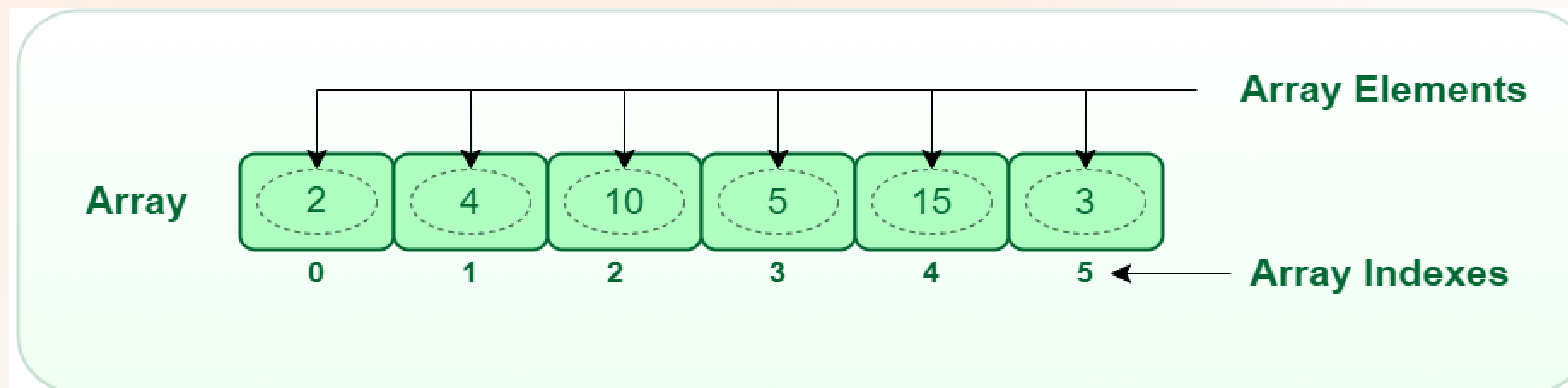
Array Index: In an array, elements are identified by their indexes. Array index starts from 0.

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

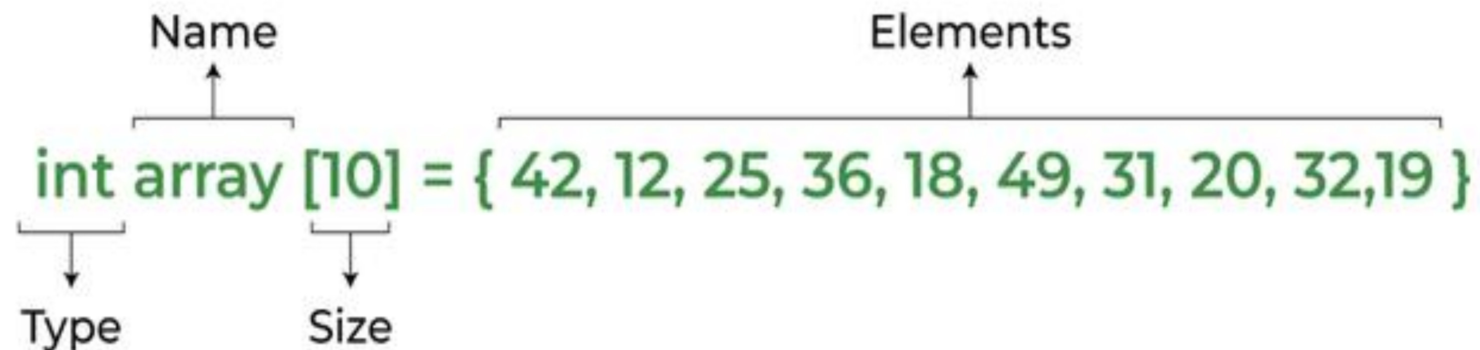
Representation of Array:

The representation of an array can be defined by its declaration. A declaration means allocating memory for an array of a given size.



Declaration of Array

```
int arr[5];      // This array will store integer type element
char arr[10];    // This array will store char type element
float arr[20];   // This array will store float type element
```



However, the above declaration is **static** or **compile-time** memory allocation, which means that the array element's memory is allocated when a program is compiled.

Here only a fixed size (i,e. the size that is mentioned in square brackets **[]**) of memory will be allocated for storage, but don't you think it will not be the same situation as we know the size of the array every time, there might be a case where we don't know the size of the array.

If we declare a larger size and store a lesser number of elements will result in a wastage of memory or either be a case where we declare a lesser size then we won't get enough memory to store the rest of the elements. In such cases, static memory allocation is not preferred.

Basics of Array

Syntax:

Datatype Array_Name[size];
Datatype Array_Name[row][column];
Datatype Array_Name [depth/Layers] [row] [column];

Operations:

Insert, Delete, Search, Sort, Traverse, Rotate, Merge..
Add, Sub, Mul, Div, Transpose, Inverse…

Arrays in Memory:

1. 1D – Contiguous
2. Multidimensional – Row Major/Column Major

KEY POINT	ARRAY
MEMORY	Array uses adjacent/ contiguous memory locations for data storage. It is the static memory allocation technique.
MEMORY UTILIZATION	It is poor in memory utilization.
DECLARATION	At the beginning of the program.
SIZE	Once the size is declared, it will be fixed during the lifetime.
FLEXIBILITY	It does not allow the data items to be inserted or deleted without affecting the other data items.
EXAMPLE	A record of hundred students.

Array Operations

Insertion: Inserting a new element in an array.

```
#include <stdio.h>
 int insert(int arr[], int n, int x, int cap, int pos)
{
	if(n == cap)
		return n;

	int idx = pos - 1;

	for(int i = n - 1; i >= idx; i--)
	{
		arr[i + 1] = arr[i];
	}

	arr[idx] = x;
	return n + 1;
}
```

```
int main() {

	int  arr[5], cap = 5, n = 3;

	arr[0] = 5; arr[1] = 10; arr[2] = 20;

	printf("Before Insertion \n");

	for(int i=0; i < n; i++)
	{
		printf("%d ",arr[i]);
	}

	printf("\n");
```

```
int  x = 7, pos = 2;

	n = insert(arr, n, x, cap, pos);

	printf("After Insertion \n");

	for(int i=0; i < n; i++)
	{
			printf("%d ",arr[i]);
	}

}
```

Arrays

Deletion: Deleting an element from the array.

```
#include <stdio.h>
 int delete(int arr[], int n, int x, int cap)
{
        int idx=-1;
        for(int i=0; i<=cap; i++)
                if  (arr[i] == x)
                        idx=i;
        if (idx == -1)
                return -1;

        for(int i = idx+1; i <cap;  i++)
        {
                arr[i - 1] = arr[i];
        }

        return n - 1;
}
```

Searching: Search for an element in the array.

```
#include <stdio.h>

int search(int arr[], int n, int x)
{
        for(int i = 0; i < n; i++)
        {
                if(arr[i] == x)
                        return i;
        }
        return -1;
}

int main() {

        int arr[] = {20, 5, 7, 25}, x = 5;

    printf("%d ",search(arr, 4, x));
}
```

Arrays

Advantages of using Arrays:

- Arrays allow random access to elements. This makes accessing elements by position faster.
- Arrays have better cache locality which makes a pretty big difference in performance.
- Arrays represent multiple data items of the same type using a single name.
- Arrays store multiple data of similar types with the same name.
- Array data structures are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

Disadvantages of Array:

- As arrays have a fixed size, once the memory is allocated to them, it cannot be increased or decreased, making it impossible to store extra data if required. An array of fixed size is referred to as a static array.
- Allocating less memory than required to an array leads to loss of data.
- An array is homogeneous in nature so, a single array cannot store values of different data types.
- Arrays store data in contiguous memory locations, which makes deletion and insertion very difficult to implement. This problem is overcome by implementing linked lists, which allow elements to be accessed randomly.

Application of Array:

- They are used in the implementation of other data structures such as array lists, heaps, hash tables, vectors, and matrices.
- Database records are usually implemented as arrays.
- It is used in lookup tables by computer.

Dynamic Arrays: Vector/ArrayList/List

Vector in C++ STL is a class that represents a dynamic array. The advantages of vector over normal arrays are,

- **We do not need to pass size as an extra parameter when we pass vector.**
- **Vectors have many in-built functions for erasing an element, inserting an element etc.**
- **Vectors support dynamic sizes, we do not have to initially specify the size of a vector. We can also resize a vector.**
- **There are many other functionalities vector provide.**

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.

In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.

Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Dynamic Arrays: Vector/ArrayList/List

Dynamic array- also called **resizable array**, dynamic table, or array list, is a random access, variable-size list data structure that allows elements to be added or removed.

VECTORS

- vector<int> iv; //creates 0-length int vector
- vector<char> cv(5); //creates 5-element char vector
- vector<char> cv(5,’x’); // initialized too
- vector<int> iv2 (iv); // creates int vector from an int vector.

-**size()** - Provides the number of elements

-**push_back()** - Appends an element to the end

-**pop_back()** - Erases the last element

-**begin()** - Provides reference to first element

-**end()** - Provides reference to end of vector

-**rbegin()** - Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element.

-**rend()** - Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

Dynamic Arrays: Vector/ArrayList/List

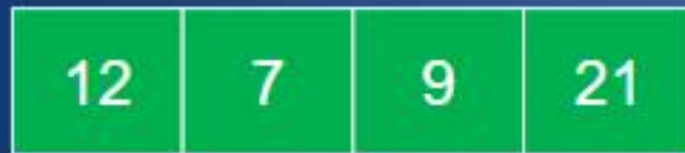
begin() – Returns an iterator pointing to the first element in the vector.
end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector.
size() – Returns the number of elements in the vector.
capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
empty() – Returns whether the container is empty.
push_back() – It push the elements into a vector from the back.
pop_back() – It is used to pop or remove elements from a vector from the back.
insert() – It inserts new elements before the element at the specified position.
erase() – It is used to remove elements from a container from the specified position or range.
swap() – It is used to swap the contents of one vector with another vector of same type and size.
clear() – It is used to remove all the elements of the vector container.
emplace() – It extends the container by inserting new element at position.
emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector.
-**rbegin()** -Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element.
-**rend()** - Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

Vector container

int array[5] = {12, 7, 9, 21, 13 };

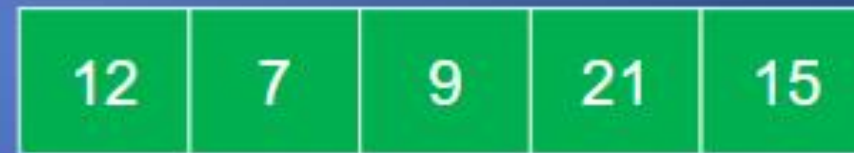
Vector<int> v(array,array+5);

v.pop_back();



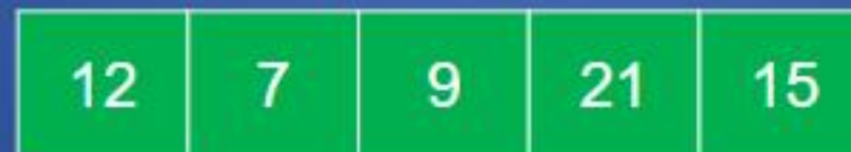
13

v.push_back(15);



...

0 1 2 3 4



v.begin();

v[3]

Dynamic Arrays: Vector

```
// C++ program to illustrate the above
functions

#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int> v;
// Push elements
   for (int i = 1; i <= 5; i++)
      v.push_back(i);
   cout << "Size : " << v.size();

 // checks if the vector is empty or not
   if (v.empty() == false)
      cout << "\nVector is not empty";
   else
      cout << "\nVector is empty";
cout << "\nOutput of begin and end: ";
   for (auto i = v.begin(); i != v.end(); ++i)
      cout << *i << " ";
// inserts at the beginning
   v.emplace(v.begin(), 5);
   cout << "\nThe first element is: " << v[0];

 // Inserts 20 at the end
   v.emplace_back(20);
   int n = v.size();
   cout << "\nThe last element is: " << v[n - 1];

// erases the vector
   v.clear();
   cout << "\nVector size after erase(): " << v.size();

   return 0;
}
```

```
Size : 5
Vector is not empty
Output of begin and end: 1 2 3 4 5
The first element is: 5
The last element is: 20
Vector size after erase(): 0
```

```
#include <iostream>
#include <vector>
 using namespace std;

int main()
{
   vector <int> g1;
   vector <int> :: iterator i;
   vector <int> :: reverse_iterator ir;

   for (int i = 1; i <= 5; i++)
            g1.push_back(i);
   cout << "Output of begin and end\t:\t";
   for (i = g1.begin(); i != g1.end(); ++i)
            cout << *i << '\t';
   cout << endl << endl;

   cout << "Output of rbegin and rend\t:\t";
   for (ir = g1.rbegin(); ir != g1.rend(); ++ir)
            cout << '\t' << *ir;
    return 0;
}
```

```
Output of begin and end :       1       2       3       4       5

Output of rbegin and rend  :    5       4       3       2       1
```

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int> g1;

   for (int i = 1; i <= 10; i++)
      g1.push_back(i * 10);

   cout << "Reference operator [g] : g1[2] = " << g1[2];
   cout << endl;
   cout << "at : g1.at(4) = " << g1.at(4);
   cout << endl;
   cout << "front() : g1.front() = " << g1.front();
   cout << endl;
   cout << "back() : g1.back() = " << g1.back();
   cout << endl;

   return 0;
}
```

```
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
```

Introduction to Array Manipulation Problems

1. ## Commonly Tested Skills

   Understand the array manipulation skills regularly assessed in coding interviews, such as **element swapping, reversing, and rotating**.

2. ## Real-World Applications/Competitive coding

   Discover how array manipulation problems relate to real-world scenarios, **including data processing, image manipulation, and algorithm optimization**.

3. ## Importance of Efficiency

   Learn why **optimizing performance** is crucial when solving array manipulation problems, and **explore techniques to improve time and space complexity**.

Example-1: Write a program to print all the LEADERS in the array. An element is a leader if it is greater than all the elements to its right side. And the rightmost element is always a leader.

Input: *arr[] = {16, 17, 4, 3, 5, 2},*
Output: *17, 5, 2*

Input: *arr[] = {1, 2, 3, 4, 5, 2},*
Output: *5, 2*

Example-2:

Given the arrival and departure times of all trains that reach a railway station, the task is to find the minimum number of platforms required for the railway station so that no train waits. We are given two arrays that represent the arrival and departure times of trains that stop.

Input: *arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}, dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}*
Output: *3*

Input: *arr[] = {9:00, 9:40}, dep[] = {9:10, 12:00}*
Output: *1*

Naïve Approach:

- The idea is to **take every interval one by one** and
- find the number of **intervals that overlap** with it.
- Keep track of the **maximum** number of intervals that **overlap** with an interval.
- Finally, return the **maximum value**.

```
for (int i = 0; i < n; i++) {
        // Initially one platform is needed
        plat_needed = result = 1;
        for (int j = 0; j < n; j++) {
                if (i != j)
                // Increment plat_needed when there is an overlap
                        if (arr[i] >= arr[j] && dep[j] >= arr[i])
                        plat_needed++;
                }

                // Update the result
                result = max(plat_needed, result);
        }
```

Time Complexity: $O(n^2)$, Two nested loops traverse the array.
Auxiliary space: O(1), As no extra space is required.

Optimized Solution-1:

1. Store the arrival time and departure time in array **arr** and **sort** this array based on **arrival time**
2. Declare a **priority queue(min-heap)** and store the departure time of the first train and also declare a counter **count** and initialize it with 1.
3. Iterate over **arr** from **1** to **n-1**

   check if the **arrival time** of the current train is **less than or equal** to the **departure time** of the previous train which is kept on top of the priority queue

   a) If true, then push the **new departure time** and **increment** the counter **count**
   b) otherwise, we **pop()** the **departure time**
   c) push **new departure time** in the priority queue
4. Finally, return the **count**.

Time Complexity: O(N*log(N)), Heaps take log(n) time for pushing element for n elements.
Auxiliary Space: O(N), Space required by heap to store the element.

Optimized Solution-2:

*The idea is to consider all events in sorted order. Once the events are in sorted order, trace the number of trains at any time **keeping track of trains that have arrived, but not departed**.*

arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}, dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

All events are sorted by time.

Total platforms at any time can be obtained by subtracting total departures from total arrivals by that time.

Time	***Event Type***	***Total Platforms Needed at this Time***
9:00	*Arrival*	*1*
9:10	*Departure*	*0*
9:40	*Arrival*	*1*
9:50	*Arrival*	*2*
11:00	*Arrival*	*3*
11:20	*Departure*	*2*
11:30	*Departure*	*1*
12:00	*Departure*	*0*
15:00	*Arrival*	*1*
18:00	*Arrival*	*2*
19:00	*Departure*	*1*
20:00	*Departure*	*0*

*Minimum Platforms needed on railway station = Maximum platforms needed at any time = **3***

Can we have the solution in linear time and constant space?

Yes, using Sweep Line Algorithm

Time Complexity: O(N * log N), One traversal O(n) of both the array is needed after sorting O(N * log N).

Auxiliary space: O(1), As no extra space is required.

Example-2: Given an array **arr[]** of size **n** and an integer **X**. Find if there's a triplet in the array which sums up to the given integer **X**.

Input: *array = {12, 3, 4, 1, 6, 9}, sum = 24;*
Output: *12, 3, 9*
Explanation: *There is a triplet (12, 3 and 9) present in the array whose sum is 24.*
Input: *array = {1, 2, 3, 4, 5}, sum = 9*
Output: *5, 3, 1*
Explanation: *There is a triplet (5, 3 and 1) present in the array whose sum is 9.*

```
// Fix the first element as A[i]
    for (int i = 0; i < arr_size - 2; i++) {

        // Fix the second element as A[j]
        for (int j = i + 1; j < arr_size - 1; j++) {

            // Now look for the third number
            for (int k = j + 1; k < arr_size; k++) {
                if (A[i] + A[j] + A[k] == sum) {
                    printf("Triplet is %d, %d, %d",
                           A[i], A[j], A[k]);
                    return true;
                }
            }
        }
    }
```

Example-2: Given an array **arr[]** of size **n** and an integer **X**. Find if there's a triplet in the array which sums up to the given integer **X**.

Input: *array = {12, 3, 4, 1, 6, 9}, sum = 24;*
Output: *12, 3, 9*
Explanation: *There is a triplet (12, 3 and 9) present in the array whose sum is 24.*
Input: *array = {1, 2, 3, 4, 5}, sum = 9*
Output: *5, 3, 1*
Explanation: *There is a triplet (5, 3 and 1) present in the array whose sum is 9.*

- Sort the given array.
- Loop over the array and fix the first element of the possible triplet, **arr[i]**.
- Then fix two pointers, one at **i + 1** and the other at **n – 1**. And look at the sum,
  - If the sum is smaller than the required sum, increment the first pointer.
  - Else, If the sum is bigger, Decrease the end pointer to reduce the sum.
  - Else, if the sum of elements at two-pointer is equal to given sum then print the triplet and break.