GeeksforGeeks

Over 800 Students, Working Professionals have earned more than 70 Lakhs

Know more

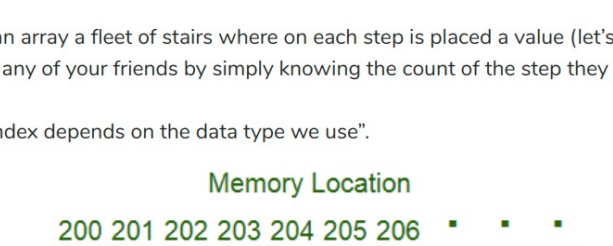
We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

− Introduction to Arrays

An array is a collection of items of same data type stored at contiguous memory locations. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

For simplicity, we can think of an array a fleet of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on.

Remember: "Location of next index depends on the data type we use".



The above image can be looked as a top-level view of a staircase where you are at the base of staircase. Each element can be uniquely identified by their index in the array (in a similar way as you could identify your friends by the step on which they were on in the above example).

Defining an Array: Array definition are similar to defining any other variable. There are two things need to be kept in mind, **data type of the array elements** and the **size** of the array. The size of the array is fixed and the memory for an array needs to be allocated before use, size of an array cannot be increased or decreased dynamically.

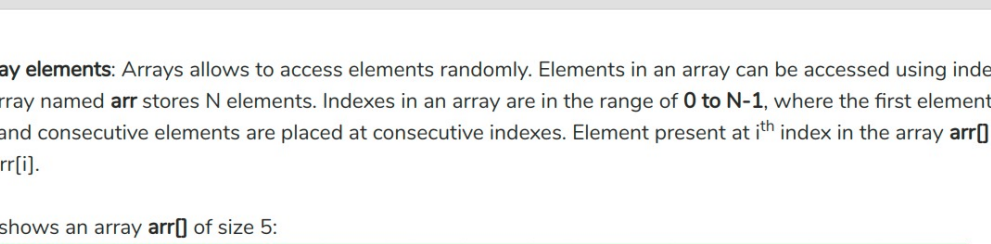
Generally, arrays are declared as:

```
dataType arrayName[arraySize];

An array is distinguished from a normal variable
by brackets [ and ].
```

Accessing array elements: Arrays allows to access elements randomly. Elements in an array can be accessed using indexes. Suppose an array named **arr** stores N elements. Indexes in an array are in the range of **0 to N-1**, where the first element is present at 0-th index and consecutive elements are placed at consecutive indexes. Element present at i^{th} index in the array **arr[]** can be accessed as arr[i].

Below image shows an array **arr[]** of size 5:



Advantages of using arrays:

- Arrays allow random access of elements. This makes accessing elements by position faster.
- Arrays have better cache locality that can make a pretty big difference in performance.

Examples:

```
// A character array in C/C++/Java

char arr1[] = {'g', 'e', 'e', 'k', 's'};


// An Integer array in C/C++/Java

int arr2[] = {10, 20, 30, 40, 50};


// Item at i'th index in array is typically accessed

// as "arr[i]".  For example arr1[0] gives us 'g'

// and arr2[3] gives us 40.
```

Searching in an Array

Searching an element in an array means to check if a given element is present in an array or not. This can be done by accessing elements of the array one by one starting from the first element and checking if any of the element matches with the given element.

We can use loops to perform the above operation of array traversal and access the elements using indexes.

Suppose the array is named **arr[]** with size **N** and the element to be searched is referred as **key**. Below is the algorithm to perform to the search operation in the given array.

```
for(i = 0; i < N; i++)
{
    if(arr[i] == key)
    {
        print "Element Found";
    }
    else
    {
        print "Element not Found";
    }
}
```

Time Complexity of this search operation will be O(N) in the worst case as we are checking every element of the array from 1st to last, so the number of operations is N.

− Insertion and Deletion in Arrays

Insertion in Arrays

Given an array of a given size. The task is to insert a new element in this array. There are two possible ways of inserting elements in an array:

1. Insert elements at the end of the array.
2. Insert element at any given index in the array.

Special Case: A special case is needed to be considered is that whether the array is already full or not. If the array is full, then the new element can not be inserted.

Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len* number of elements already present in this array.

- **Insert an element K at end in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,
  ```
  if(len < N)
      // space left
  else
      // array is full
  ```
  If there is space left for the new element, insert it directly at the end at position **len + 1** and index **len**:
  ```
  arr[len] = k;
  ```
  Time Complexity of this insert operation is constant, i.e. O(1) as we are directly inserting the element in a single operation.
- **Insert an element K at position, pos in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,
  ```
  if(len < N)
      // space left
  else
      // array is full
  ```
  Now, if there is space left, the element can be inserted. The index of the new element will be **idx = pos - 1**.

  Now, before inserting the element at the index idx, shift all elements from the index idx till end of the array to the right by 1 place. This can be done as:
  ```
  for(i = len-1; i >= idx; i--)
  {
      arr[i+1] = arr[i];
  }
  ```
  After shifting the elements, place the element K at index idx.
  ```
  arr[idx] = K;
  ```
  Time Complexity in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the right.

Deletion in Arrays

To delete a given element from an array, we will have to first search the element in the array. If the element is present in the array then delete operation is performed for the element otherwise the user is notified that the array does not contains the given element.

Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len* number of elements already present in this array.

Deleting an element K from the array arr[]: Search the element K in the array arr[] to find the index at which it is present.

```
for(i = 0; i < N; i++)
{
    if(arr[i] == K)
        idx = i; return;
    else
        Element not Found;
}
```

Now, to delete the element present at index **idx**, left shift all of the elements present after *idx* by one place and finally reduce the length of the array by 1.

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

len = len-1;
```

Time Complexity in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the left.

− Array Rotation

As the term rotation signifies, array rotation means to rotate the elements of an array by given positions.

Consider the below array:

1 2 3 4 5 6 7

The above array is rotated counter-clockwise(towards left) by 2 elements. After rotation, the array will be:

3 4 5 6 7 1 2

Visually, the process of counter clock-wise array rotation(rotated by say K elements) looks like:

- Shift all elements after K-th element to the left by K positions.
- Fill the K blank spaces at the end of the array by first K elements from the original array.

Note: The similar approach can also be applied for clockwise array rotation.

Implementations

- **Simple Method**: The simplest way to rotate an array is to implement the above visually observed approach by using extra space.
  1. Store the first K elements in a temporary array say temp[].
  2. Shift all elements after K-th element to the left by K positions in the original array.
  3. Fill the K blank spaces at the end of the original array by the K elements from the temp array.
  ```
  Say, arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2
  1) Store first K elements in a temp array
     temp[] = [1, 2]
  2) Shift rest of the arr[]
     arr[] = [3, 4, 5, 6, 7, 6, 7]
  3) Store back the K elements from temp
     arr[] = [3, 4, 5, 6, 7, 1, 2]
  ```
  Time Complexity: O(N), where N is the number of elements in the array.
  Auxiliary Space: O(K) where K is the number of places by which elements will be rotated.
- **Another Method (Without extra space)**: We can also rotate an array by avoiding the use of temporary array. The idea is to rotate the array one by one K times.
  ```
  leftRotate(arr[], d, n)
  start
    For i = 0 to i < d
      Left rotate all elements of arr[] by one
  end
  ```
  To rotate an array by 1 position to the left:
  1. Store the first element in a temporary variable say temp.
  2. Left shift all elements after the first element by 1 position. That is, move arr[1] to arr[0], arr[2] to arr[1] and so on.
  3. Initialize arr[N-1] with temp.

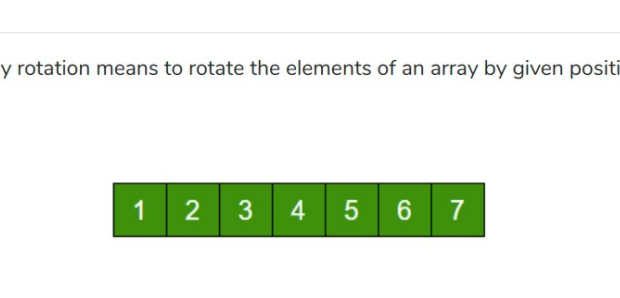
  To rotate an array by K position to the left, repeat the above process K times.
  Take the same example,
  ```
  arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2

  Rotate arr[] one by one 2 times.

  After 1st rotation: [2, 3, 4, 5, 6, 7, 1]
  After 2nd rotation: [ 3, 4, 5, 6, 7, 1, 2]
  ```
  Time Complexity: O(N*K), where N is the number of elements in the array and K is the number of places by which elements will be rotated.
  Auxiliary Space: O(1).
- **Juggling Algorithm**: This is an extension of the above method. Instead of moving one by one, divide the array in different sets, where number of sets is equal to GCD of N and K and move the elements within sets.

  If GCD is 1 as is for the above example array (N = 7 and K = 2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[l+d] to arr[l] and finally store temp at the right place.

  Here is an example for N = 12 and K = 3. GCD of N and K is 3:
  ```
  Let arr[] be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

  a) Elements are first moved in first set – (See below
     diagram for this movement)
  ```
  

  ```
       arr[] after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}

  b) Then in second set.
       arr[] after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}

  c) Finally in third set.
       arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}
  ```
  Time Complexity: O(N), where N is the number of elements in the array.
  Auxiliary Space: O(1).

− Reversing an Array

Reversing an array means reversing the order of elements in the given array.

Problem: Given an array of N elements. The task is to reverse the order of elements in the given array.

For Example:

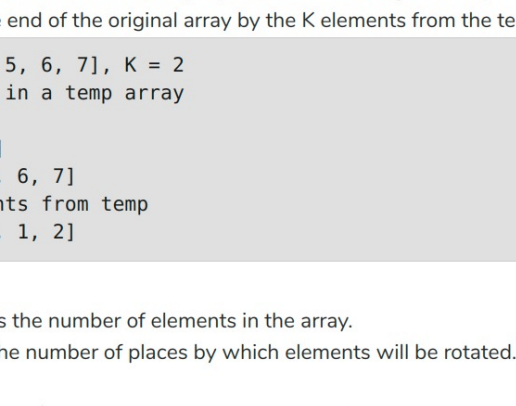
```
Input  : arr[] = {1, 2, 3}
Output : arr[] = {3, 2, 1}

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

Iterative Solution

- **Method 1 (Using Temporary Array)**: The idea is to first copy all of the elements of the given array in a temporary array. Then traverse the temporary array from end and replace elements in original array by elements of temp array.

  This method will take extra space in order of O(N).
- **Method 2 (Efficient)**: This method is efficient then the above method and avoids using extra spaces. The idea is to traverse the array from both ends and keep swapping elements from both ends until middle of the array is reached.
  ```
  1) Initialize start and end indexes as
     start = 0, end = N-1
  2) In a loop, swap arr[start] with arr[end]
     and change start and end as follows :
        start = start + 1,
        end = end – 1
  ```
  For Example:

  Time Complexity: O(N), where N is the number of elements in the array.

Recursive Solution

The recursive approach is almost similar to that of the method 2 of the iterative solution. Below is the recursive algorithm to reverse an array:

```
1) Initialize start and end indexes as
   start = 0, end = n-1
2) Swap arr[start] with arr[end]
3) Recursively call reverse for rest of the array.
```

Below is the recursive function to reverse an array:

```

void rvereseArray(arr[], start, end)
{
    if (start >= end)
        return;

    // Swap elements at start and end
    temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;

    // Recursive Function calling
    rvereseArray(arr, start + 1, end - 1);
}

```

Run

Time Complexity: O(N), where N is the number of elements in the array.

− Sliding Window Technique

This technique shows how a nested for loop in few problems can be converted to single for loop and hence reducing the time complexity.

Let's start with a problem for illustration where we can apply this technique:

```
Given an array of integers of size 'n'.
Our aim is to calculate the maximum sum of 'k'
consecutive elements in the array.

Input  : arr[] = {100, 200, 300, 400}
         k = 2
Output : 700

Input  : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}
         k = 4
Output : 39
We get maximum sum by adding subarray {4, 2, 10, 23}
of size 4.

Input  : arr[] = {2, 3}
         k = 3
Output : Invalid
There is no subarray of size 3 as size of whole
array is 2.
```

The **Naive Approach** to solve this problem is to calculate sum for each of the blocks of K consecutive elements and compare which block has the maximum sum possible. The time complexity of this approach will be O(n * k).

Window Sliding Technique

The above problem can be solved in Linear Time Complexity by using Window Sliding Technique by avoiding the overhead of calculating sum repeatedly for each block of k elements.

The technique can be best understood with the window pane in bus, consider a window of length **n** and the pane which is fixed in it of length **k**. Consider, initially the pane is at extreme left i.e., at 0 units from the left. Now, co-relate the window with array arr[] of size n and plane with current_sum of size k elements. Now, if we apply force on the window such that it moves a unit distance ahead. The pane will cover next **k** consecutive elements.

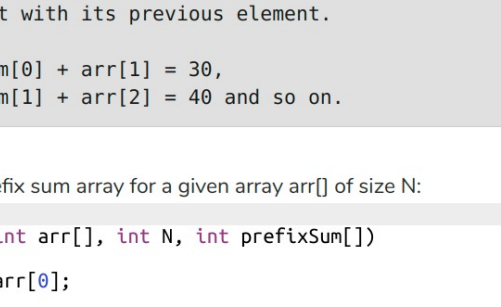
Consider an array **arr[]** = {5 , 2 , -1 , 0 , 3} and value of **k** = 3 and **n** = 5

Applying sliding window technique :

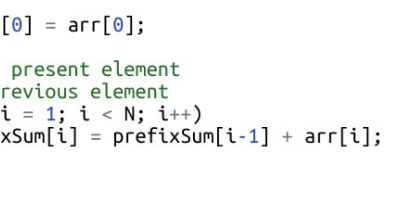
1. We compute the sum of first k elements out of n terms using a linear loop and store the sum in variable window_sum.
2. Then we will graze linearly over the array till it reaches the end and simultaneously keep track of maximum sum.
3. To get the current sum of block of k elements just subtract the first element from the previous block and add the last element of the current block .

The below representation will make it clear how the window slides over the array.

This is the initial phase where we have calculated the initial window sum starting from index 0 . At this stage the window sum is 6. Now, we set the maximum_sum as current_window i.e 6.

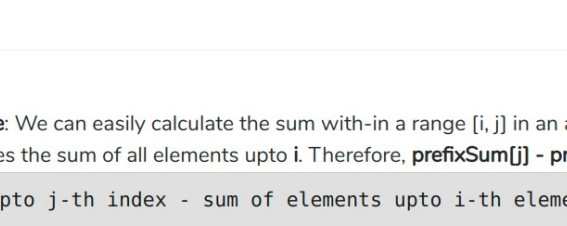


Now, we slide our window by a unit index. Therefore, now it discards 5 from the window and adds 0 to the window. Hence, we will get our new window sum by subtracting 5 and then adding 0 to it. So, our window sum now becomes 1. Now, we will compare this window sum with the maximum_sum. As it is smaller we wont the change the maximum_sum.



Similarly, now once again we slide our window by a unit index and obtain the new window sum to be 2. Again we check if this current window sum is greater than the maximum_sum till now. Once, again it is smaller so we don't change the maximum_sum.

Therefore, for the above array our maximum_sum is 6.



− Prefix Sum Array

Given an array arr[] of size N, the task is to generate the *prefix sum array* of the given array.

Prefix Sum Array: The prefix sum array of any array, arr[] is defined as an array of same size say, prefixSum[] such that the value at any index **i** in prefixSum[] is sum of all elements from indexes **0 to i** in arr[].
That is,

```
prefixSum[i] = arr[0] + arr[1] + arr[2] + . . . . + arr[i]

for all 0 <= i <= N.
```

Examples:

```
Input  : arr[] = {10, 20, 10, 5, 15}
Output : prefixSum[] = {10, 30, 40, 45, 60}

Explanation : While traversing the array, update
the element by adding it with its previous element.
prefixSum[0] = 10,
prefixSum[1] = prefixSum[0] + arr[1] = 30,
prefixSum[2] = prefixSum[1] + arr[2] = 40 and so on.
```

Below function generates a prefix sum array for a given array arr[] of size N:

```

void fillPrefixSum(int arr[], int N, int prefixSum[])
{
    prefixSum[0] = arr[0];

    // Adding present element
    // with previous element
    for (int i = 1; i < N; i++)
        prefixSum[i] = prefixSum[i-1] + arr[i];
}

```

Run

Finding sum in a Range: We can easily calculate the sum with-in a range [i, j] in an array using the prefix sum array. Since the array prefixSum[i] stores the sum of all elements upto **i**. Therefore, **prefixSum[j] - prefixSum[i]** will give:

```
sum of elements upto j-th index - sum of elements upto i-th element
```

The above total sum will exclude the i-th element.

Therefore, we can get the sum of elements in range [i,j] by:

```
prefixSum[j] - prefixSum[i-1]
```

The above formula will not work in the case when **i = 0**.

Therefore,

```
sumInRange = prefixSum[j]   , if i = 0

otherwise,

sumInRange = prefixSum[j] - prefixSum[i-1]   ,   if (i != 0).
```

Sample Problem: Consider an array of size N with all initial values as 0. Perform given 'm' add operations from index 'a' to 'b' and evaluate highest element in array. An add operation adds 100 to all elements from index a to b (both inclusive).

Example:

```
Input : n = 5 // We consider array {0, 0, 0, 0, 0}
        m = 3.
        a = 2, b = 4.
        a = 1, b = 3.
        a = 1, b = 2.
Output : 300

Explanation :
After I operation -
A : 0 100 100 100 0

After II operation -
A : 100 200 200 100 0

After III operation -
A : 200 300 200 100 0

Highest element : 300
```

Solution using Prefix Sum:

```
1 : Run a loop for 'm' times, inputting 'a' and 'b'.
2 : Add 100 at index 'a' and subtract 100 from index 'b+1'.
3 : After completion of 'm' operations, compute the prefix sum array.
4 : Scan the largest element and we're done.
```

What we did was adding 100 at 'a' because this will add 100 to all elements while taking prefix sum array. Subtracting 100 from 'b+1' will reverse the changes made by adding 100 to elements from 'b' onward.

For better understanding :

```
After I operation -
A : 0 100 0 0 -100
Prefix Sum Array : 0 100 100 100 0

After II operation -
A : 100 100 0 -100 -100
Prefix Sum Array : 100 200 200 100 0

After III operation -
A : 200 100 -100 -100 -100
Prefix Sum Array : 200 300 200 100 0

Final Prefix Sum Array : 200 300 200 100 0

The required highest element : 300
```

− Implementing Arrays in C++ using STL

We already have discussed the basic declaration of arrays. Arrays can also be implemented using some built-in classes available in the C++ Standard Template Library.

Some of the most commonly used classes for implementing sequential lists or arrays are:

- Vector
- List

Let's look at each of these classes in details.

Vector

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.

To use the Vector class, include the below header file in your program:

```
#include< vector >
```

Declaring Vector:

```
vector< Type_of_element > vector_name;

Here, Type_of_element can be any valid C++ data type,
or can be any other container also like Pair, List etc.
```

Some important and commonly used functions of Vector class are:

- **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.

Below program illustrate the above methods:

```

// 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);
```

Run

Output:

```
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
```

List

Lists are sequence containers that allow non-contiguous memory allocation. List in C++ STL implements a doubly linked list and not arrays. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked lists. For implementing a singly linked list, we can use **forward_list** class in C++ STL.

To use the List class, include the below header file in your program:

```
#include< list >
```

Declaring List:

```
list< Type_of_element > list_name;

Here, Type_of_element can be any valid C++ data type,
or can be any other container also like Pair, List etc.
```

Some important and commonly used functions of List are:

- **front()** – Returns the value of the first element in the list.
- **back()** – Returns the value of the last element in the list.
- **push_front(g)** – Adds a new element 'g' at the beginning of the list.
- **push_back(g)** – Adds a new element 'g' at the end of the list.
- **pop_front()** – Removes the first element of the list, and reduces the size of the list by 1.
- **pop_back()** – Removes the last element of the list, and reduces the size of the list by 1.
- **begin()** and **end()** – begin() function returns an iterator pointing to the first element of the list.
- **empty()** – Returns whether the list is empty(1) or not(0).
- **insert()** – Inserts new elements in the list before the element at a specified position.
- **reverse()** – Reverses the list.
- **size()** – Returns the number of elements in the list.
- **sort()** – Sorts the list in increasing order.

Below program illustrate the above functions:

```

#include <iostream>
#include <list>
#include <iterator>
using namespace std;

//function for printing the elements in a list
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main()
{

    list <int> gqlist1, gqlist2;


    for (int i = 0; i < 10; ++i)
    {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
```

Run

Output:

```
List 1 (gqlist1) is :     0    2    4    6
8    10    12    14    16    18

List 2 (gqlist2) is :     27    24    21    18
15    12    9    6    3    0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :     2    4    6    8
10    12    14    16    18

gqlist2.pop_back() :     27    24    21    18
15    12    9    6    3

gqlist1.reverse() :     18    16    14    12
10    8    6    4    2

gqlist2.sort():     3    6    9    12
15    18    21    24    27
```

+ Iterators in C++ STL

− Implementing Arrays in Java

Arrays in Java are used to store a group of elements of same data type at contiguous memory locations.

The general form of **Array declaration** in Java is:

```
type array-name[];

OR

type[] array-name;
```

An array declaration has two components: *the type* and *the name*. Type declares the element type of the array. The element type determines the data type of each element that comprises the array. Like array of int type, we can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class). Thus, the element type for the array determines what type of data the array will hold.

For Example:

```
// both are valid declarations
int intArray[];
or int[] intArray;

byte byteArray[];
short shortsArray[];
boolean booleanArray[];
long longArray[];
float floatArray[];
double doubleArray[];
char charArray[];
```

Instantiating an Array: When an array is declared, only a reference of array is created. To actually create or give memory to array, you create an array like this:

```
array-name = new type [size];
```

Here,

- **type** specifies the type of data being allocated.
- **size** specifies the number of elements in the array.
- **array-name** is the name of array variable that is linked to the array.

Example:

```
int intArray[];    //declaring array
intArray = new int[20];  // allocating memory to array
```

OR

```
int[] intArray = new int[20]; // combining both statements in one
```

Accessing Java Array Elements using for Loop: Each element in the array is accessed by its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop as shown below.

```


// Accessing the elements of the specified array

for (int i = 0; i < arr.length; i++)

  System.out.println("Element at index " + i +

                                " : "+ arr[i]);



Here, arr.length gives the number of elements

present in the array named arr.
```

Implementation:

```

// Java program to illustrate creating an array
// of integers,  puts some values in the array,
// and prints each value to standard output.

class GFG
{
    public static void main (String[] args)
    {
      // declares an Array of integers.
      int[] arr;

      // allocating memory for 5 integers.
      arr = new int[5];

      // initialize the first elements of the array
      arr[0] = 10;

      // initialize the second elements of the array
      arr[1] = 20;

      // so on...
      arr[2] = 30;
      arr[3] = 40;
      arr[4] = 50;

      // accessing the elements of the specified array
      for (int i = 0; i < arr.length; i++)
         System.out.println("Element at index " + i +
                                      " : "+ arr[i]);
```

Run

Output:

```
Element at index 0 : 10
Element at index 1 : 20
Element at index 2 : 30
Element at index 3 : 40
Element at index 4 : 50
```

Java also provides some inbuilt classes which can be used for implementing arrays or sequential lists. Let's look at some of these in details.

ArrayList in Java

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of array manipulation is needed.

Constructors in Java ArrayList:

- **ArrayList()**: This constructor is used to build an empty array list.
- **ArrayList(Collection c)**: This constructor is used to build an array list initialized with the elements from collection c.
- **ArrayList(int capacity)**: This constructor is used to build an array list with initial capacity being specified.

Creating a generic integer ArrayList:

```
ArrayList arrli = new ArrayList();
```

Implementation:

```

// Java program to demonstrate working of
// ArrayList in Java

import java.io.*;
import java.util.*;

class arrayli
{
    public static void main(String[] args)
                       throws IOException
    {
        // size of ArrayList
        int n = 5;

        // declaring ArrayList with initial size n
        ArrayList<Integer> arrli = new ArrayList<Integer>(n);

        // Appending the new element at the end of the list
        for (int i=1; i<=n; i++)
            arrli.add(i);

        // Printing elements
        System.out.println(arrli);

        // Remove element at index 3
        arrli.remove(3);

        // Displaying ArrayList after deletion
        System.out.println(arrli);
```

Run

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Click Here to learn ArrayList in Java in details!

Vector Class in Java

The Vector class implements a growable array of objects. Vectors basically falls in legacy classes but now it is fully compatible with collections.

- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.
- They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
- It extends AbstractList and implements List interfaces.

Constructor:

- **Vector()**: Creates a default vector of initial capacity is 10.
- **Vector(int size)**: Creates a vector whose initial capacity is specified by size.
- **Vector(int size, int incr)**: Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.
- **Vector(Collection c)**: Creates a vector that contains the elements of collection c.

Implementation:

```

// Java code to illustrate Vector

import java.util.*;
class Vector_demo {
    public static void main(String[] arg)
    {
        // Create a vector
        Vector<Integer> v = new Vector<Integer>();

        // Insert elements in the Vector
        v.add(1);
        v.add(2);
        v.add(3);
        v.add(4);
        v.add(3);

        // Print the Vector
        System.out.println("Vector is " + v);
    }
}

```

Run

Output:

```
Vector is [1, 2, 3, 4, 3]
```

Click Here to learn Vector in Java in details!

+ Sample Problems on Array

Report An Issue

If you are facing any issue on this page. Please let us know.

GeeksforGeeks
A computer science portal for geeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org

COMPANY
About Us
Careers
Privacy Policy
Contact Us

LEARN
Algorithms
Data Structures
Languages
CS Subjects
Video Tutorials

PRACTICE
Courses
Company-wise
Topic-wise
How to begin?

CONTRIBUTE
Write an Article
Interview Experience
Internships
Videos

@geeksforgeeks, Some rights reserved