Contents

Contents 2

**1.** **The C++ Standard Template Library (STL)** 3

**1.1.** **Algorithms** : 5

**1.1.1.** **Sort in C++ Standard Template Library (STL)** 5

**1.1.2.** **Binary Search in C++ Standard Template Library (STL)** 7

**1.1.3.** **Algorithm Library | C++ Magicians STL Algorithm** 10

**1.1.4.** **Array algorithms in C++ STL** 13

**1.1.5.** **std::partition in C++ STL** 17

**1.2.** **Containers** 21

**1.2.1.** **Sequence Containers:** 21

**1.2.3.1.** **Vector in C++ STL** 21

**1.2.3.2.** **List in C++ Standard Template Library (STL)** 32

**1.2.3.3.** **Deque in C++ Standard Template Library (STL)** 36

**1.2.3.4.** **Array class in C++** 38

**1.2.3.5.** **Forward List in C++ | Set 1 (Introduction and Important Functions)** 44

1.2.2. **Container** **Adaptors**: 51

**1.2.3.1.** **Queue in Standard Template Library (STL)** 51

**1.2.3.2.** **Priority Queue in C++ Standard Template Library (STL)** 52

**1.2.3.3.** **Stack in C++ Standard Template Library (STL)** 54

**1.2.3.** **Associative Containers :** 56

**1.2.3.1.** **Set in C++ Standard Template Library (STL)** 56

**1.2.3.2.** **Multiset in C++ Standard Template Library (STL)** 59

**1.2.3.3.** **Map in C++ Standard Template Library (STL)** 62

**1.2.3.4.** **Multimap in C++ Standard Template Library (STL)** 66

**1.3.** **Functors in C++** 70

**1.4.** **Iterators in C++ STL** 72

**1.5.** **Pair in C++ Standard Template Library (STL)** 75

1.5.1. **Initializing a pair** 75

**1.5.2.** **Member** **Functions** 77

**1.5.3.1.** **make\_pair()** : 77

**1.5.3.2.** **operators(=, ==, !=, >=, <=) :** 77

**1.5.3.3.** **swap** 79

# **The C++ Standard Template Library (STL)**

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of [template classes](https://www.geeksforgeeks.org/template-specialization-c/) is a prerequisite for working with STL.

**STL has four components**

* Algorithms
* Containers
* Functions
* Iterators

**Algorithms :**The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations  for the contents of the containers.

* Algorithm
  + [Sorting](http://quiz.geeksforgeeks.org/sort-algorithms-the-c-standard-template-library-stl/)
  + [Searching](http://quiz.geeksforgeeks.org/binary-search-algorithms-the-c-standard-template-library-stl/)
  + [Important STL Algorithms](https://www.geeksforgeeks.org/c-magicians-stl-algorithms/)
  + [Useful Array algorithms](https://www.geeksforgeeks.org/useful-array-algorithms-in-c-stl/)
  + [Partition Operations](https://www.geeksforgeeks.org/stdpartition-in-c-stl/)
* Numeric
  + valarray class

**Containers :** Containers or container classes store objects and data. There are in total seven standard “first-class” container classes  and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

* Sequence Containers:  implement data structures which can be accessed in a sequential manner.
  + [vector](http://quiz.geeksforgeeks.org/vector-sequence-containers-the-c-standard-template-library-stl-set-1/)
  + [list](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/)
  + [deque](http://quiz.geeksforgeeks.org/deque-sequence-containers-the-c-standard-template-library-stl/)
  + [arrays](https://www.geeksforgeeks.org/array-class-c/)
  + [forward\_list](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)( Introduced in C++11)
* Container Adaptors:  provide a different interface for sequential containers.
  + [queue](http://quiz.geeksforgeeks.org/queue-container-adaptors-the-c-standard-template-library-stl/)
  + [priority\_queue](http://quiz.geeksforgeeks.org/priority-queue-container-adaptors-the-c-standard-template-library-stl/)
  + [stack](http://quiz.geeksforgeeks.org/stack-container-adaptors-the-c-standard-template-library-stl/)
* Associative Containers:  implement sorted data structures that can be quickly searched (O(log n)complexity).
  + [set](http://quiz.geeksforgeeks.org/set-associative-containers-the-c-standard-template-library-stl/)
  + [multiset](http://quiz.geeksforgeeks.org/multiset-associative-containers-the-c-standard-template-library-stl/)
  + [map](http://quiz.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)
  + [multimap](http://quiz.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)

**Functions :** The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

* [Functors](https://www.geeksforgeeks.org/functors-in-cpp/)

**Iterators :** As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

* [Iterators](https://www.geeksforgeeks.org/iterators-c-stl/)

**Utility Library :** Defined under <utility header>

* [pair](http://quiz.geeksforgeeks.org/pair-simple-containers-the-c-standard-template-library-stl/)

## **Algorithms** :

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations  for the contents of the containers.

### **Sort in C++ Standard Template Library (STL)**

Sorting is one of the most basic functions applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing. There is a builtin function in C++ STL by the name of sort().

Internally this function is implemented as Quick-sort. The complexity of it is O(N\*log(N)).

The prototype for sort is :

sort(startaddress, endaddress)

startaddress: the address of the first element of the array

endaddress: the address of the last element of the array

|  |
| --- |
| #include <iostream>  #include <algorithm>  using namespace std;    void show(int a[])  {      for(int i = 0; i < 10; ++i)          cout << a[i] << " ";  }  int main()  {      int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};      cout << "\n The array before sorting is : ";      show(a);        sort(a, a+10);        cout << "\n\n The array after sorting is : ";      show(a);        return 0;  } |

The outut of the above program is :

The array before sorting is : 1 5 8 9 6 7 3 4 2 0

The array after sorting is : 0 1 2 3 4 5 6 7 8 9

### **Binary Search in C++ Standard Template Library (STL)**

[Binary search](https://www.geeksforgeeks.org/binary-search/) is a widely used searching algorithm that requires the array to be sorted before search is applied. The main idea behind this algorithm is to keep dividing the array in half (divide and conquer) until the element is found, or all the elements are exhausted.

It works by comparing the middle item of the array with our target, if it matches, it returns true otherwise if the middle term is greater than the target, the search is performed in the left sub-array.

If the middle term is less than target, the search is performed in the right sub-array.

The prototype for binary search is :

binary\_search(startaddress, endaddress, valuetofind)

startaddress: the address of the first element of the array.

endaddress: the address of the last element of the array.

valuetofind: the target value which we have to search for.

#include <iostream>

#include <algorithm>

using namespace std;

void show(int a[], int arraysize)

{

for(int i = 0; i < arraysize; ++i)

cout << a[i] <<" ";

}

int main()

{

int a[]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};

int asize = sizeof(a) / sizeof(a[0]);

cout << "\n The array is : ";

show(a, asize);

cout << "\n\nLet's say we want to search for 2 in the array";

cout << "\n So, we first sort the array";

sort(a, a + 10);

cout << "\n\n The array after sorting is : ";

show(a, asize);

cout << "\n\nNow, we do the binary search";

if (binary\_search(a, a + 10, 2))

cout << "\nElement found in the array";

else

cout << "\nElement not found in the array";

cout << "\n\nNow, say we want to search for 10";

if (binary\_search(a, a + 10, 10))

cout << "\nElement found in the array";

else

cout << "\nElement not found in the array";

return 0;

}

The output of the above program is :

The array is : 1 5 8 9 0 6 7 3 4 2 0

Let's say we want to search for 2 in the array

So, we first sort the array

The array after sorting is : 0 1 2 3 4 5 6 7 8 9

Now, we do the binary search

Element found in the array

Now, say we want to search for 10

Element not found in the array

### **Algorithm Library | C++ Magicians STL Algorithm**

For all those who aspire to excel in competitive programming, only having a knowledge about containers of STL is of less use till one is not aware what all STL has to offer.

STL has an ocean of algorithms, for all < algorithm > library functions : Refer here.

Some of the most used algorithms on vectors and most useful one’s in Competitive Programming are mentioned as follows :

**Non-Manipulating Algorithms**

1. [**sort**](https://www.geeksforgeeks.org/sort-c-stl/)**(first\_iterator, last\_iterator)** – To sort the given vector.
2. **reverse(first\_iterator, last\_iterator)** – To reverse a vector.
3. **\*max\_element (first\_iterator, last\_iterator)** – To find the maximum element of a vector.
4. **\*min\_element (first\_iterator, last\_iterator)**– To find the minimum element of a vector.
5. **accumulate(first\_iterator, last\_iterator, initial value of sum)** – Does the summation of vector elements

|  |
| --- |
| // A C++ program to demonstrate working of sort() and reverse()  #include <algorithm>  #include <iostream>  #include <vector>  #include <numeric> //For accumulate operation  using namespace std;    int main()  {      // Initializing vector with array values      int arr[] = {10, 20, 5, 23 ,42 , 15};      int n = sizeof(arr)/sizeof(arr[0]);      vector<int> vect(arr, arr+n);        cout << "Vector is: ";      for (int i=0; i<n; i++)          cout << vect[i] << " ";      sort(vect.begin(), vect.end()); // Sorting in Ascending order        cout << "\nVector after sorting is: ";      for (int i=0; i<n; i++)         cout << vect[i] << " ";        reverse(vect.begin(), vect.end()); // Reversing        cout << "\nVector after reversing is: ";      for (int i=0; i<6; i++)          cout << vect[i] << " ";        cout << "\nMaximum element of vector is: ";      cout << \*max\_element(vect.begin(), vect.end());        cout << "\nMinimum element of vector is: ";      cout << \*min\_element(vect.begin(), vect.end());        // Starting the summation from 0      cout << "\nThe summation of vector elements is: ";      cout << accumulate(vect.begin(), vect.end(), 0);        return 0;  } |

Output:

Vector before sorting is: 10 20 5 23 42 15

Vector after sorting is: 5 10 15 20 23 42

Vector before reversing is: 5 10 15 20 23 42

Vector after reversing is: 42 23 20 15 10 5

Maximum element of vector is: 42

Minimum element of vector is: 5

The summation of vector elements is: 115

1. **count(first\_iterator, last\_iterator,x)** – To count the occurrences of x in vector.
2. **find(first\_iterator, last\_iterator, x)** – Points to last address of vector ((name\_of\_vector).end()) if element is not present in vector.

|  |
| --- |
| // C++ program to demonstrate working of count()and find()  #include <algorithm>  #include <iostream>  #include <vector>  using namespace std;    int main()  {      int arr[] = {10, 20, 5, 23 ,42, 20, 15};      int n = sizeof(arr)/sizeof(arr[0]);      vector<int> vect(arr, arr+n);        cout << "Occurrences of 20 in vector : ";        // Counts the occurrences of 20 from 1st to      // last element      cout << count(vect.begin(), vect.end(), 20);        // find() returns iterator to last address if      // element not present      find(vect.begin(), vect.end(),5) != vect.end()?                           cout << "\nElement found":                       cout << "\nElement not found";        return 0;  } |

Output:

Occurrences of 20 in vector: 2

Element found

### **Array algorithms in C++ STL**

From C++11 onwards, some new and interesting algorithms are added in STL of C++. These algorithms operate on an array and are useful in saving time during coding and hence useful in competitive programming as well.

**all\_of() :** This function operates on whole range of array elements and can save time to run a loop to check each elements one by one. It checks for a given property on every element and returns true when each element in range satisfies specified property, else returns false.

|  |
| --- |
| // C++ code to demonstrate working of all\_of()  #include<iostream>  #include<algorithm> // for all\_of()  using namespace std;  int main()  {      int ar[6] =  {1, 2, 3, 4, 5, -6};        // Checking if all elements are positive      all\_of(ar, ar+6, [](int x) { return x>0; })?            cout << "All are positive elements" :            cout << "All are not positive elements";        return 0;  } |

Output:

All are not positive elements

In the above code, -6 being a negative element negates the condition and returns false.

**any\_of() :** This function checks for a given range if there’s even one element satisfying a given property mentioned in function. Returns true if at least one element satisfies the property else returns false.

|  |
| --- |
| // C++ code to demonstrate working of any\_of()  #include<iostream>  #include<algorithm> // for any\_of()  using namespace std;  int main()  {       int ar[6] =  {1, 2, 3, 4, 5, -6};        // Checking if any element is negative      any\_of(ar, ar+6, [](int x){ return x<0; })?            cout << "There exists a negative element" :            cout << "All are positive elements";        return 0;  } |

Output:

There exists a negative element

In above code, -6 makes the condition positive.

**none\_of() :** This function returns true if none of elements satisfies the given condition else returns false.

|  |
| --- |
| // C++ code to demonstrate working of none\_of()  #include<iostream>  #include<algorithm> // for none\_of()  using namespace std;  int main()  {      // Initializing array      int ar[6] =  {1, 2, 3, 4, 5, 6};        // Checking if no element is negative      none\_of(ar, ar+6, [](int x){ return x<0; })?            cout << "No negative elements" :            cout << "There are negative elements";        return 0;  } |

Output:

No negative elements

Since all elements are positive, the function returns true.

**copy\_n()**

copy\_n() copies one array elements to new array. This type of copy creates a deep copy of array. This function takes 3 arguments, source array name, size of array and the target array name.

|  |
| --- |
| // C++ code to demonstrate working of copy\_n()  #include<iostream>  #include<algorithm> // for copy\_n()  using namespace std;  int main()  {      int ar[6] =  {1, 2, 3, 4, 5, 6}; // Initializing array        int ar1[6];  // Declaring second array      copy\_n(ar, 6, ar1); // Using copy\_n() to copy contents        cout << "The new array after copying is : ";     // Displaying the copied array      for (int i=0; i<6 ; i++)         cout << ar1[i] << " ";        return 0;   } |
| Output:  The new array after copying is : 1 2 3 4 5 6 |
|  |

In the above code, the elements of ar are copied in ar1 using copy\_n()

**iota()**

This function is used to assign continuous values to array. This function accepts 3 arguments, the array name, size, and the starting number.

|  |
| --- |
| // C++ code to demonstrate working of iota()  #include<iostream>  #include<numeric> // for iota()  using namespace std;  int main()  {      // Initializing array with 0 values      int ar[6] =  {0};        // Using iota() to assign values      iota(ar, ar+6, 20);        // Displaying the new array      cout << "The new array after assigning values is : ";      for (int i=0; i<6 ; i++)         cout << ar[i] << " ";        return 0;   } |

Output:

The new array after assigning values is : 20 21 22 23 24 25

In the above code, continuous values are assigned to array using iota().

### **std::partition in C++ STL**

C++ has a class in its STL algorithms library which allows us easy partition algorithms using certain inbuilt functions. Partition refers to act of dividing elements of containers depending upon a given condition.  
 **Partition operations** **:**

**1. partition(beg, end, condition)** :- This function is used to **partition the elements** on **basis of condition**mentioned in its arguments.

**2. is\_partitioned(beg, end, condition)** :- This function returns boolean **true if container is partitioned** else returns false.

|  |
| --- |
| // C++ code to demonstrate the working of partition() and is\_partitioned()  #include<iostream>  #include<algorithm> // for partition algorithm  #include<vector> // for vector  using namespace std;  int main()  {      // Initializing vector      vector<int> vect = { 2, 1, 5, 6, 8, 7 };        // Checking if vector is partitioned      // using is\_partitioned()      is\_partitioned(vect.begin(), vect.end(), [](int x)      {          return x%2==0;        })?        cout << "Vector is partitioned":      cout << "Vector is not partitioned";      cout << endl;        // partitioning vector using partition()      partition(vect.begin(), vect.end(), [](int x)      {          return x%2==0;        });        // Checking if vector is partitioned      // using is\_partitioned()      is\_partitioned(vect.begin(), vect.end(), [](int x)      {          return x%2==0;        })?        cout << "Now, vector is partitioned after partition operation":      cout << "Vector is still not partitioned after partition operation";      cout << endl;        // Displaying partitioned Vector      cout << "The partitioned vector is : ";      for (int &x : vect) cout << x << " ";        return 0;    } |

Vector is not partitioned

Now, vector is partitioned after partition operation

The partitioned vector is : 2 8 6 5 1 7

In the above code, partition function partitions the vector depending on whether an element is even or odd, even elements are partitioned from odd elements in no particular order.

**3. stable\_partition(beg, end, condition)**:- This function is used to **partition the elements** on **basis of condition**mentiond in its arguments in **sorted order**.

**4. partition\_point(beg, end, condition)**:- This function**returns an iterator pointing to the partition point** of container i.e. the first element in the partitioned range [beg,end) for which

condition is not true. The container should already be partitioned for this function to work.

|  |
| --- |
| // C++ code to demonstrate the working of  // stable\_partition() and partition\_point()  #include<iostream>  #include<algorithm> // for partition algorithm  #include<vector> // for vector  using namespace std;  int main()  {      // Initializing vector      vector<int> vect = { 2, 1, 5, 6, 8, 7 };        // partitioning vector using stable\_partition()      // in sorted order      stable\_partition(vect.begin(), vect.end(), [](int x)      {          return x%2 == 0;      });        // Displaying partitioned Vector      cout << "The partitioned vector is : ";      for (int &x : vect) cout << x << " ";      cout << endl;        // Declaring iterator      vector<int>::iterator it1;        // using partition\_point() to get ending position of partition      auto it = partition\_point(vect.begin(), vect.end(), [](int x)      {          return x%2==0;      });        // Displaying partitioned Vector      cout << "The vector elements returning true for condition are : ";      for ( it1= vect.begin(); it1!=it; it1++)      cout << \*it1 << " ";      cout << endl;        return 0;    } |

The partitioned vector is : 2 6 8 1 5 7

The vector elements returning true for condition are : 2 6 8

In the above code, even and odd elements are partitioned and in the increasing order (sorted)

**5. partition\_copy(beg, end, beg1, beg2, condition)** :- This function **copies the partitioned elements** in the differenet containers mentioned in its arguments. It takes 5 arguments. **Beginning and ending position of container, beginning position of new container where elements have to be copied (elements returning true for condition), beginning position of new container where other elements have to be copied (elements returning false for condition) and the condition**.**Resizing** new containers **is necessary** for this function.

|  |
| --- |
| // C++ code to demonstrate the working of partition\_copy()  #include<iostream>  #include<algorithm> // for partition algorithm  #include<vector> // for vector  using namespace std;  int main()  {      vector<int> vect = { 2, 1, 5, 6, 8, 7 };        vector<int> vect1;     // Declaring vector1        vector<int> vect2;  // Declaring vector1        // Resizing vectors to suitable size using count\_if() and resize()      int n = count\_if (vect.begin(), vect.end(), [](int x)      {          return x%2==0;        } );      vect1.resize(n);      vect2.resize(vect.size()-n);        // Using partition\_copy() to copy partitions      partition\_copy(vect.begin(), vect.end(), vect1.begin(),                                       vect2.begin(), [](int x)      {          return x%2==0;      });          // Displaying partitioned Vector      cout << "The elements that return true for condition are : ";      for (int &x : vect1)              cout << x << " ";      cout << endl;        // Displaying partitioned Vector      cout << "The elements that return false for condition are : ";      for (int &x : vect2)              cout << x << " ";      cout << endl;        return 0;  } |

The elements that return true for condition are : 2 6 8

The elements that return false for condition are : 1 5 7

## **Containers**

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes  and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

### **Sequence Containers:**

#### **Vector in C++ STL**

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.

Certain functions associated with the vector are:

**Iterators**

1. [begin()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – Returns an iterator pointing to the first element in the vector
2. [end()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/) – Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. [rbegin()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. [rend()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/) – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. [cbegin()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/) – Returns a constant iterator pointing to the first element in the vector.
6. [cend()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/) – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. [crbegin()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/) – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
8. [crend()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/) – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

// C++ progra to illustrate the iterators in vector

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> g1;

for (int i = 1; i <= 5; i++)

g1.push\_back(i);

cout << "Output of begin and end: ";

for (auto i = g1.begin(); i != g1.end(); ++i)

cout << \*i << " ";

cout << "\nOutput of cbegin and cend: ";

for (auto i = g1.cbegin(); i != g1.cend(); ++i)

cout << \*i << " ";

cout << "\nOutput of rbegin and rend: ";

for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)

cout << \*ir << " ";

cout << "\nOutput of crbegin and crend : ";

for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)

cout << \*ir << " ";

return 0;

}

**Output:**

Output of begin and end: 1 2 3 4 5

Output of cbegin and cend: 1 2 3 4 5

Output of rbegin and rend: 5 4 3 2 1

Output of crbegin and crend : 5 4 3 2 1

**Capacity**

1. [size()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/) – Returns the number of elements in the vector.
2. [max\_size()](https://www.geeksforgeeks.org/vector-max_size-function-in-c-stl/) – Returns the maximum number of elements that the vector can hold.
3. [capacity()](https://www.geeksforgeeks.org/vector-capacity-function-in-c-stl/) – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. [resize()](https://www.geeksforgeeks.org/vector-resize-c-stl/) – Resizes the container so that it contains ‘g’ elements.
5. [empty()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/) – Returns whether the container is empty.
6. [shrink\_to\_fit()](https://www.geeksforgeeks.org/vector-shrink_to_fit-function-in-c-stl/) – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. [reserve()](https://www.geeksforgeeks.org/using-stdvectorreserve-whenever-possible/)– Requests that the vector capacity be at least enough to contain n elements.

// C++ program to illustrate the

// capacity function in vector

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> g1;

for (int i = 1; i <= 5; i++)

g1.push\_back(i);

cout << "Size : " << g1.size();

cout << "\nCapacity : " << g1.capacity();

cout << "\nMax\_Size : " << g1.max\_size();

g1.resize(4); // resizes the vector size to 4

cout << "\nSize : " << g1.size(); // prints the vector size after resize()

if (g1.empty() == false) // checks if the vector is empty or not

cout << "\nVector is not empty";

else

cout << "\nVector is empty";

g1.shrink\_to\_fit(); // Shrinks the vector

cout << "\nVector elements are: ";

for (auto it = g1.begin(); it != g1.end(); it++)

cout << \*it << " ";

return 0;

}

**Output:**

Size : 5

Capacity : 8

Max\_Size : 4611686018427387903

Size : 4

Vector is not empty

Vector elements are: 1 2 3 4

**Element access:**

1. [reference operator [g]](https://www.geeksforgeeks.org/vectoroperator-vectoroperator-c-stl/) – Returns a reference to the element at position ‘g’ in the vector
2. [at(g)](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/) – Returns a reference to the element at position ‘g’ in the vector
3. [front()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/) – Returns a reference to the first element in the vector
4. [back()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/) – Returns a reference to the last element in the vector
5. [data()](https://www.geeksforgeeks.org/vector-data-function-in-c-stl/) – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

// C++ program to illustrate the

// element accesser in vector

#include <bits/stdc++.h>

using namespace std;

int main()

{

vector<int> g1;

for (int i = 1; i <= 10; i++)

g1.push\_back(i \* 10);

cout << "\nReference operator [g] : g1[2] = " << g1[2];

cout << "\nat : g1.at(4) = " << g1.at(4);

cout << "\nfront() : g1.front() = " << g1.front();

cout << "\nback() : g1.back() = " << g1.back();

// pointer to the first element

int\* pos = g1.data();

cout << "\nThe first element is " << \*pos;

return 0;

}

**Output:**

Reference operator [g] : g1[2] = 30

at : g1.at(4) = 50

front() : g1.front() = 10

back() : g1.back() = 100

The first element is 10

**Modifiers:**

1. [assign()](https://www.geeksforgeeks.org/vector-assign-in-c-stl/)– It assigns new value to the vector elements by replacing old ones
2. [push\_back()](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/) – It push the elements into a vector from the back
3. [pop\_back()](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/) – It is used to pop or remove elements from a vector from the back.
4. insert() – It inserts new elements before the element at the specified position
5. [erase()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/) – It is used to remove elements from a container from the specified position or range.
6. [swap()](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/) – It is used to swap the contents of one vector with another vector of same type and size.
7. [clear()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/) – It is used to remove all the elements of the vector container
8. [emplace()](https://www.geeksforgeeks.org/vector-emplace-function-in-c-stl/) – It extends the container by inserting new element at position
9. [emplace\_back()](https://www.geeksforgeeks.org/vectoremplace_back-c-stl/) – It is used to insert a new element into the vector container, the new element is added to the end of the vector

.

// C++ program to illustrate the

// Modifiers in vector

#include <bits/stdc++.h>

#include <vector>

using namespace std;

int main()

{

vector<int> v; // Assign vector

v.assign(5, 10); // fill the array with 10 five times

cout << "The vector elements are: ";

for (int i = 0; i < v.size(); i++)

cout << v[i] << " ";

v.push\_back(15); // inserts 15 to the last position

int n = v.size();

cout << "\nThe last element is: " << v[n - 1];

v.pop\_back(); // removes last element

cout << "\nThe vector elements are: "; // prints the vector

for (int i = 0; i < v.size(); i++)

cout << v[i] << " ";

// inserts 5 at the beginning

v.insert(v.begin(), 5);

cout << "\nThe first element is: " << v[0];

// removes the first element

v.erase(v.begin());

cout << "\nThe first element is: " << v[0];

// 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);

n = v.size();

cout << "\nThe last element is: " << v[n - 1];

// erases the vector

v.clear();

cout << "\nVector size after erase(): " << v.size();

// two vector to perform swap

vector<int> v1, v2;

v1.push\_back(1);

v1.push\_back(2);

v2.push\_back(3);

v2.push\_back(4);

cout << "\n\nVector 1: ";

for (int i = 0; i < v1.size(); i++)

cout << v1[i] << " ";

cout << "\nVector 2: ";

for (int i = 0; i < v2.size(); i++)

cout << v2[i] << " ";

// Swaps v1 and v2

v1.swap(v2);

cout << "\nAfter Swap \nVector 1: ";

for (int i = 0; i < v1.size(); i++)

cout << v1[i] << " ";

cout << "\nVector 2: ";

for (int i = 0; i < v2.size(); i++)

cout << v2[i] << " ";

}

**Output:**

The vector elements are: 10 10 10 10 10

The last element is: 15

The vector elements are: 10 10 10 10 10

The first element is: 5

The first element is: 10

The first element is: 5

The last element is: 20

Vector size after erase(): 0

Vector 1: 1 2

Vector 2: 3 4

After Swap

Vector 1: 3 4

Vector 2: 1 2

**All Vector Functions :**

* [vector::begin() and vector::end()](https://www.geeksforgeeks.org/vectorbegin-vectorend-c-stl/)
* [vector rbegin() and rend()](https://www.geeksforgeeks.org/vector-rbegin-and-rend-function-in-c-stl/)
* [vector::cbegin() and vector::cend()](https://www.geeksforgeeks.org/vector-cbegin-vector-cend-c-stl/)
* [vector::crend() and vector::crbegin()](https://www.geeksforgeeks.org/vectorcrend-vectorcrbegin-examples/)
* [vector::assign()](https://www.geeksforgeeks.org/vector-assign-in-c-stl/)
* [vector::at()](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/)
* [vector::back()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/)
* [vector::capacity()](https://www.geeksforgeeks.org/vector-capacity-function-in-c-stl/)
* [vector::clear()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/)
* [vector::push\_back()](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/)
* [vector::pop\_back()](https://www.geeksforgeeks.org/vectorpush_back-vectorpop_back-c-stl/)
* [vector::empty()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/)
* [vector::erase()](https://www.geeksforgeeks.org/vectorclear-vectorerase-c-stl/)
* [vector::size()](https://www.geeksforgeeks.org/vectorempty-vectorsize-c-stl/)
* [vector::swap()](https://www.geeksforgeeks.org/vectorat-vectorswap-c-stl/)
* [vector::reserve()](https://www.geeksforgeeks.org/using-stdvectorreserve-whenever-possible/)
* [vector::resize()](https://www.geeksforgeeks.org/vector-resize-c-stl/)
* [vector::shrink\_to\_fit()](https://www.geeksforgeeks.org/vector-shrink_to_fit-function-in-c-stl/)
* [vector::operator=](https://www.geeksforgeeks.org/vectoroperator-vectoroperator-c-stl/)
* [vector::operator[]](https://www.geeksforgeeks.org/vectoroperator-vectoroperator-c-stl/)
* [vector::front()](https://www.geeksforgeeks.org/vectorfront-vectorback-c-stl/)
* [vector::data()](https://www.geeksforgeeks.org/vector-data-function-in-c-stl/)
* [vector::emplace\_back()](https://www.geeksforgeeks.org/vectoremplace_back-c-stl/)
* [vector::emplace()](https://www.geeksforgeeks.org/vector-emplace-function-in-c-stl/)
* [vector::max\_size()](https://www.geeksforgeeks.org/vector-max_size-function-in-c-stl/)
* [vector::insert()](https://www.geeksforgeeks.org/vector-insert-function-in-c-stl/)

#### **List in C++ Standard Template Library (STL)**

Lists are sequence containers that allow non-contiguous memory allocation. 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 list. For implementing a singly linked list, we use forward list.

 Functions used with List :

1. front() – Returns the value of the first element in the list
2. back() – Returns the value of the last element in the list
3. push\_front(g) – Adds a new element ‘g’ at the beginning of the list
4. push\_back(g) – Adds a new element ‘g’ at the end of the list
5. pop\_front() – Removes the first element of the list, and reduces size of the list by 1
6. pop\_back() – Removes the last element of the list, and reduces size of the list by 1
7. begin() – Returns an iterator pointing to the first element of the list
8. end() – Returns an iterator pointing to the theoretical last element which follows the last element
9. empty() – Returns whether the list is empty(1) or not(0)
10. insert() – Inserts new elements in the list before the element at a specified position
11. erase() – Removes a single element or a range of elements from the list
12. assign() – Assigns new elements to list by replacing current elements and resizes the list
13. remove() – Removes all the elements from the list, which are equal to given element
14. reverse() – Reverses the list
15. size() – Returns the number of elements in the list
16. sort() – Sorts the list in increasing order

|  |
| --- |
| #include <iostream>  #include <list>  #include <iterator>    void showlist(list <int> g) //function for printing the elements in a list  {      list <int> :: iterator it;      for(it = g.begin(); it != g.end(); ++it)          cout << '\t' << \*it;  }    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 : ";      showlist(gqlist2);        cout << "\ngqlist1.front() : " << gqlist1.front();      cout << "\ngqlist1.back() : " << gqlist1.back();        cout << "\ngqlist1.pop\_front() : ";      gqlist1.pop\_front();      showlist(gqlist1);        cout << "\ngqlist2.pop\_back() : ";      gqlist2.pop\_back();      showlist(gqlist2);        cout << "\ngqlist1.reverse() : ";      gqlist1.reverse();      showlist(gqlist1);        cout << "\ngqlist2.sort(): ";      gqlist2.sort();      showlist(gqlist2);        return 0;    } |

The output of the above program is :

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

#### **Deque in C++ Standard Template Library (STL)**

Double ended queues are sequence containers with the feature of expansion and contraction on both the ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements at the end, and also the beginning. Unlike vectors, contiguous storage allocation may not be guaranteed.

The functions for deque are same as [vector](http://quiz.geeksforgeeks.org/vector-sequence-containers-the-c-standard-template-library-stl-set-1), with an addition of push and pop operations for both front and back.

|  |
| --- |
| #include <iostream>  #include <deque>  using namespace std;    void showdq(deque <int> g)  {      deque <int> :: iterator it;      for (it = g.begin(); it != g.end(); ++it)          cout << '\t' << \*it;      cout << '\n';  }    int main()  {      deque <int> gquiz;      gquiz.push\_back(10);      gquiz.push\_front(20);      gquiz.push\_back(30);      gquiz.push\_front(15);      cout << "The deque gquiz is : ";      showdq(gquiz);        cout << "\ngquiz.size() : " << gquiz.size();      cout << "\ngquiz.max\_size() : " << gquiz.max\_size();        cout << "\ngquiz.at(2) : " << gquiz.at(2);      cout << "\ngquiz.front() : " << gquiz.front();      cout << "\ngquiz.back() : " << gquiz.back();        cout << "\ngquiz.pop\_front() : ";      gquiz.pop\_front();      showdq(gquiz);        cout << "\ngquiz.pop\_back() : ";      gquiz.pop\_back();      showdq(gquiz);        return 0;  } |

The deque gquiz is : 15 20 10 30

gquiz.size() : 4

gquiz.max\_size() : 4611686018427387903

gquiz.at(2) : 10

gquiz.front() : 15

gquiz.back() : 30

gquiz.pop\_front() : 20 10 30

gquiz.pop\_back() : 20 10

#### **Array class in C++**

The introduction of array class from C++11 has offered a better alternative for C-style arrays. The advantages of array class over C-style array are :-

* Array classes knows its own size, whereas C-style arrays lack this property. So when passing to functions, we don’t need to pass size of Array as a separate parameter.
* With C-style array there is more risk of [array being decayed into a pointer](https://www.geeksforgeeks.org/what-is-array-decay-in-c-how-can-it-be-prevented/). Array classes don’t decay into pointers
* Array classes are generally more efficient, light-weight and reliable than C-style arrays.

**Operation on Array**

1. **at()** :- This function is used to access the elements of array.
2. **get()** :- This function is also used to access the elements of array. This function is not the member of array class but overloaded function from class tuple.
3. **operator[]** :- This is similar to C-style arrays. This method is also used to access array elements.

|  |
| --- |
| // C++ code to demonstrate working of array,  // to() and get()  #include<iostream>  #include<array> // for array, at()  #include<tuple> // for get()  using namespace std;  int main()  {      array<int,6> ar = {1, 2, 3, 4, 5, 6};        // Printing array elements using at()      cout << "The array elemets are (using at()) : ";      for ( int i=0; i<6; i++)      cout << ar.at(i) << " ";      cout << endl;        // Printing array elements using get()      cout << "The array elemets are (using get()) : ";      cout << get<0>(ar) << " " << get<1>(ar) << " ";      cout << get<2>(ar) << " " << get<3>(ar) << " ";      cout << get<4>(ar) << " " << get<5>(ar) << " ";      cout << endl;        // Printing array elements using operator[]      cout << "The array elements are (using operator[]) : ";      for ( int i=0; i<6; i++)      cout << ar[i] << " ";      cout << endl;        return 0;  } |

The array elemets are (using at()) : 1 2 3 4 5 6

The array elemets are (using get()) : 1 2 3 4 5 6

The array elements are (using operator[]) : 1 2 3 4 5 6

1. **front()** :- This returns the first element of array.
2. **back()** :- This returns the last element of array.

|  |
| --- |
| // C++ code to demonstrate working of  // front() and back()  #include<iostream>  #include<array> // for front() and back()  using namespace std;  int main()  {      // Initializing the array elements      array<int,6> ar = {1, 2, 3, 4, 5, 6};        // Printing first element of array      cout << "First element of array is : ";      cout << ar.front() << endl;        // Printing last element of array      cout << "Last element of array is : ";      cout << ar.back() << endl;        return 0;    } |

First element of array is : 1

Last element of array is : 6

1. **size()** :- It returns the number of elements in array. This is a property that C-style arrays lack.
2. **max\_size()** :- It returns the maximum number of elements array can hold i.e, the size with which array is declared. The size() and max\_size() return the same value.

|  |
| --- |
| // C++ code to demonstrate working of size() and max\_size()  #include<iostream>  #include<array> // for size() and max\_size()  using namespace std;  int main()  {      array<int,6> ar = {1, 2, 3, 4, 5, 6};        // Printing number of array elements      cout << "The number of array elements is : "<< ar.size() << endl;        // Printing maximum elements array can hold      cout << "Max elements array can hold is : "<< ar.max\_size() << endl;        return 0;   } |

The number of array elements is : 6

Maximum elements array can hold is : 6

1. **swap()** :- The swap() swaps all elements of one array with other.

|  |
| --- |
| // C++ code to demonstrate working of swap()  #include<iostream>  #include<array> // for swap() and array  using namespace std;  int main()  {      array<int,6> ar = {1, 2, 3, 4, 5, 6};      array<int,6> ar1 = {7, 8, 9, 10, 11, 12};        // Printing 1st and 2nd array before swapping      cout << "The first array elements before swapping are : ";      for (int i=0; i<6; i++)      cout << ar[i] << " ";      cout << endl;      cout << "The second array elements before swapping are : ";      for (int i=0; i<6; i++)      cout << ar1[i] << " ";      cout << endl;        // Swapping ar1 values with ar      ar.swap(ar1);        // Printing 1st and 2nd array after swapping      cout << "The first array elements after swapping are : ";      for (int i=0; i<6; i++)      cout << ar[i] << " ";      cout << endl;      cout << "The second array elements after swapping are : ";      for (int i=0; i<6; i++)      cout << ar1[i] << " ";      cout << endl;        return 0;    } |
|  |

The first array elements before swapping are : 1 2 3 4 5 6

The second array elements before swapping are : 7 8 9 10 11 12

The first array elements after swapping are : 7 8 9 10 11 12

The second array elements after swapping are : 1 2 3 4 5 6

1. **empty()** :- This function returns true when the array size is zero else returns false.
2. **fill()** :- This function is used to fill the entire array with a particular value.

|  |
| --- |
| // C++ code to demonstrate working of empty() and fill()  #include<iostream>  #include<array> // for fill() and empty()  using namespace std;  int main()  {      array<int,6> ar;      array<int,0> ar1;        // Checking size of array if it is empty      ar1.empty()? cout << "Array empty":          cout << "Array not empty";      cout << endl;        ar.fill(0);     // Filling array with 0        // Displaying array after filling      cout << "Array after filling operation is : ";      for ( int i=0; i<6; i++)          cout << ar[i] << " ";        return 0;  } |

Array empty

Array after filling operation is : 0 0 0 0 0 0

#### **Forward List in C++ | Set 1 (Introduction and Important Functions)**

Forward list in STL implements singly linked list. Introduced from C++11, forward list are useful than other containers in insertion, removal and moving operations (like sort) and allows time constant insertion and removal of elements.

It differs from [list](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/) by the fact that forward list keeps track of location of only next element while list keeps track to both next and previous elements, thus increasing the storage space required to store each element. The drawback of forward list is that it cannot be iterated backwards and its individual elements cannot be accessed directly.

Forward List is preferred over list when only forward traversal is required (same as singly linked list is preferred over doubly linked list) as we can save space. Some example cases are, chaining in hashing, adjacency list representation of graph, etc.

**Operations:**

1. **assign()**:- This function is used to assign values to forward list, its another variant is used to assign repeated elements.

|  |
| --- |
| // C++ code to demonstrate forward list and assign()  #include<iostream>  #include<forward\_list>    int main()  {      forward\_list<int> flist1;        forward\_list<int> flist2;        // Assigning values using assign()      flist1.assign({1, 2, 3});        // Assigning repeating val using assign() 5 elements with value 10      flist2.assign(5, 10);        // Displaying forward lists      cout << "The elements of first forward list are : ";      for (int&a : flist1)          cout << a << " ";      cout << endl;        cout << "The elements of second forward list are : ";      for (int&b : flist2)          cout << b << " ";      cout << endl;        return 0;  } |

The elements of first forward list are : 1 2 3

The elements of second forward list are : 10 10 10 10 10

1. **push\_front()** :- This function is used to insert the element at the first position on forward list. The value from this function is copied to the space before first element in the container. The size of forward list increases by 1.
2. **emplace\_front()** :- This function is similar to the previous function but in this no copying operation occurs, the element is created directly at the memory before the first element of the forward list.
3. **pop\_front()** :- This function is used to delete the first element of list.

// C++ code to demonstrate working of push\_front(), emplace\_front() and pop\_front()

|  |
| --- |
| #include<iostream>  #include<forward\_list>    int main()  {      forward\_list<int> flist = {10, 20, 30, 40, 50};        // Inserting value using push\_front() Inserts 60 at front      flist.push\_front(60);        // Displaying the forward list      cout << "The forward list after push\_front operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Inserting value using emplace\_front() Inserts 70 at front      flist.emplace\_front(70);        // Displaying the forward list      cout << "The forward list after emplace\_front operation : ";      for (int&c : flist)         cout << c << " ";      cout << endl;        // Deleting first value using pop\_front() Pops 70      flist.pop\_front();        // Displaying the forward list      cout << "The forward list after pop\_front operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        return 0;  } |

The forward list after push\_front operation : 60 10 20 30 40 50

The forward list after emplace\_front operation : 70 60 10 20 30 40 50

The forward list after pop\_front operation : 60 10 20 30 40 50

**4. insert\_after()** This function gives us a choice to insert elements at any position in forward list. The arguments in this function are copied at the desired position.

**5. emplace\_after()** This function also does the same operation as above function but the elements are directly made without any copy operation.

**6. erase\_after()**This function is used to erase elements from a particular position in the forward list.

|  |
| --- |
| // C++ code to demonstrate working of  // insert\_after(), emplace\_after() and erase\_after()  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist = {10, 20, 30} ;        // Declaring a forward list iterator      forward\_list<int>::iterator ptr;        // Inserting value using insert\_after()      // starts insertion from second position      ptr =  flist.insert\_after(flist.begin(), {1, 2, 3});        // Displaying the forward list      cout << "The forward list after insert\_after operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Inserting value using emplace\_after()      // inserts 2 after ptr      ptr = flist.emplace\_after(ptr,2);        // Displaying the forward list      cout << "The forward list after emplace\_after operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Deleting value using erase.after Deleted 2      // after ptr      ptr = flist.erase\_after(ptr);        // Displaying the forward list      cout << "The forward list after erase\_after operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        return 0;  } |

Run on IDE

Output:

The forward list after insert\_after operation : 10 1 2 3 20 30

The forward list after emplace\_after operation : 10 1 2 3 2 20 30

The forward list after erase\_after operation : 10 1 2 3 2 30

**7. remove()** :- This function removes the particular element from the forward list mentioned in its argument.

**8. remove\_if()** :- This function removes according to the condition in its argument.

|  |
| --- |
| // C++ code to demonstrate working of remove() and  // remove\_if()  #include<iostream>  #include<forward\_list>  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist = {10, 20, 30, 25, 40, 40};        // Removing element using remove()      // Removes all occurrences of 40      flist.remove(40);        // Displaying the forward list      cout << "The forward list after remove operation : ";      for (int&c : flist)          cout << c << " ";      cout << endl;        // Removing according to condition. Removes      // elements greater than 20. Removes 25 and 30      flist.remove\_if([](int x){ return x>20;});        // Displaying the forward list      cout << "The forward list after remove\_if operation : ";      for (int&c : flist)         cout << c << " ";      cout << endl;        return 0;    } |

Run on IDE

Output:

The forward list after remove operation : 10 20 30 25

The forward list after remove\_if operation : 10 20

**9. splice\_after()** :- This function transfers elements from one forward list to other.

|  |
| --- |
| // C++ code to demonstrate working of  // splice\_after()  #include<iostream>  #include<forward\_list> // for splice\_after()  using namespace std;    int main()  {      // Initializing forward list      forward\_list<int> flist1 = {10, 20, 30};        // Initializing second list      forward\_list<int> flist2 = {40, 50, 60};        // Shifting elements from first to second      // forward list after 1st position      flist2.splice\_after(flist2.begin(),flist1);        // Displaying the forward list      cout << "The forward list after splice\_after operation : ";      for (int&c : flist2)         cout << c << " ";      cout << endl;        return 0;  } |

Run on IDE

Output:

The forward list after splice\_after operation : 40 10 20 30 50 60

### **Container** **Adaptors**:

#### **Queue in Standard Template Library (STL)**

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

**The functions supported by queue are :**

1. empty() – Returns whether the queue is empty
2. size() – Returns the size of the queue
3. front() – Returns a reference to the first element of the queue
4. back() – Returns a reference to the last element of the queue
5. push(g) – Adds the element ‘g’ at the end of the queue
6. pop() – Deletes the first element of the queue

|  |
| --- |
| // CPP code to illustrate Queue in Standard Template Library (STL)  #include <iostream>  #include <queue>    void showq(queue <int> gq)  {      queue <int> g = gq;      while (!g.empty())      {          cout << '\t' << g.front();          g.pop();      }  }  int main()  {      queue <int> gquiz;      gquiz.push(10);      gquiz.push(20);      gquiz.push(30);        cout << "The queue gquiz is : ";      showq(gquiz);        cout << "\ngquiz.size() : " << gquiz.size();      cout << "\ngquiz.front() : " << gquiz.front();      cout << "\ngquiz.back() : " << gquiz.back();        cout << "\ngquiz.pop() : ";      gquiz.pop();      showq(gquiz);        return 0;  } |

The queue gquiz is : 10 20 30

gquiz.size() : 3

gquiz.front() : 10

gquiz.back() : 30

gquiz.pop() : 20 30

#### **Priority Queue in C++ Standard Template Library (STL)**

Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non decreasing order(hence we can see that each element of the queue has a priority{fixed order}).

The functions associated with priority queue are:

1. empty() – Returns whether the queue is empty
2. size() – Returns the size of the queue
3. top() – Returns a reference to the top most element of the queue
4. push(g) – Adds the element ‘g’ at the end of the queue
5. pop() – Deletes the first element of the queue

#include <iostream>

#include <queue>

using namespace std;

void showpq(priority\_queue <int> gq)

{

priority\_queue <int> g = gq;

while (!g.empty())

{

cout << '\t' << g.top();

g.pop();

}

}

int main ()

{

priority\_queue <int> gquiz;

gquiz.push(10);

gquiz.push(30);

gquiz.push(20);

gquiz.push(5);

gquiz.push(1);

cout << "The priority queue gquiz is : ";

showpq(gquiz);

cout << "\ngquiz.size() : " << gquiz.size();

cout << "\ngquiz.top() : " << gquiz.top();

cout << "\ngquiz.pop() : ";

gquiz.pop();

showpq(gquiz);

return 0;

}

The priority queue gquiz is : 30 20 10 5 1

gquiz.size() : 5

gquiz.top() : 30

gquiz.pop() : 20 10 5 1

#### **Stack in C++ Standard Template Library (STL)**

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

The functions associated with stack are:

1. empty() – Returns whether the stack is empty
2. size() – Returns the size of the stack
3. top() – Returns a reference to the top most element of the stack
4. push(g) – Adds the element ‘g’ at the top of the stack
5. pop() – Deletes the top most element of the stack

|  |
| --- |
| #include <iostream>  #include <stack>    void showstack(stack <int> gq)  {      stack <int> g = gq;      while (!g.empty())      {          cout << '\t' << g.top();          g.pop();      }  }    int main ()  {      stack <int> gquiz;      gquiz.push(10);      gquiz.push(30);      gquiz.push(20);      gquiz.push(5);      gquiz.push(1);        cout << "The stack gquiz is : ";      showstack(gquiz);        cout << "\ngquiz.size() : " << gquiz.size();      cout << "\ngquiz.top() : " << gquiz.top();          cout << "\ngquiz.pop() : ";      gquiz.pop();      showstack(gquiz);        return 0;  } |

The stack gquiz is : 1 5 20 30 10

gquiz.size() : 5

gquiz.top() : 1

gquiz.pop() : 5 20 30 10

### **Associative Containers :**

Implement sorted data structures that can be quickly searched (O(log n)complexity).

#### **Set in C++ Standard Template Library (STL)**

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Functions associated with Set:

1. begin() – Returns an iterator to the first element in the set
2. end() – Returns an iterator to the theoretical element that follows last element in the set
3. size() – Returns the number of elements in the set
4. max\_size() – Returns the maximum number of elements that the set can hold
5. empty() – Returns whether the set is empty
6. pair <iterator, bool> insert(const g) – Adds a new element ‘g’ to the set
7. iterator insert (iterator position, const g) – Adds a new element ‘g’ at the position pointed by iterator
8. erase(iterator position) – Removes the element at the position pointed by the iterator
9. erase(const g)- Removes the value ‘g’ from the set
10. clear() – Removes all the elements from the set
11. key\_comp() / value\_comp() – Returns the object that determines how the elements in the set are ordered (‘<‘ by default)
12. find(const g) – Returns an iterator to the element ‘g’ in the set if found, else returns the iterator to end
13. count(const g) – Returns 1 or 0 based on the element ‘g’ is present in the set or not.
14. lower\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the set
15. upper\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will go after the element ‘g’ in the set

|  |
| --- |
| #include <iostream>  #include <set>  #include <iterator>  using namespace std;    int main()  {      // empty set container      set <int, greater <int> > gquiz1;        // insert elements in random order      gquiz1.insert(40);      gquiz1.insert(30);      gquiz1.insert(60);      gquiz1.insert(20);      gquiz1.insert(50);      gquiz1.insert(50); // only one 50 will be added to the set      gquiz1.insert(10);        // printing set gquiz1      set <int, greater <int> > :: iterator itr;      cout << "\nThe set gquiz1 is : ";      for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)      {          cout << '\t' << \*itr;      }      cout << endl;        // assigning the elements from gquiz1 to gquiz2      set <int> gquiz2(gquiz1.begin(), gquiz1.end());        // print all elements of the set gquiz2      cout << "\nThe set gquiz2 after assign from gquiz1 is : ";      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << \*itr;      }      cout << endl;        // remove all elements up to 30 in gquiz2      cout << "\ngquiz2 after removal of elements less than 30 : ";      gquiz2.erase(gquiz2.begin(), gquiz2.find(30));      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << \*itr;      }        // remove element with value 50 in gquiz2      int num;      num = gquiz2.erase (50);      cout << "\ngquiz2.erase(50) : ";      cout << num << " removed \t" ;      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << \*itr;      }        cout << endl;        //lower bound and upper bound for set gquiz1      cout << "gquiz1.lower\_bound(40) : "           << \*gquiz1.lower\_bound(40) << endl;      cout << "gquiz1.upper\_bound(40) : "           << \*gquiz1.upper\_bound(40) << endl;        //lower bound and upper bound for set gquiz2      cout << "gquiz2.lower\_bound(40) : "           << \*gquiz2.lower\_bound(40) << endl;      cout << "gquiz2.upper\_bound(40) : "           << \*gquiz2.upper\_bound(40) << endl;        return 0;    } |

The output of the above program is :

The set gquiz1 is : 60 50 40 30 20 10

The set gquiz2 after assign from gquiz1 is : 10 20 30 40 50 60

gquiz2 after removal of elements less than 30 : 30 40 50 60

gquiz2.erase(50) : 1 removed 30 40 60

gquiz1.lower\_bound(40) : 40

gquiz1.upper\_bound(40) : 30

gquiz2.lower\_bound(40) : 40

gquiz2.upper\_bound(40) : 60

#### **Multiset in C++ Standard Template Library (STL)**

Multisets are a type of associative containers similar to set, with an exception that multiple elements can have same values.

Functions associated with multiset:

1. begin() – Returns an iterator to the first element in the multiset
2. end() – Returns an iterator to the theoretical element that follows last element in the multiset
3. size() – Returns the number of elements in the multiset
4. max\_size() – Returns the maximum number of elements that the multiset can hold
5. empty() – Returns whether the multiset is empty
6. pair insert(const g) – Adds a new element ‘g’ to the multiset
7. iterator insert (iterator position,const g) – Adds a new element ‘g’ at the position pointed by iterator
8. erase(iterator position) – Removes the element at the position pointed by the iterator
9. erase(const g)- Removes the value ‘g’ from the multiset
10. clear() – Removes all the elements from the multiset
11. key\_comp() / value\_comp() – Returns the object that determines how the elements in the multiset are ordered (‘<' by default)
12. find(const g) – Returns an iterator to the element ‘g’ in the multiset if found, else returns the iterator to end
13. count(const g) – Returns the number of matches to element ‘g’ in the multiset
14. lower\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will not go before the element ‘g’ in the multiset
15. upper\_bound(const g) – Returns an iterator to the first element that is equivalent to ‘g’ or definitely will go after the element ‘g’ in the multiset

|  |
| --- |
| #include <iostream>  #include <set>  #include <iterator>    using namespace std;    int main()  {      // empty multiset container      multiset <int, greater <int> > gquiz1;        // insert elements in random order      gquiz1.insert(40);      gquiz1.insert(30);      gquiz1.insert(60);      gquiz1.insert(20);      gquiz1.insert(50);      gquiz1.insert(50); // 50 will be added again to the multiset unlike set      gquiz1.insert(10);        // printing multiset gquiz1      multiset <int, greater <int> > :: iterator itr;      cout << "\nThe multiset gquiz1 is : ";      for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)      {          cout << '\t' << \*itr;      }      cout << endl;        // assigning the elements from gquiz1 to gquiz2      multiset <int> gquiz2(gquiz1.begin(), gquiz1.end());        // print all elements of the multiset gquiz2      cout << "\nThe multiset gquiz2 after assign from gquiz1 is : ";      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << \*itr;      }      cout << endl;        // remove all elements up to element with value 30 in gquiz2      cout << "\ngquiz2 after removal of elements less than 30 : ";      gquiz2.erase(gquiz2.begin(), gquiz2.find(30));      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << \*itr;      }        // remove all elements with value 50 in gquiz2      int num;      num = gquiz2.erase(50);      cout << "\ngquiz2.erase(50) : ";      cout << num << " removed \t" ;      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << \*itr;      }        cout << endl;        //lower bound and upper bound for multiset gquiz1      cout << "gquiz1.lower\_bound(40) : "           << \*gquiz1.lower\_bound(40) << endl;      cout << "gquiz1.upper\_bound(40) : "           << \*gquiz1.upper\_bound(40) << endl;        //lower bound and upper bound for multiset gquiz2      cout << "gquiz2.lower\_bound(40) : "           << \*gquiz2.lower\_bound(40) << endl;      cout << "gquiz2.upper\_bound(40) : "           << \*gquiz2.upper\_bound(40) << endl;             return 0;    } |

The output of the above program is :

The multiset gquiz1 is : 60 50 50 40 30 20 10

The multiset gquiz2 after assign from gquiz1 is : 10 20 30 40 50 50 60

gquiz2 after removal of elements less than 30 : 30 40 50 50 60

gquiz2.erase(50) : 2 removed 30 40 60

gquiz1.lower\_bound(40) : 40

gquiz1.upper\_bound(40) : 30

gquiz2.lower\_bound(40) : 40

gquiz2.upper\_bound(40) : 60

#### **Map in C++ Standard Template Library (STL)**

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

Functions associated with Map:

1. begin() – Returns an iterator to the first element in the map
2. end() – Returns an iterator to the theoretical element that follows last element in the map
3. size() – Returns the number of elements in the map
4. max\_size() – Returns the maximum number of elements that the map can hold
5. empty() – Returns whether the map is empty
6. pair insert(keyvalue,mapvalue) – Adds a new element to the map
7. erase(iterator position) – Removes the element at the position pointed by the iterator
8. erase(const g)- Removes the key value ‘g’ from the map
9. clear() – Removes all the elements from the map
10. key\_comp() / value\_comp() – Returns the object that determines how the elements in the map are ordered (‘<' by default)
11. find(const g) – Returns an iterator to the element with key value ‘g’ in the map if found, else returns the iterator to end
12. count(const g) – Returns the number of matches to element with key value ‘g’ in the map
13. lower\_bound(const g) – Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the map
14. upper\_bound(const g) – Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the map

|  |
| --- |
| #include <iostream>  #include <map>  #include <iterator>    using namespace std;    int main()  {      map <int, int> gquiz1;        // empty map container        // insert elements in random order      gquiz1.insert(pair <int, int> (1, 40));      gquiz1.insert(pair <int, int> (2, 30));      gquiz1.insert(pair <int, int> (3, 60));      gquiz1.insert(pair <int, int> (4, 20));      gquiz1.insert(pair <int, int> (5, 50));      gquiz1.insert(pair <int, int> (6, 50));      gquiz1.insert(pair <int, int> (7, 10));        // printing map gquiz1      map <int, int> :: iterator itr;      cout << "\nThe map gquiz1 is : \n";      cout << "\tKEY\tELEMENT\n";      for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)      {          cout  <<  '\t' << itr->first                <<  '\t' << itr->second << '\n';      }      cout << endl;        // assigning the elements from gquiz1 to gquiz2      map <int, int> gquiz2(gquiz1.begin(), gquiz1.end());        // print all elements of the map gquiz2      cout << "\nThe map gquiz2 after assign from gquiz1 is : \n";      cout << "\tKEY\tELEMENT\n";      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout  <<  '\t' << itr->first                <<  '\t' << itr->second << '\n';      }      cout << endl;        // remove all elements up to element with key=3 in gquiz2      cout << "\ngquiz2 after removal of elements less than key=3 : \n";      cout << "\tKEY\tELEMENT\n";      gquiz2.erase(gquiz2.begin(), gquiz2.find(3));      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout  <<  '\t' << itr->first                <<  '\t' << itr->second << '\n';      }        // remove all elements with key = 4      int num;      num = gquiz2.erase (4);      cout << "\ngquiz2.erase(4) : ";      cout << num << " removed \n" ;      cout << "\tKEY\tELEMENT\n";      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout  <<  '\t' << itr->first                <<  '\t' << itr->second << '\n';      }        cout << endl;        //lower bound and upper bound for map gquiz1 key = 5      cout << "gquiz1.lower\_bound(5) : " << "\tKEY = ";      cout << gquiz1.lower\_bound(5)->first << '\t';      cout << "\tELEMENT = " << gquiz1.lower\_bound(5)->second << endl;      cout << "gquiz1.upper\_bound(5) : " << "\tKEY = ";      cout << gquiz1.upper\_bound(5)->first << '\t';      cout << "\tELEMENT = " << gquiz1.upper\_bound(5)->second << endl;        return 0;    } |

Run on IDE

The output of the above program is :

The map gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 20

5 50

6 50

7 10

The map gquiz2 after assign from gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 20

5 50

6 50

7 10

gquiz2 after removal of elements less than key=3 :

KEY ELEMENT

3 60

4 20

5 50

6 50

7 10

gquiz2.erase(4) : 1 removed

KEY ELEMENT

3 60

5 50

6 50

7 10

gquiz1.lower\_bound(5) : KEY = 5 ELEMENT = 50

gquiz1.upper\_bound(5) : KEY = 6 ELEMENT = 50

#### **Multimap in C++ Standard Template Library (STL)**

Multimap is similar to mapwith an addition that multiple elements can have same keys. Rather than each element being unique, the key value and mapped value pair has to be unique in this case.

**Functions associated with multimap:**

1. **begin()** – Returns an iterator to the first element in the multimap
2. **end()** – Returns an iterator to the theoretical element that follows last element in the multimap
3. **size()** – Returns the number of elements in the multimap
4. **max\_size()** – Returns the maximum number of elements that the multimap can hold
5. **empty()** – Returns whether the multimap is empty
6. **pair<int,int> insert(keyvalue,multimapvalue**) – Adds a new element to the multimap
7. **erase(iterator position)** – Removes the element at the position pointed by the iterator
8. **erase(const g)**– Removes the key value ‘g’ from the multimap
9. **clear() –** Removes all the elements from the multimap
10. **key\_comp() / value\_comp()** – Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default)
11. **find(const g)** – Returns an iterator to the element with key value ‘g’ in the multimap if found, else returns the iterator to end
12. **count(const g)** – Returns the number of matches to element with key value ‘g’ in the multimap
13. **lower\_bound(const g)** – Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will not go before the element with key value ‘g’ in the multimap
14. **upper\_bound(const g)** – Returns an iterator to the first element that is equivalent to multimapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the multimap

**C++ implementation to illustrate above functions**

|  |
| --- |
| #include <iostream>  #include <map>  #include <iterator>  using namespace std;    int main()  {      multimap <int, int> gquiz1;        // empty multimap container        // insert elements in random order      gquiz1.insert(pair <int, int> (1, 40));      gquiz1.insert(pair <int, int> (2, 30));      gquiz1.insert(pair <int, int> (3, 60));      gquiz1.insert(pair <int, int> (4, 20));      gquiz1.insert(pair <int, int> (5, 50));      gquiz1.insert(pair <int, int> (6, 50));      gquiz1.insert(pair <int, int> (6, 10));        // printing multimap gquiz1      multimap <int, int> :: iterator itr;      cout << "\nThe multimap gquiz1 is : \n";      cout << "\tKEY\tELEMENT\n";      for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)      {          cout  <<  '\t' << itr->first                <<  '\t' << itr->second << '\n';      }      cout << endl;        // assigning the elements from gquiz1 to gquiz2      multimap <int, int> gquiz2(gquiz1.begin(),gquiz1.end());        // print all elements of the multimap gquiz2      cout << "\nThe multimap gquiz2 after assign from gquiz1 is : \n";      cout << "\tKEY\tELEMENT\n";      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << itr->first               << '\t' << itr->second << '\n';      }      cout << endl;        // remove all elements up to element with value 30 in gquiz2      cout << "\ngquiz2 after removal of elements less than key=3 : \n";      cout << "\tKEY\tELEMENT\n";      gquiz2.erase(gquiz2.begin(), gquiz2.find(3));      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << itr->first               << '\t' << itr->second << '\n';      }        // remove all elements with key = 4      int num;      num = gquiz2.erase(4);      cout << "\ngquiz2.erase(4) : ";      cout << num << " removed \n" ;      cout << "\tKEY\tELEMENT\n";      for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)      {          cout << '\t' << itr->first               << '\t' << itr->second << '\n';      }        cout << endl;        //lower bound and upper bound for multimap gquiz1 key = 5      cout << "gquiz1.lower\_bound(5) : " << "\tKEY = ";      cout << gquiz1.lower\_bound(5)->first << '\t';      cout << "\tELEMENT = " << gquiz1.lower\_bound(5)->second << endl;      cout << "gquiz1.upper\_bound(5) : " << "\tKEY = ";      cout << gquiz1.upper\_bound(5)->first << '\t';      cout << "\tELEMENT = " << gquiz1.upper\_bound(5)->second << endl;        return 0;    } |

Output:

The multimap gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 20

5 50

6 50

6 10

The multimap gquiz2 after assign from gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 20

5 50

6 50

6 10

gquiz2 after removal of elements less than key=3 :

KEY ELEMENT

3 60

4 20

5 50

6 50

6 10

gquiz2.erase(4) : 1 removed

KEY ELEMENT

3 60

5 50

6 50

6 10

gquiz1.lower\_bound(5) : KEY = 5 ELEMENT = 50

gquiz1.upper\_bound(5) : KEY = 6 ELEMENT = 50

## **Functors in C++**

Please note that the title is **Functors** (Not Functions)!!

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

**Functors** are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

Below program uses [transform() in STL](https://www.geeksforgeeks.org/transform-c-stl-perform-operation-elements/) to add 1 to all elements of arr[].

|  |
| --- |
| // A C++ program uses transform() in STL to add 1 to all elements of arr[]  #include <bits/stdc++.h>  using namespace std;    int increment(int x) {  return (x+1); }    int main()  {      int arr[] = {1, 2, 3, 4, 5};      int n = sizeof(arr)/sizeof(arr[0]);        // Apply increment to all elements of      // arr[] and store the modified elements back in arr[]      transform(arr, arr+n, arr, increment);        for (int i=0; i<n; i++)          cout << arr[i] << S" ";        return 0;  } |

2 3 4 5 6

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what’s happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the operator().

**The line,**

MyFunctor(10);

**Is same as**

MyFunctor.operator()(10);

Let’s delve deeper and understand how this can actually be used in conjunction with STLs.

|  |
| --- |
| // C++ program to demonstrate working of  // functors.  #include <bits/stdc++.h>  using namespace std;    // A Functor  class increment  {  private:      int num;  public:      increment(int n) : num(n) {  }        // This operator overloading enables calling      // operator function () on objects of increment      int operator () (int arr\_num) const {          return num + arr\_num;      }  };    // Driver code  int main()  {      int arr[] = {1, 2, 3, 4, 5};      int n = sizeof(arr)/sizeof(arr[0]);      int to\_add = 5;        transform(arr, arr+n, arr, increment(to\_add));        for (int i=0; i<n; i++)          cout << arr[i] << " ";  } |

Output:

6 7 8 9 10

Thus, here, Increment is a functor, a c++ class that acts as a function.

**The line,**

transform(arr, arr+n, arr, increment(to\_add));

**is the same as writing below two lines,**

// Creating object of increment

increment obj(to\_add);

// Calling () on object

transform(arr, arr+n, arr, obj);

## **Iterators in C++ STL**

Iterators are used to point at the memory addresses of [STL](http://quiz.geeksforgeeks.org/the-c-standard-template-library-stl/) containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

**Operations of iterators** :-

1. **begin()** :- This function is used to return the **beginning position** of the container.
2. **end()** :- This function is used to return the**end position** of the container.

|  |
| --- |
| // C++ code to demonstrate the working of iterator, begin() and end()  #include<iostream>  #include<iterator> // for iterators  #include<vector> // for vectors  using namespace std;  int main()  {      vector<int> ar = { 1, 2, 3, 4, 5 };        // Declaring iterator to a vector      vector<int>::iterator ptr;        // Displaying vector elements using begin() and end()      cout << "The vector elements are : ";      for (ptr = ar.begin(); ptr < ar.end(); ptr++)          cout << \*ptr << " ";        return 0;  } |

The vector elements are : 1 2 3 4 5

1. **advance() :- This function is used to increment the iterator position till the specified number mentioned in its arguments.**

|  |
| --- |
| // C++ code to demonstrate the working of  // advance()  #include<iostream>  #include<iterator> // for iterators  #include<vector> // for vectors  using namespace std;  int main()  {      vector<int> ar = { 1, 2, 3, 4, 5 };        // Declaring iterator to a vector      vector<int>::iterator ptr = ar.begin();        // Using advance() to increment iterator position      // points to 4      advance(ptr, 3);        // Displaying iterator position      cout << "The position of iterator after advancing is : ";      cout << \*ptr << " ";        return 0;    } |

The position of iterator after advancing is : 4

1. **next()** :- This function **returns the new iterator** that the iterator would point after **advancing the positions**mentioned in its arguments.
2. **prev()** :- This function **returns the new iterator** that the iterator would point **after decrementing the positions**mentioned in its arguments.

|  |
| --- |
| // C++ code to demonstrate the working of  // next() and prev()  #include<iostream>  #include<iterator> // for iterators  #include<vector> // for vectors  using namespace std;  int main()  {      vector<int> ar = { 1, 2, 3, 4, 5 };       vector<int>::iterator ptr = ar.begin();      vector<int>::iterator ftr = ar.end();          // Using next() to return new iterator points to 4      auto it = next(ptr, 3);        // Using prev() to return new iterator Points to 3      auto it1 = prev(ftr, 3);        // Displaying iterator position      cout << "The position of new iterator using next() is : ";      cout << \*it << " ";      cout << endl;        // Displaying iterator position      cout << "The position of new iterator using prev()  is : ";      cout << \*it1 << " ";      cout << endl;        return 0;  } |

The position of new iterator using next() is : 4

The position of new iterator using prev() is : 3

1. **inserter()** :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.

|  |
| --- |
| // C++ code to demonstrate the working of inserter()  #include<iostream>  #include<iterator> // for iterators  #include<vector> // for vectors  using namespace std;  int main()  {      vector<int> ar = { 1, 2, 3, 4, 5 };      vector<int> ar1 = {10, 20, 30};        // Declaring iterator to a vector      vector<int>::iterator ptr = ar.begin();        // Using advance to set position      advance(ptr, 3);        // copying 1 vector elements in other using inserter()      // inserts ar1 after 3rd position in ar      copy(ar1.begin(), ar1.end(), inserter(ar,ptr));        // Displaying new vector elements      cout << "The new vector after inserting elements is : ";      for (int &x : ar)          cout << x << " ";        return 0;  } |

Output:

The new vector after inserting elements is : 1 2 3 10 20 30 4 5

## **Pair in C++ Standard Template Library (STL)**

The pair container is a simple container defined in **<utility>** header consisting of two data elements or objects.

* The first element is referenced as ‘first’ and the second element as ‘second’ and the order is fixed (first, second).
* Pair is used to combine together two values which may be different in type. Pair provides a way to store two heterogeneous objects as a single unit.
* Pair can be assigned, copied and compared. The array of objects allocated in a map or hash\_map are of type ‘pair’ by default in which all the ‘first’ elements are unique keys associated with their ‘second’ value objects.
* To access the elements, we use variable name followed by dot operator followed by the keyword first or second.

**Syntax :**

pair (data\_type1, data\_type2) Pair\_name;

|  |
| --- |
| //CPP program to illustrate pair STL  #include <iostream>  #include <utility>  using namespace std;    int main()  {      pair <int, char> PAIR1 ;        PAIR1.first = 100;      PAIR1.second = 'G' ;        cout << PAIR1.first << " " ;      cout << PAIR1.second << endl ;        return 0;  } |

100 G

### **Initializing a pair**

**Syntax :**

pair (data\_type1, data\_type2) Pair\_name (value1, value2) ;

Different ways to initialize pair:

pair g1; //default

pair g2(1, 'a'); //initialized, different data type

pair g3(1, 10); //initialized, same data type

pair g4(g3); //copy of g3

Another way to initialize a pair is by using the make\_pair() function.

g2 = make\_pair(1, 'a');

|  |
| --- |
| //CPP program to illustrate Initializing of pair STL  #include <iostream>  #include <utility>  using namespace std;    int main()  {      pair <string,double> PAIR2 ("GeeksForGeeks", 1.23);        cout << PAIR2.first << " " ;      cout << PAIR2.second << endl ;    return 0;  } |

GeeksForGeeks 1.23

**Note:** If not initialized, the first value of the pair gets automatically initialized.

|  |
| --- |
| //CPP program to illustrate auto-initializing of pair STL  #include <iostream>  #include <utility>    using namespace std;    int main()  {      pair <int, double> PAIR1 ;      pair <string, char> PAIR2 ;        cout << PAIR1.first ;  //it is initialised to 0      cout << PAIR1.second ; //it is initialised to 0        cout << " ";        cout << PAIR2.first ;  //it prints nothing i.e NULL      cout << PAIR2.second ; //it prints nothing i.e NULL        return 0;  } |

00

### **Member** **Functions**

#### **make\_pair()** :

This template function allows to create a value pair without writing the types explicitly.  
**Syntax :**

Pair\_name = make\_pair (value1,value2);

|  |
| --- |
| #include <iostream>  #include <utility>  using namespace std;    int main()  {      pair <int, char> PAIR1 ;      pair <string, double> PAIR2 ("GeeksForGeeks", 1.23) ;      pair <string, double> PAIR3 ;        PAIR1.first = 100;      PAIR1.second = 'G' ;        PAIR3 = make\_pair ("GeeksForGeeks is Best",4.56);        cout << PAIR1.first << " " ;      cout << PAIR1.second << endl ;        cout << PAIR2.first << " " ;      cout << PAIR2.second << endl ;        cout << PAIR3.first << " " ;      cout << PAIR3.second << endl ;        return 0;  } |

100 G

GeeksForGeeks 1.23

GeeksForGeeks is Best 4.56

#### **operators(=, ==, !=, >=, <=) :**

We can use operators with pairs as well.

1. **using equal(=)** : It assigns new object for a pair object.

pair& operator= (const pair& pr);

This Assigns pr as the new content for the pair object. The first value is assigned the first value of pr and the second value is assigned the second value of pr .

1. **Comparison (==) operator with pair :** For given two pairs say pair1 and pair2, the comparison operator compares the first value and second value of those two pairs i.e. if pair1.first is equal to pair2.first or not AND if pair1.second is equal to pair2.second or not .
2. **Not equal (!=) operator with pair :** For given two pairs say pair1 and pair2, the != operator compares the first values of those two pairs i.e. if pair1.first is equal to pair2.first or not, if they are equal then it checks the second values of both.
3. **Logical( >=, <= )operators with pair :** For given two pairs say pair1 and pair2, the =, >, can be used with pairs as well.

.

|  |
| --- |
| //CPP code to illustrate operators in pair  #include <iostream>  #include<utility>  using namespace std;    int main()  {      pair<int, int>pair1 = make\_pair(1, 12);      pair<int, int>pair2 = make\_pair(9, 12);          cout << (pair1 == pair2) << endl;      cout << (pair1 != pair2) << endl;      cout << (pair1 >= pair2) << endl;      cout << (pair1 <= pair2) << endl;      cout << (pair1 > pair2) << endl;      cout << (pair1 < pair2) << endl;        return 0;  } |

0

1

0

1

0

1

#### **swap**

This function swaps the contents of one pair object with the contents of another pair object. The pairs must be of same type.

**Syntax :**

pair1.swap(pair2) ;

For two given pairs say pair1 and pair2 of same type, swap function will swap the pair1.first with pair2.first and pair1.second with pair2.second.

|  |
| --- |
| #include <iostream>  #include<utility>    using namespace std;    int main()  {      pair<char, int>pair1 = make\_pair('A', 1);      pair<char, int>pair2 = make\_pair('B', 2);        cout << "Before swapping:\n " ;      cout << "Contents of pair1 = " << pair1.first << " " << pair1.second ;      cout << "Contents of pair2 = " << pair2.first << " " << pair2.second ;      pair1.swap(pair2);        cout << "\nAfter swapping:\n ";      cout << "Contents of pair1 = " << pair1.first << " " << pair1.second ;      cout << "Contents of pair2 = " << pair2.first << " " << pair2.second ;        return 0;  } |

Before swapping:

Contents of pair1 = (A, 1)

Contents of pair2 = (B, 2)

After swapping:

Contents of pair1 = (B, 2)

Contents of pair2 = (A, 1)

|  |
| --- |
| //CPP program to illustrate pair in STL  #include <iostream>  #include <utility>  #include <string>    int main()  {      pair <string, int> g1;      pair <string, int> g2("Quiz", 3);      pair <string, int> g3(g2);      pair <int, int> g4(5, 10);        g1 = make\_pair(string("Geeks"), 1);      g2.first = ".com";      g2.second = 2;        cout << "This is pair g" << g1.second << " with "          << "value " << g1.first << "." << endl << endl;        cout << "This is pair g" << g3.second          << " with value " << g3.first          << "This pair was initialized as a copy of "          << "pair g2" << endl << endl;        cout << "This is pair g" << g2.second          << " with value " << g2.first          << "\nThe values of this pair were"          << " changed after initialization."          << endl << endl;        cout << "This is pair g4 with values "          << g4.first << " and " << g4.second          << " made for showing addition. \nThe "          << "sum of the values in this pair is "          << g4.first+g4.second          << "." << endl << endl;        cout << "We can concatenate the values of"          << " the pairs g1, g2 and g3 : "          << g1.first + g3.first + g2.first << endl << endl;        cout << "We can also swap pairs "          << "(but type of pairs should be same) : " << endl;      cout << "Before swapping, " << "g1 has " << g1.first          << " and g2 has " << g2.first << endl;      swap(g1, g2);      cout << "After swapping, "          << "g1 has " << g1.first << " and g2 has " << g2.first;        return 0;  } |

This is pair g1 with value Geeks.

This is pair g3 with value QuizThis pair was initialized as a copy of pair g2

This is pair g2 with value .com

The values of this pair were changed after initialization.

This is pair g4 with values 5 and 10 made for showing addition.

The sum of the values in this pair is 15.

We can concatenate the values of the pairs g1, g2 and g3 : GeeksQuiz.com

We can also swap pairs (but type of pairs should be same) :

Before swapping, g1 has Geeks and g2 has .com

After swapping, g1 has .com and g2 has Geeks