# MQF633 C++ FOR FINANCIAL ENGINEERING

# Lecture 6: C++ Standard Template Library

## Part I: C++ Standard Template Library

#### Definition

*STL is a powerful library in C++ that provides several generic classes and functions with templates to implement popular data structures and algorithms. It's designed to work seamlessly with various data types.*

#### Key Components

#### Containers

*Containers are objects that store data. Some commonly used containers include vectors, lists, queues, stacks, sets, maps, etc.*

#### Algorithms

*STL provides a set of generic algorithms that operate on these containers. These algorithms include sorting, searching, and manipulation functions, making it easy to perform common operations on different container types.*

#### Iterators

*Iterators are used to traverse elements in a container. They act as a generalization of pointers and provide a way to access elements in a container sequentially.*

### STL Containers

#### Sequence Containers:

*Examples include vector, list, deque.*

#### Associative Containers:

*Examples include set, map, multiset, multimap.*

#### Container Adapters:

#### *Examples include stack, queue, priority\_queue.*

#### Vector: std::vector

*Vectors are the 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 the array may need to be extended. 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. (why?)*

// C++ program to illustrate the element access 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;

}

// C++ program 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;

}

Example 3

// 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();

// resizes the vector size to 4

g1.resize(4);

// prints the vector size after resize()

cout << "\nSize : " << g1.size();

// checks if the vector is empty or not

if (g1.empty() == false)

cout << "\nVector is not empty";

else

cout << "\nVector is empty";

// Shrinks the vector

g1.shrink\_to\_fit();

cout << "\nVector elements are: ";

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

cout << \*it << " ";

return 0;

}

// C++ program to illustrate the

// Modifiers in vector

#include <bits/stdc++.h>

#include <vector>

using namespace std;

int main()

{

// Assign vector

vector<int> v;

// fill the vector with 10 five times

v.assign(5, 10);

// inserts 15 to the last position

v.push\_back(15);

int n = v.size();

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

// removes last element

v.pop\_back();

// inserts 5 at the beginning

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

// removes the first element

v.erase(v.begin());

// 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();

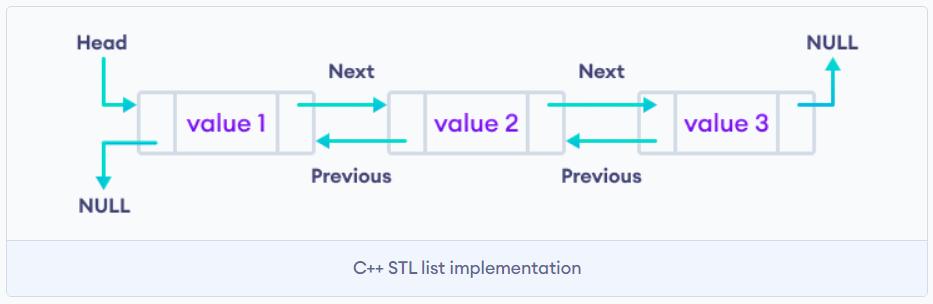
cout << "\nVector size after clear(): " << v.size();

}

### List: std::list

*C++ List is a STL container that stores elements randomly in unrelated locations. To maintain sequential ordering, every list element includes two links:*

* *One that points to the previous element*
* *Another that points to the next element*



In C++, the STL list implements the doubly-linked list data structure. As a result, we can iterate both forward and backward.

**Syntax**:

#include<list>

std::list<Type> list\_name = {value1, value2, ...};

Exercise: Check the element address within same list

#include <iostream>

#include <list>

int main() {

// Creating a list of integers

std::list<int> myList;

// Adding elements to the list

myList.push\_back(10);

myList.push\_back(20);

myList.push\_back(30);

myList.push\_front(5);

// Iterating through the list using an iterator

std::cout << "List elements: ";

for (std::list<int>::iterator it = myList.begin(); it != myList.end(); ++it) {

std::cout << \*it << " ";

}

// Accessing elements using range-based for loop (C++11 and later)

std::cout << "\nList elements (using range-based for loop): ";

for (const auto &element : myList) {

std::cout << element << " ";

}

// Size of the list

std::cout << "\nSize of the list: " << myList.size() << std::endl;

// Removing elements from the list

myList.pop\_back();

myList.pop\_front();

// Checking if the list is empty

std::cout << "Is the list empty? " << (myList.empty() ? "Yes" : "No") << std::endl;

// Clearing all elements from the list

myList.clear();

// Checking the size after clearing

std::cout << "Size of the list after clearing: " << myList.size() << std::endl;

return 0;

}

#### Set: std::set

*In C++, std::set is a part of the Standard Template Library (STL) and is a container that stores unique, ordered elements. It is implemented as a sorted binary tree, typically a Red-Black Tree, which ensures that the elements are always sorted in ascending order. As a result, the insertion, deletion, and search operations have a time complexity of O(log n).*

***Key features***

* *Uniqueness: Each element in a set must be unique. If an attempt is made to insert a duplicate element, it will not be added to the set.*
* *Ordered: The elements in a set are always ordered. This allows for efficient searching, insertion, and deletion operations.*
* *Associative Container: std::set is an associative container, meaning that it is based on key-value pairs. However, in a set, the key and value are the same, representing the element itself.*
* *Dynamic Sizing: The size of a set can grow or shrink dynamically as elements are inserted or removed.*

#include <iostream>

#include <set>

int main() {

// Creating a set of integers

std::set<int> mySet;

// Inserting elements

mySet.insert(30);

mySet.insert(10);

mySet.insert(20);

mySet.insert(40);

mySet.insert(20); // Duplicate elements are ignored

// Iterating through the set

for (const auto &element : mySet) {

std::cout << element << " ";

}

// Output: 10 20 30 40

return 0;

}

Another example

#include <iostream>

#include <set>

int main() {

// Creating a set of integers

std::set<int> mySet;

// Iterating through the set using an iterator

std::cout << "Set elements: ";

for (std::set<int>::iterator it = mySet.begin(); it != mySet.end(); ++it) {

std::cout << \*it << " ";

}

// Accessing elements using range-based for loop (C++11 and later)

std::cout << "\nSet elements (using range-based for loop): ";

for (const auto &element : mySet) {

std::cout << element << " ";

}

// Checking if an element is present in the set

int searchElement = 20;

std::cout << "\nIs " << searchElement << " present in the set? " << (mySet.count(searchElement) ? "Yes" : "No") << std::endl;

// Erasing an element from the set

mySet.erase(20);

// Size of the set

std::cout << "Size of the set: " << mySet.size() << std::endl;

// Clearing all elements from the set

mySet.clear();

// Checking the size after clearing

std::cout << "Size of the set after clearing: " << mySet.size() << std::endl;

return 0;

}

## Map: std::map

*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 the same key values. std::map is the class template for map containers and it is defined inside the <map> header file.*

// C++ program to illustrate the begin and end iterator

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main()

{

// Create a map of strings to integers

map<string, int> mp;

// Insert some values into the map

mp["one"] = 1;

mp["two"] = 2;

mp["three"] = 3;

// Print the size of the map

cout << "Size of map: " << map.size() << endl;

// Get an iterator pointing to the first element in the map

map<string, int>::iterator it = mp.begin();

// Iterate through the map and print the elements

while (it != mp.end()) {

cout << "Key: " << it->first

<< ", Value: " << it->second << endl;

++it;

}

return 0;

}

// CPP Program to demonstrate the implementation in Map

#include <iostream>

#include <iterator>

#include <map>

using namespace std;

int main()

{

// empty map container

map<int, int> gquiz1;

// 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));

// assigning the elements from gquiz1 to gquiz2

map<int, int> gquiz2(gquiz1.begin(), gquiz1.end());

// print all elements of the map gquiz2

map<int, int>::iterator itr;

for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {

cout << '\t' << itr->first << '\t' << itr->second

<< '\n';

}

// remove all elements up to element with key=3 in gquiz2

gquiz2.erase(gquiz2.begin(), gquiz2.find(3));

// remove all elements with key = 4

int num;

num = gquiz2.erase(4);

return 0;

}

// C++ program to demonstrate functionality of unordered\_map

#include <iostream>

#include <unordered\_map>

using namespace std;

// Driver code

int main()

{

// Declaring umap to be of

// <string, int> type key

// will be of STRING type

// and mapped VALUE will

// be of int type

unordered\_map<string, int> umap;

// inserting values by using [] operator

umap["GeeksforGeeks"] = 10;

umap["Practice"] = 20;

umap["Contribute"] = 30;

// Traversing an unordered map

for (auto x : umap)

cout << x.first << " " <<

x.second << endl;

}

#### std:unordered\_map vs std::map

*When it comes to efficiency, there is a huge difference between maps and unordered maps.*

| map | unordered\_map

---------------------------------------------------------

Ordering | increasing order | no ordering

| of keys(by default) |

Implementation| Self balancing BST | Hash Table

search time | log(n) | O(1) -> Average

| | O(n) -> Worst Case

Insertion time | log(n) + Rebalance | Same as search

Deletion time | log(n) + Rebalance | Same as search

***Use std::map when***

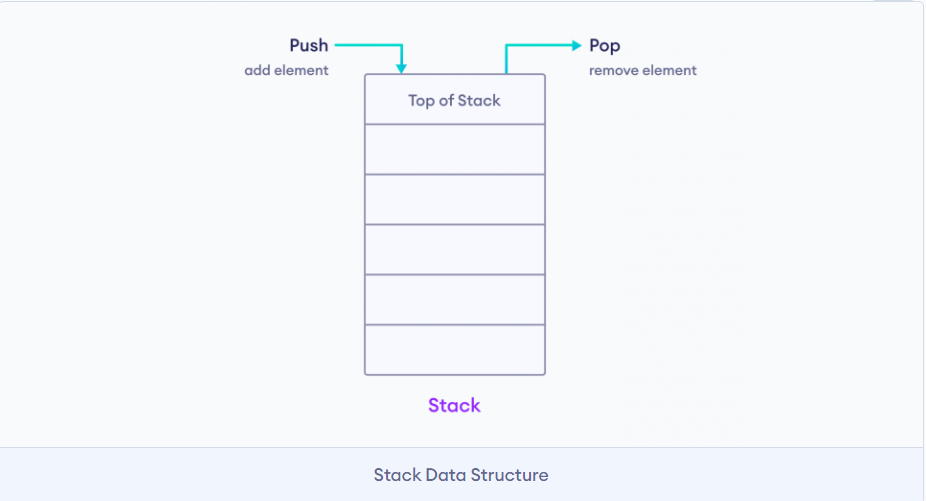
* *You need ordered data.*
* *You would have to print/access the data (in sorted order).*
* *You need predecessor/successor of elements.*

***Use std::unordered\_map when***

* *You need to keep count of some data (Example – strings) and no ordering is required.*
* *You need single element access i.e. no traversal.*

## Stack: std::stack

*The STL stack provides the functionality of a stack data structure in C++.**The stack data structure follows the LIFO (Last In First Out) principle. That is, the element added last will be removed first.*



#include <iostream>

#include <stack>

using namespace std;

int main() {

// create a stack of strings

stack<string> languages;

// add element to the Stack

languages.push("C++");

languages.push("Java");

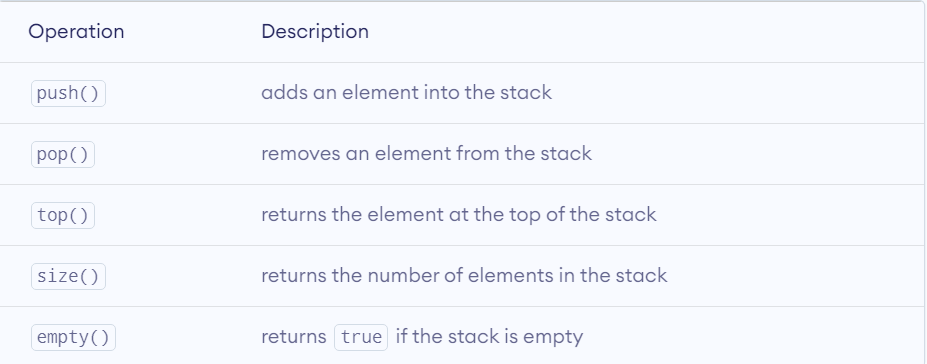
languages.push("Python");

// print top element

cout << languages.top();

return 0;

}



## Queue: std::queue

*Queues are a type of container adaptors that operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.*

| **Method** | **Definition** |
| --- | --- |
| [queue::empty()](https://www.geeksforgeeks.org/queueempty-queuesize-c-stl/) | Returns whether the queue is empty. It return true if the queue is empty otherwise returns false. |
| [queue::size()](https://www.geeksforgeeks.org/queueempty-queuesize-c-stl/) | Returns the size of the queue. |
| [queue::swap()](https://www.geeksforgeeks.org/queue-swap-cpp-stl/) | Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ. |
| [queue::emplace()](https://www.geeksforgeeks.org/queueemplace-c-stl/) | Insert a new element into the queue container, the new element is added to the end of the queue. |
| [queue::front()](https://www.geeksforgeeks.org/queuefront-queueback-c-stl/) | Returns a reference to the first element of the queue. |
| [queue::back()](https://www.geeksforgeeks.org/queuefront-queueback-c-stl/) | Returns a reference to the last element of the queue. |
| [queue::push(g)](https://www.geeksforgeeks.org/queue-push-and-queue-pop-in-cpp-stl/) | Adds the element ‘g’ at the end of the queue. |
| [queue::pop()](https://www.geeksforgeeks.org/queue-push-and-queue-pop-in-cpp-stl/) | Deletes the first element of the queue. |

#include <iostream>

#include <queue>

using namespace std;

void print\_queue(queue<int> q)

{

queue<int> temp = q;

while (!temp.empty()) {

cout << temp.front()<<" ";

temp.pop();

}

cout << '\n';

}

int main()

{

queue<int> q1;

q1.push(1);

q1.push(2);

q1.push(3);

cout << "The first queue is : ";

print\_queue(q1);

queue<int> q2;

q2.push(4);

q2.push(5);

q2.push(6);

cout << "The second queue is : ";

print\_queue(q2);

q1.swap(q2);

cout << "After swapping, the first queue is : ";

print\_queue(q1);

cout << "After swapping the second queue is : ";

print\_queue(q2);

cout<<q1.empty(); //returns false since q1 is not empty

return 0;

}

*print\_queue function:*

*Time complexity: O(n), where n is the number of elements in the queue.*

*Space complexity: O(n), where n is the number of elements in the queue.*

*q1.push(1), q1.push(2), q1.push(3), q2.push(4), q2.push(5), q2.push(6):*

*Time complexity: O(1) for each push operation.*

*Space complexity: O(n), where n is the total number of elements in both queues.*

*q1.swap(q2):*

*Time complexity: O(1) for each swap operation.*

*Space complexity: O(1), as this operation only swaps the internal pointers of the two queues.*

*q1.empty():*

*Time complexity: O(1), as this operation simply checks if the queue is empty.*

*Space complexity: O(1), as no extra space is used for this operation.*

*Overall, the time and space complexities of this code are reasonable and efficient for typical use cases.*

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

* *sort(first\_iterator, last\_iterator) – To sort the given vector.*
* *reverse(first\_iterator, last\_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)*
* *\*max\_element (first\_iterator, last\_iterator) – To find the maximum element of a vector.*
* *\*min\_element (first\_iterator, last\_iterator) – To find the minimum element of a vector.*
* *accumulate(first\_iterator, last\_iterator, initial value of sum) – Does the summation of vector elements*
* *count(first\_iterator, last\_iterator,x) – To count the occurrences of x in vector.*
* *find(first\_iterator, last\_iterator, x) – Returns an iterator to the first occurrence of x in vector and points to last address of vector ((name\_of\_vector).end()) if element is not present in vector.*

### Sorting: std::sort()

*It generally takes two parameters, the first one being the point of the array/vector from where the sorting needs to begin and the second parameter being the length up to which we want the array/vector to get sorted. The third parameter is optional and can be used in cases such as if we want to sort the elements lexicographically. By default, the sort() function sorts the elements in ascending order.*

// C++ program to demonstrate default behaviour of

// sort() in STL.

#include <bits/stdc++.h>

using namespace std;

int main()

{

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

int n = sizeof(arr) / sizeof(arr[0]);

/\*Here we take two parameters, the beginning of the

array and the length n upto which we want the array to

be sorted\*/

sort(arr, arr + n);

cout << "\nArray after sorting using "

"default sort is : \n";

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

cout << arr[i] << " ";

return 0;

}

**Output**

Array after sorting using default sort is :

0 1 2 3 4 5 6 7 8 9

**Time Complexity:**O(N log N)  
**Auxiliary Space:**O(1)

#### What about sorting in descending order?

*std::sort() takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass the “greater()” function to sort in descending order. This function does a comparison in a way that puts greater elements before.*

// C++ program to demonstrate descending order sort using

// greater<>().

#include <bits/stdc++.h>

using namespace std;

int main()

{

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

int n = sizeof(arr) / sizeof(arr[0]);

sort(arr, arr + n, greater<int>());

cout << "Array after sorting : \n";

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

cout << arr[i] << " ";

return 0;

}

*std::greater() defintion*

template <class T> struct greater {

bool operator() (const T& x, const T& y) const {return x>y;}

typedef T first\_argument\_type;

typedef T second\_argument\_type;

typedef bool result\_type;

};

## Find max element: std::max\_element()

*We have std::max to find maximum of 2 or more elements, but what if we want to find the largest element in an array or vector or list or in a sub-section. To serve this purpose, we have std::max\_element in C++. std::max\_element is defined inside the header file and it returns an iterator pointing to the element with the largest value in the range [first, last). std::max\_element can be used in two ways. The comparisons can be performed either usingoperator < (first version), or using a pre-defined function (second version). If more than one element satisfies the condition of being the largest, the iterator returned points to the first of such elements.*

1. *For comparing elements using “<“:*

// C++ program to demonstrate the use of std::max\_element

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

 int v[] = { 'a', 'c', 'k', 'd', 'e', 'f', 'h' };

 // Finding the maximum value between the first and the fourth element

 int\* i1;

 i1 = std::max\_element(v, v + 4);

 cout << char(\*i1) << "\n";

 return 0;

}

**Output**

k

**Time Complexity:**O(n)

1. *For comparison based on a pre-defined function:*

Template ForwardIterator max\_element (ForwardIterator first, ForwardIterator last, Compare comp);

// C++ program to demonstrate the use of std::max\_element

#include <iostream>

#include <algorithm>

using namespace std;

// Defining the BinaryFunction

bool comp(int a, int b)

{

return (a < b);

}

int main()

{

int v[] = { 9, 4, 7, 2, 5, 10, 11, 12, 1, 3, 6 };

// Finding the maximum value between the third and the ninth element

int\* i1;

i1 = std::max\_element(v + 2, v + 9, comp);

cout << \*i1 << "\n";

return 0;

}

**Output:**

12

**Time complexity:**O(n)

## Find elemenet in container: std::find()

*std::find is a function defined inside <algorithm> header file that finds the element in the given range. It returns an iterator to the first occurrence of the specified element in the given sequence. If the element is not found, an iterator to the end is returned.*

***Syntax:***

input\_iterator std::find(input\_iterator first, input\_iterator last, const T& value);

*Parameters:*

*first: iterator to the initial position in the sequence.*

*last: iterator to position just after the final position in the sequence. (Note that vector.end() points to the next position to the last element of the sequence and not to the last position of the sequence).*

*value: value to be searched.*

*Return Value :*

*If the value is found in the sequence, the iterator to its position is returned.*

*If the value is not found, the iterator to the last position is returned.*

// C++ program to Demonstrate std::find for vectors

#include <bits/stdc++.h>

// Driver code

int main()

{

std::vector<int> vec{10, 20, 30, 40};

// Iterator used to store the position of searched element

std::vector<int>::iterator it;

// Print Original Vector

std::cout << "Original vector :";

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

std::cout << " " << vec[i];

std::cout << "\n";

// Element to be searched

int ser = 30;

// std::find function call

it = std::find(vec.begin(), vec.end(), ser);

if (it != vec.end()) {

std::cout << "Element " << ser << " found at position : ";

std::cout << it - vec.begin() << " (counting from zero) \n";

}

else

std::cout << "Element not found.\n\n";

return 0;

}

**Output**

Original vector: 10 20 30 40

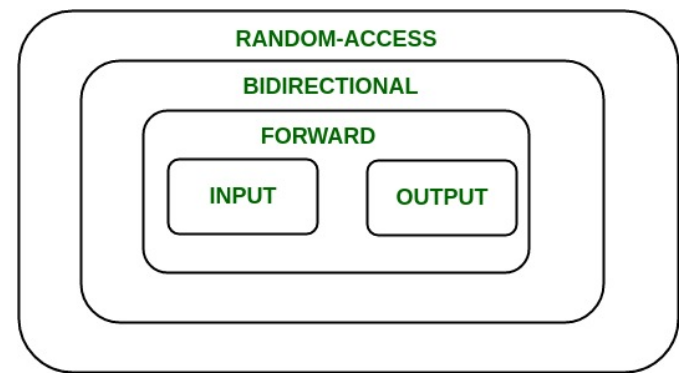
Element 30 found at position: 2 (counting from zero)

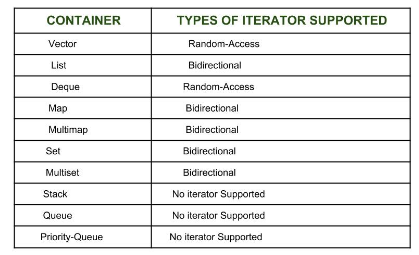
**Time Complexity:**O(n)  
**Auxiliary Space:**O(1)

# **STL Iterator**

### Introduction to Iterators in C++

*An iterator is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of an iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers. Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.*





#### Types of iterators

*Based upon the functionality of the iterators, they can be classified into five major categories:*

*Input Iterators:*

*They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.*

*Output Iterators:*

*Just like input iterators, they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.*

*Forward Iterator:*

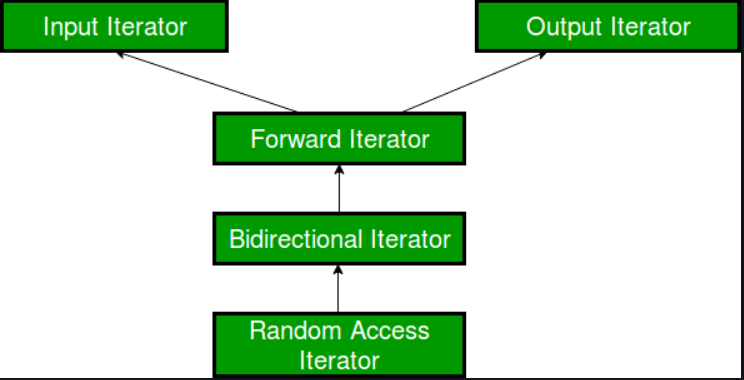
*They are higher in the hierarchy than input and output iterators, and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.*

*Bidirectional Iterators:*

*They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions. This is why their name is bidirectional.*

*Random-Access Iterators:*

*They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.*



#### Input Iterator

*Input iterators are considered to be the weakest as well as the simplest among all the iterators available, based upon their functionality and what can be achieved using them. They are the iterators that can be used in sequential input operations, where each value pointed by the iterator is read-only once and then the iterator is incremented.*

1. ***Usability****: Input iterators can be used only with single-pass algorithms, i.e., algorithms in which we can go to all the locations in the range at most once, like when we have to search or find any element in the range, we go through the locations at most once.*
2. ***Equality / Inequality Comparison****: An input iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not. So, the following two expressions are valid if A and B are input iterators:*

*A == B // Checking for equality*

*A != B // Checking for inequality*

1. ***Dereferencing****: An input iterator can be dereferenced, using the operator \* and -> as an rvalue to obtain the value stored at the position being pointed to by the iterator. So, the following two expressions are valid if A is an input iterator:*

*\*A // Dereferencing using \**

*A -> m // Accessing a member element m*

1. ***Incrementable****: An input iterator can be incremented, so that it refers to the next element in the sequence, using operator ++().*

*A++ // Using post increment operator*

*++A // Using pre increment operator*

1. ***Swappable****: The value pointed to by these iterators can be exchanged or swapped.*

Example 1: incrementing and dereferencing

// C++ program to demonstrate iterators

#include <iostream>

#include <vector>

using namespace std;

int main()

{

// Declaring a vector

vector<int> v = { 1, 2, 3 };

// Declaring an iterator

vector<int>::iterator i;

int j;

cout << "Without iterators = ";

// Accessing the elements without using iterators

for (j = 0; j < 3; ++j)

{

cout << v[j] << " ";

}

cout << "\nWith iterators = ";

// Accessing the elements using iterators

for (i = v.begin(); i != v.end(); ++i)

{

cout << \*i << " ";

}

return 0;

}

Example 2: Dynamic processing of the container: Iterators provide us the ability to dynamically add or remove elements from the container as and when we want with ease.

// C++ program to demonstrate iterators

#include <iostream>

#include <vector>

using namespace std;

int main()

{

// Declaring a vector

vector<int> v = { 1, 2, 3 };

// Declaring an iterator

vector<int>::iterator i;

// Inserting element using iterators

for (i = v.begin(); i != v.end(); ++i) {

if (i == v.begin()) {

i = v.insert(i, 5);

// inserting 5 at the beginning of v

}

}

// v contains 5 1 2 3

// Deleting a element using iterators

for (i = v.begin(); i != v.end(); ++i) {

if (i == v.begin() + 1) {

i = v.erase(i);

// i now points to the element after the

// deleted element

}

}

// v contains 5 2 3

return 0;

}

***Limitations***

1. *Only accessing, no assigning: One of the biggest deficiency is that we cannot assign any value to the location pointed by this iterator, it can only be used to access elements and not assign elements.*

*\*i1 = 7;*

*So, this is not allowed in the input iterator. However, if you try this for the above code, it will work, because vectors return iterators higher in the hierarchy than input iterators.That big deficiency is the reason why many algorithms like std::copy, which requires copying a range into another container cannot use input iterator for the resultant container, because we can’t assign values to it with such iterators and instead make use of output iterators.*

1. *Cannot be decremented: Just like we can use operator ++() with input iterators for incrementing them, we cannot decrement them. If A is an input iterator, then*

*A-- // Not allowed with input iterators*

1. *Use in multi-pass algorithms: Since it is unidirectional and can only move forward, therefore, such iterators cannot be used in multi-pass algorithms, in which we need to process the container multiple times.*

1. *Relational Operators: Although, input iterators can be used with equality operator (==), but it can not be used with other relational operators like <=. If A and B are input iterators, then*

*A == B // Allowed*

*A <= B // Not Allowed*

1. *Arithmetic Operators: Similar to relational operators, they also can’t be used with arithmetic operators like +, – and so on. This means that input operators can only move in one direction that too forward and that too sequentially. If A and B are input iterators, then*

*A + 1 // Not allowed*

*B - 2 // Not allowed*

#### Output Iterator

*After going through the template definition of various STL algorithms like std::copy, std::move, std::transform, you must have found their template definition consisting of objects of type Output Iterator. Output iterators are considered to be the exact opposite of input iterators, as they perform the opposite function of input iterators. They can be assigned values in a sequence, but cannot be used to access values, unlike input iterators which do the reverse of accessing values and cannot be assigned values. So, we can say that input and output iterators are complementary to each other.*

1. ***Usability****: Just like input iterators, Output iterators can be used only with single-pass algorithms, i.e., algorithms in which we can go to all the locations in the range at most once, such that these locations can be dereferenced or assigned value only once.*
2. ***Equality / Inequality Comparison****: Unlike input iterators, output iterators cannot be compared for equality with another iterator. So, the following two expressions are invalid if A and B are output iterators:*

*A == B // Invalid - Checking for equality*

*A != B // Invalid - Checking for inequality*

1. ***Dereferencing****: An input iterator can be dereferenced as an rvalue, using operator \* and ->, whereas an output iterator can be dereferenced as an lvalue to provide the location to store the value. So, the following two expressions are valid if A is an output iterator:*

*\*A = 1 // Dereferencing using \**

*A -> m = 7 // Assigning a member element m*

1. ***Incrementable****: An output iterator can be incremented, so that it refers to the next element in sequence, using operator ++(). So, the following two expressions are valid if A is an output iterator:*

*A++ // Using post increment operator*

*++A // Using pre increment operator*

1. ***Swappable****: The value pointed to by these iterators can be exchanged or swapped.*

#include <iostream>

#include <iterator>

#include <vector>

int main() {

// Creating a vector of integers

std::vector<int> numbers = {1, 2, 3, 4, 5};

// Using std::ostream\_iterator as an output iterator

std::ostream\_iterator<int> outputIterator(std::cout, " ");

// Writing elements from the vector to the standard output

for (const auto& element : numbers) {

\*outputIterator = element; // Writing to the output iterator

++outputIterator; // Advancing the iterator

}

std::cout << std::endl;

return 0;

}

1. *A std::vector of integers is created.*
2. *An std::ostream\_iterator<int> is created, specifying std::cout as the output stream and a space as the delimiter.*
3. *A loop is used to iterate over the elements of the vector, and each element is written to the output iterator using the assignment \*outputIterator = element;.*
4. *The output iterator is then advanced to the next position using ++outputIterator;*

*Example of std::move()*

*std::move() is a utility function in C++ that indicates that the object should be "moved from" rather than copied from. It is often used in the context of move semantics, which allows for more efficient transfers of resources (like memory ownership) between objects*

***Template OutputIterator move (InputIterator first, InputIterator last, OutputIterator result)****;*

***Parameters*** *:*

*the first, last Input iterators to the initial and final positions in a sequence to be moved. The range used is [first,last], which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.*

*Result**output iterator to the initial position in the destination sequence. This shall not point to any element in the range [first,last].*

***Return type*** *:*

*An iterator to the end of the destination range where elements have been moved.*

// CPP program to illustrate

// std::move and std::move\_backward

// STL library functions

#include<bits/stdc++.h>

// Driver code

int main()

{

std :: vector <int> vec1 {1, 2, 3, 4, 5};

std :: vector <int> vec2 {7, 7, 7, 7, 7};

// Print elements

std :: cout << "Vector1 contains :";

for(int i = 0; i < vec1.size(); i++)

std :: cout << " " << vec1[i];

std :: cout << "\n";

// Print elements

std :: cout << "Vector2 contains :";

for(unsigned int i = 0; i < vec2.size(); i++)

std :: cout << " " << vec2[i];

std :: cout << "\n\n";

// std :: move function

// move first 4 element from vec1 to starting position of vec2

std :: move (vec1.begin(), vec1.begin() + 4, vec2.begin() + 1);

// Print elements

std :: cout << "Vector2 contains after std::move function:";

for(unsigned int i = 0; i < vec2.size(); i++)

std :: cout << " " << vec2[i];

std :: cout << "\n";

return 0;

}

## Forward Iterators in C++

*Forward iterators are considered to be the combination of input as well as output iterators. It provides support to the functionality of both of them. It permits values to be both accessed and modified.*

1. *Usability: Performing operations on a forward iterator that is dereferenceable never makes its iterator value non-dereferenceable, as a result this enables algorithms that use this category of iterators to use multiple copies of an iterator to pass more than once by the same iterator values. So, it can be used in multi-pass algorithms.*
2. *Equality / Inequality Comparison: A forward iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not. So, the following two expressions are valid if A and B are forward iterators:*

*A == B // Checking for equality*

*A != B // Checking for inequality*

1. *Dereferencing: Because an input iterator can be dereferenced, using the operator \* and -> as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators can be used for both the purposes.*
2. *Incrementable: A forward iterator can be incremented, so that it refers to the next element in sequence, using operator ++(). Note: The fact that we can use forward iterators with increment operator doesn’t mean that operator – -() can also be used with them. Remember, that forward iterators are unidirectional and can only move in the forward direction. So, the following two expressions are valid if A is a forward iterator:*

*A++ // Using post increment operator*

*++A // Using pre increment operator*

1. *Swappable: The value pointed to by these iterators can be exchanged or swapped.*

*Limitations*

*After studying the salient features, one must also know its deficiencies as well although there are not as many as there are in input or output iterators as it is higher in the hierarchy.*

1. *Can not be decremented: Just like we can use operator ++() with forward iterators for incrementing them, we cannot decrement them. Although it is higher in the hierarchy than input and output iterators, still it can’t overcome this deficiency. That is why, its name is forward, which shows that it can move only in forward direction.*

*If A is a forward iterator, then*

*A-- // Not allowed with forward iterators*

1. *Relational Operators: Although, forward iterators can be used with equality operator (==), it can not be used with other relational operators like =.*

*If A and B are forward iterators, then*

*A == B // Allowed*

*A <= B // Not Allowed*

1. *Arithmetic Operators: Similar to relational operators, they also can’t be used with arithmetic operators like +, – and so on. This means that forward operators can only move in one direction that too forward and that too sequentially.*

*If A and B are forward iterators, then*

*A + 1 // Not allowed*

*B - 2 // Not allowed*

1. *Use of offset dereference operator ([ ]): Forward iterators do not support offset dereference operator ([ ]), which is used for random-access.*

*If A is a forward iterator, then*

*A[3] // Not allowed*

*Std::lower\_bound() example of forward iterator*

*The lower\_bound() method in C++ is used to return an iterator pointing to the first element in the range [first, last) which has a value not less than val. This means that the function returns an iterator pointing to the next smallest number just greater than or equal to that number. If there are multiple values that are equal to val, lower\_bound() returns the iterator of the first such value. The elements in the range shall already be sorted or at least partitioned with respect to val.*

***Templates:***

***Syntax 1:***

***ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, const T& val);***

***Syntax 2:***

***ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);***

***Parameters****:*

*The above methods accept the following parameters.*

*first, last: The range used is [first, last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.*

*val: Value of the lower bound to be searched for in the range.*

*comp: Binary function that accepts two arguments (the first of the type pointed by ForwardIterator, and the second, always val), and returns a value convertible to bool. The function shall not modify any of its arguments. This can either be a function pointer or a function object.*

***Return Value:***

*An iterator to the lower bound of val in the range. If all the elements in the range compare less than val, the function returns last. If all the elements in the range are larger than val, the function returns a pointer to the first element.*

// CPP program to illustrate

// std :: lower\_bound

#include <bits/stdc++.h>

// Driver code

int main()

{

// Input vector

std::vector<int> v{ 10, 20, 30, 30, 30, 40, 50 };

// Print vector

std::cout << "Vector contains :";

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

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

std::cout << "\n";

std::vector<int>::iterator low1, low2, low3;

// std :: lower\_bound

low1 = std::lower\_bound(v.begin(), v.end(), 30);

low2 = std::lower\_bound(v.begin(), v.end(), 35);

low3 = std::lower\_bound(v.begin(), v.end(), 55);

// Printing the lower bounds

std::cout

<< "\nlower\_bound for element 30 at position : "

<< (low1 - v.begin());

std::cout

<< "\nlower\_bound for element 35 at position : "

<< (low2 - v.begin());

std::cout

<< "\nlower\_bound for element 55 at position : "

<< (low3 - v.begin());

return 0;

}

### Quiz

Question 1:

Which STL container is implemented as a doubly-linked list?

a) std::vector

b) std::list

c) std::deque

d) std::set

Question 2:

What is the purpose of the std::algorithm library in C++ STL?

a) String manipulation

b) Input and output operations

c) Container algorithms

d) Memory management

Question 3:

What is the difference between std::map and std::unordered\_map?

a) std::map is sorted; std::unordered\_map is not

b) std::map uses a hash function; std::unordered\_map uses a comparator

c) std::map allows duplicate keys; std::unordered\_map does not

d) There is no difference; they are the same container

Question 4:

Which STL algorithm is used to sort elements in a range?

a) std::sort

b) std::find

c) std::search

d) std::merge

Question 5:

What is the purpose of the std::pair template class?

a) Representing a sequence of elements

b) Associating two values together

c) Managing dynamic memory allocation

d) Sorting elements in a container

Question 6:

Which STL container provides constant time complexity for insertion and deletion of elements at the beginning and end?

a) std::vector

b) std::list

c) std::deque

d) std::queue

Question 7:

What is the purpose of the std::unique algorithm?

a) Removing consecutive duplicate elements in a range

b) Finding the first occurrence of a value in a range

c) Merging two sorted ranges

d) Reversing the order of elements in a range

Question 8:

Which STL container is typically implemented as a binary heap?

a) std::vector

b) std::list

c) std::queue

d) std::stack