C++ STL

Vector : 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.

Iterators:

1. begin() – Returns an iterator pointing to the first element in the vector
2. end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector

Capacity:

1. size() – Returns the number of elements in the vector.
2. max_size() – Returns the maximum number of elements that the vector can hold.
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. resize() – Resizes the container so that it contains 'g' elements.
5. empty() – Returns whether the container is empty.
6. shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. reserve() – Requests that the vector capacity be at least enough to contain n elements.

Element access:

1. at(g) – Returns a reference to the element at position 'g' in the vector
2. front() – Returns a reference to the first element in the vector
3. back() – Returns a reference to the last element in the vector
4. data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

Modifiers:

1. assign() – It assigns new value to the vector elements by replacing old ones
2. push_back() – It push the elements into a vector from the back
3. pop_back() – 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() – It is used to remove elements from a container from the specified position or range.
6. swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
7. clear() – It is used to remove all the elements of the vector container

Queue: 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.

Functions :

1. empty() – Returns whether the queue is empty.
2. size() – Returns the size of the queue.
3. queue::swap() in C++ STL: Exchange the contents of two queues but the queues must be of same type, although sizes may differ.
4. queue::emplace() in C++ STL: Insert a new element into the queue container, the new element is added to the end of the queue.
5. queue::front() and queue::back() in C++ STL– front() function returns a reference to the first element of the queue. back() function returns a reference to the last element of the queue.
6. push(g) and pop() – push() function adds the element 'g' at the end of the queue. pop() function deletes the first element of the queue.

Stack in C++ 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.

- empty() – Returns whether the stack is empty – Time Complexity : O(1)
- size() – Returns the size of the stack – Time Complexity : O(1)
- top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)
- push(g) – Adds the element 'g' at the top of the stack – Time Complexity : O(1)
- pop() – Deletes the top most element of the stack – Time Complexity : O(1)

List (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 lis**t, we use forward list.

Functions used with List:

- front() – Returns the value of the first element in the list.
- back() – Returns the value of the last element in the list .
- push_front(g) – Adds a new element 'g' at the beginning of the list .
- push_back(g) – Adds a new element 'g' at the end of the list.
- pop_front() – Removes the first element of the list, and reduces size of the list by 1.
- pop_back() – Removes the last element of the list, and reduces size of the list by 1
- list::begin() and list::end() in C++ STL– begin() function returns an iterator pointing to the first element of the list
- end()– end() function returns an iterator pointing to the theoretical last element which follows the last element.
- empty() – Returns whether the list is empty(1) or not(0).
- insert() – Inserts new elements in the list before the element at a specified position.
- erase() – Removes a single element or a range of elements from the list.
- assign() – Assigns new elements to list by replacing current elements and resizes the list.
- remove() – Removes all the elements from the list, which are equal to given element.
- reverse() – Reverses the list.

- sort() – Sorts the list in increasing order.
- merge() – Merges two sorted lists into one

Set in C++ (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.4

- begin() – Returns an iterator to the first element in the set.
- end() – Returns an iterator to the theoretical element that follows last element in the set.
- size() – Returns the number of elements in the set.
- max_size() – Returns the maximum number of elements that the set can hold.
- empty() – Returns whether the set is empty.
- begin() – Returns an iterator to the first element in the set.
- end() – Returns an iterator to the theoretical element that follows last element in the set.
- size() – Returns the number of elements in the set.
- max_size() – Returns the maximum number of elements that the set can hold.
- empty() – Returns whether the set is empty.
- clear() – Removes all the elements from the set.
- count(const g) – Returns 1 or 0 based on the element 'g' is present in the set or not.
- find(const g) – Returns an iterator to the element 'g' in the set if found, else returns the iterator to end.
- erase(const g)– Removes the value 'g' from the set.

Map in C++ (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.

Some basic functions associated with Map:

- begin() – Returns an iterator to the first element in the map
- end() – Returns an iterator to the theoretical element that follows last element in the map
- size() – Returns the number of elements in the map
- max_size() – Returns the maximum number of elements that the map can hold
- empty() – Returns whether the map is empty
- pair insert(keyvalue, mapvalue) – Adds a new element to the map
- erase(iterator position) – Removes the element at the position pointed by the iterator
- erase(const g)– Removes the key value 'g' from the map
- clear() – Removes all the elements from the map
- map insert() in C++ STL– Insert elements with a particular key in the map container. .
- map count() function in C++ STL– Returns the number of matches to element with key value 'g' in the map.
- map equal_range() in C++ STL– Returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to k.
- map erase() function in C++ STL– Used to erase element from the container.
- map find() function in C++ STL– Returns an iterator to the element with key value 'g' in the map if found, else returns the iterator to end.
- map emplace() in C++ STL– Inserts the key and its element in the map container.

- map::size() in C++ STL– Returns the number of elements in the map.
- map::empty() in C++ STL– Returns whether the map is empty.
- map::begin() and end() in C++ STL– begin() returns an iterator to the first element in the map. end() returns an iterator to the theoretical element that follows last element in the map
- map::clear() in C++ STL– Removes all the elements from the map.
- map::at() and map::swap() in C++ STL– at() function is used to return the reference to the element associated with the key k. swap() function is used to exchange the contents of two maps but the maps must be of same type, although sizes may differ.

unordered_map in C++ STL:

unordered_map is an associated container that stores elements formed by combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.

Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of hash table that is why performance of data structure depends on hash function a lot but on an average the cost of search, insert and delete from hash table is O(1).

unordered_map vs map :
map (like set) is an ordered sequence of unique keys whereas in unordered_map key can be stored in any order, so unordered.
Map is implemented as balanced tree structure that is why it is possible to maintain an order between the elements (by specific tree traversal). Time complexity of map operations is O(Log n) while for unordered_set, it is O(1) on average.

Methods of unordered_map :

- at(): This function in C++ unordered_map returns the reference to the value with the element as key k.
- begin(): Returns an iterator pointing to the first element in the container in the unordered_map container
- end(): Returns an iterator pointing to the position past the last element in the container in the unordered_map container
- bucket(): Returns the bucket number where the element with the key k is located in the map.
- bucket_count: bucket_count is used to count the total no. of buckets in the unordered_map. No parameter is required to pass into this function.
- bucket_size: Returns number of elements in each bucket of the unordered_map.
- count(): Count the number of elements present in an unordered_map with a given key.
- equal_range: Return the bounds of a range that includes all the elements in the container with a key that compares equal to k.

Priority Queue in C++ (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 increasing order(hence we can see that each element of the queue has a priority{fixed order}).

- priority_queue::empty() in C++ STL– empty() function returns whether the queue is empty.
- priority_queue::size() in C++ STL– size() function returns the size of the queue.
- priority_queue::top() in C++ STL– Returns a reference to the top most element of the queue
- priority_queue::push() in C++ STL– push(g) function adds the element 'g' at the end of the queue.
- priority_queue::pop() in C++ STL– pop() function deletes the first element of the queue.
- priority_queue::swap() in C++ STL– This function is used to swap the contents of one priority queue with another priority queue of same type and size.
- priority_queue::emplace() in C++ STL – This function is used to insert a new element into the priority queue container, the new element is added to the top of the priority queue.
- priority_queue value_type in C++ STL– Represents the type of object stored as an element in a priority_queue. It acts as a synonym for the template parameter.

Binary Search in C++ (STL): 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.

- binary_search(a, a + 10, 2)

Algorithm Library | C++ Magicians STL Algorithm:

#include <algorithm>
#include <iostream>
#include <vector>
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(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
6. **count**(first_iterator, last_iterator,x) – To count the occurrences of x in vector.
7. **find**(first_iterator, last_iterator, x) – Points to last address of vector ((name_of_vector).end()) if element is not present in vector.
8. **binary_search**(first_iterator, last_iterator, x) – Tests whether x exists in sorted vector or not.

9. lower_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.
10. upper_bound(first_iterator, last_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.
11. **arr.erase(position to be deleted)** – This erases selected element in vector and shifts and resizes the vector elements accordingly.
12. **arr.erase(unique(arr.begin(),arr.end()),arr.end())** – This erases the duplicate occurrences in sorted vector in a single line.
13. next_permutation(first_iterator, last_iterator) – This modified the vector to its next permutation.
14. prev_permutation(first_iterator, last_iterator) – This modified the vector to its previous permutation.