STL

STL has 4 components:

- Algorithms
- Containers
- Functors
- Iterators

Algorithms:

- Sorting
- Searching
- Important STL Algorithms
- Useful Array algorithms
- Partition Operations
- valarray class

Sorting: sort(startaddress, endaddress)

Searching: binary_search(startaddress, endaddress, valuetofind)

Important STL Algorithms:

Non-Manipulating Algorithms

1. sort(first_iterator, last_iterator) – To sort the given vector.
2. sort(first_iterator, last_iterator, greater<int>()) – To sort the given container/vector in descending order
3. reverse(first_iterator, last_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)
4. *max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
5. *min_element (first_iterator, last_iterator) – To find the minimum element of a vector.
6. accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements
7. count(first_iterator, last_iterator,x) – To count the occurrences of x in vector.
8. 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.

9. binary_search(first_iterator, last_iterator, x) – Tests whether x exists in sorted vector or not.
10. 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’.
11. 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’.

Some Manipulating Algorithms

1. **arr.erase(position to be deleted)** – This erases selected element in vector and shifts and resizes the vector elements accordingly.
2. **arr.erase(unique(arr.begin(),arr.end()),arr.end())** – This erases the duplicate occurrences in sorted vector in a single line.
3. **next_permutation(first_iterator, last_iterator)** – This modified the vector to its next permutation.
4. **prev_permutation(first_iterator, last_iterator)** – This modified the vector to its previous permutation.
5. **distance(first_iterator,desired_position)** – It returns the distance of desired position from the first iterator.This function is very useful while finding the index.

Useful Array algorithms:

- 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.
- 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.
- none_of() This function returns true if none of elements satisfies the given condition else returns false.
- 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.
- iota() This function is used to assign continuous values to array. This function accepts 3 arguments, the array name, size, and the starting number.

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.

3. stable_partition(beg, end, condition) :- This function is used to partition the elements on basis of condition mentioned in its arguments in such a way that the relative order of the elements is preserved..
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.

5. partition_copy(beg, end, beg1, beg2, condition) :- This function copies the partitioned elements in the different 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.

Valarray Class:

1. apply() :- This function applies the manipulation given in its arguments to all the valarray elements at once and returns a new valarray with manipulated values.
2. sum() :- This function returns the summation of all the elements of valarrays at once.
3. min() :- This function returns the smallest element of valarray.
4. max() :- This function returns the largest element of valarray.
5. shift() :- This function returns the new valarray after shifting elements by the number mentioned in its argument. If the number is positive, left-shift is applied, if number is negative, right-shift is applied.
6. cshift() :- This function returns the new valarray after circularly shifting(rotating) elements by the number mentioned in its argument. If the number is positive, left-circular shift is applied, if number is negative, right-circular shift is applied.
7. swap() :- This function swaps one valarray with other.

Containers:

Containers or container classes store objects and data. There are in total seven standards "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 that can be accessed in a sequential manner.
    - vector
    - list
    - deque
    - arrays
    - forward_list(Introduced in C++11)
- Container Adaptors: provide a different interface for sequential containers.
    - queue
    - priority_queue
    - stack
- Associative Containers: implement sorted data structures that can be quickly searched (O(log n) complexity).
    - set
    - multiset
    - map
    - multimap
- Unordered Associative Containers: implement unordered data structures that can be quickly searched
    - unordered_set (Introduced in C++11)
    - unordered_multiset (Introduced in C++11)
    - unordered_map (Introduced in C++11)
    - unordered_multimap (Introduced in C++11)

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.

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
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. cbegin() – Returns a constant iterator pointing to the first element in the vector.
6. cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
8. crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

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(n) – Resizes the container so that it contains ‘n’ 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. reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
2. at(g) – Returns a reference to the element at position ‘g’ in the vector
3. front() – Returns a reference to the first element in the vector
4. back() – Returns a reference to the last element in the vector
5. 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
8. emplace() – It extends the container by inserting new element at position
9. emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

List:

Lists are sequence containers that allow non-contiguous memory allocation. As compared to the vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick (constant time). Normally, when we say a List, we talk about a doubly linked list. For implementing a singly linked list, we use a forward_list.

Functions	Definition
front()	Returns the value of the first element in the list.
back()	Returns the value of the last element in the list.
push_front(g)	Adds a new element ‘g’ at the beginning of the list.
push_back(g)	Adds a new element ‘g’ at the end of the list.
pop_front()	Removes the first element of the list, and reduces the size of the list by 1.
pop_back()	Removes the last element of the list, and reduces the size of the list by 1.
list::begin()	begin() function returns an iterator pointing to the first element of the list.
list::end()	end() function returns an iterator pointing to the theoretical last element which follows the last element.
list rbegin() and rend()	rbegin() returns a reverse iterator which points to the last element of the list. rend() returns a reverse iterator that points to the position before the beginning of the list.
list cbegin() and cend()	cbegin() returns a constant random access iterator which points to the beginning of the list. cend() returns a constant random access iterator which points to the end of the list.
list crbegin() and crend()	crbegin() returns a constant reverse iterator which points to the last element of the list i.e reversed beginning of the container. crend() returns a constant reverse iterator which points to the theoretical element preceding the first element in the list i.e. the reverse end of the list.

Functions	Definition
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 the list by replacing current elements and resizing the list.
remove()	Removes all the elements from the list, which are equal to a given element.
list::remove_if()	Used to remove all the values from the list that correspond true to the predicate or condition given as a parameter to the function.
reverse()	Reverses the list.
size()	Returns the number of elements in the list.
list resize()	Used to resize a list container.
sort()	Sorts the list in increasing order.
list max_size()	Returns the maximum number of elements a list container can hold.
list unique()	Removes all duplicate consecutive elements from the list.
list::emplace_front() and list::emplace_back()	emplace_front() function is used to insert a new element into the list container, and the new element is added to the beginning of the list. emplace_back() function is used to insert a new element into the list container, and the new element is added to the end of the list.

Functions	Definition
list::clear()	clear() function is used to remove all the elements of the list container, thus making it size 0.
list::operator=	This operator is used to assign new contents to the container by replacing the existing contents.
list::swap()	This function is used to swap the contents of one list with another list of the same type and size.
list splice()	Used to transfer elements from one list to another.
list merge()	Merges two sorted lists into one.
list emplace()	Extends the list by inserting a new element at a given position.

Deque:

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

Double Ended Queues are basically an implementation of the data structure double-ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

Method	Definition
deque::insert()	Inserts an element. And returns an iterator that points to the first of the newly inserted elements.

Method	Definition
deque::rbegin()	Returns a reverse iterator which points to the last element of the deque (i.e., its reverse beginning).
deque::rend()	Returns a reverse iterator which points to the position before the beginning of the deque (which is considered its reverse end).
deque::cbegin()	Returns a constant iterator pointing to the first element of the container, that is, the iterator cannot be used to modify, only traverse the deque.
deque::max_size()	Returns the maximum number of elements that a deque container can hold.
deque::assign()	Assign values to the same or different deque container.
deque::resize()	Function which changes the size of the deque.
deque::push_front()	It is used to push elements into a deque from the front.
deque::push_back()	This function is used to push elements into a deque from the back.
deque::pop_front() and deque::pop_back()	pop_front() function is used to pop or remove elements from a deque from the front. pop_back() function is used to pop or remove elements from a deque from the back.
deque::front() and deque::back()	front() function is used to reference the first element of the deque container. back() function is used to reference the last element of the deque container.
deque::clear() and deque::erase()	clear() function is used to remove all the elements of the deque container, thus making its size 0. erase() function is used to remove elements from a container from the specified position or range.

Method	Definition
deque::empty() and deque::size()	empty() function is used to check if the deque container is empty or not. size() function is used to return the size of the deque container or the number of elements in the deque container.
deque::operator= and deque::operator[]	operator= operator is used to assign new contents to the container by replacing the existing contents. operator[] operator is used to reference the element present at position given inside the operator.
deque::at() and deque::swap()	at() function is used reference the element present at the position given as the parameter to the function. swap() function is used to swap the contents of one deque with another deque of same type and size.
deque::begin() and deque::end()	begin() function is used to return an iterator pointing to the first element of the deque container. end() function is used to return an iterator pointing to the last element of the deque container.
deque::emplace_front() and deque::emplace_back()	emplace_front() function is used to insert a new element into the deque container. The new element is added to the beginning of the deque. emplace_back() function is used to insert a new element into the deque container. The new element is added to the end of the deque.

Arrays:

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.

4. front() :- This returns reference to the first element of array.
5. back() :- This returns reference to the last element of array.

6. size() :- It returns the number of elements in array. This is a property that C-style arrays lack.

7. 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.

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

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

Forward List:

Forward list in STL implements singly linked list. Introduced from C++11, forward list are more useful than other containers in insertion, removal, and moving operations (like sort) and allow time constant insertion and removal of elements.
It differs from the list by the fact that the forward list keeps track of the location of only the next element while the list keeps track of both the next and previous elements, thus increasing the storage space required to store each element. The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.
Forward List is preferred over the list when only forward traversal is required (same as the 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 the graph, etc.

Method	Definition
front()	This function is used to reference the first element of the forward list container.
begin()	This function is used to return an iterator pointing to the first element of the forward list container.
end()	This function is used to return an iterator pointing to the last element of the list container.
cbegin()	Returns a constant iterator pointing to the first element of the forward_list.
cend()	Returns a constant iterator pointing to the past-the-last element of the forward_list.
before_begin()	Returns an iterator that points to the position before the first element of the forward_list.

Method	Definition
cbefore_begin()	Returns a constant random access iterator which points to the position before the first element of the forward_list.
max_size()	Returns the maximum number of elements that can be held by forward_list.
resize()	Changes the size of forward_list.
unique()	Removes all consecutive duplicate elements from the forward_list. It uses a binary predicate for comparison.
reverse()	Reverses the order of the elements present in the forward_list.

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()	Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
queue::size()	Returns the size of the queue.
queue::swap()	Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ.
queue::emplace()	Insert a new element into the queue container, the new element is added to the end of the queue.
queue::front()	Returns a reference to the first element of the queue.

Method	Definition
queue::back()	Returns a reference to the last element of the queue.
queue::push(g)	Adds the element ‘g’ at the end of the queue.
queue::pop()	Deletes the first element of the queue.

Priority Queue:

A C++ priority queue is a type of container adapter, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue, and elements are in non-increasing or non-decreasing order (hence we can see that each element of the queue has a priority {fixed order}).

In C++ STL, the top element is always the greatest by default. We can also change it to the smallest element at the top. Priority queues are built on the top of the max heap and use an array or vector as an internal structure. In simple terms, STL Priority Queue is the implementation of Heap Data Structure.

Method	Definition
priority_queue::empty()	Returns whether the queue is empty.
priority_queue::size()	Returns the size of the queue.
priority_queue::top()	Returns a reference to the topmost element of the queue.
priority_queue::push()	Adds the element ‘g’ at the end of the queue.
priority_queue::pop()	Deletes the first element of the queue.
priority_queue::swap()	Used to swap the contents of two queues provided the queues must be of the same type, although sizes may differ.

Method	Definition
priority_queue::emplace()	Used to insert a new element into the priority queue container.
priority_queue value_type	Represents the type of object stored as an element in a priority_queue. It acts as a synonym for the template parameter.

Stack:

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 (top) and an element is removed from that end only. Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

If there is confusion in remembering the basic difference between stack and queue, then just have a real life example for this differentiation, for stack, stacking of books we can take the top book easily and for queue remember when you have to stand in queue front of ATM for taking out the cash, then first person near to ATM has the first chance to take out the money from ATM. So, queue is the FIFO (First In First Out) type working.

The functions associated with stack are:
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 most recent entered element of the stack – Time Complexity : O(1)

Set:

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending.

Function	Description
begin()	Returns an iterator to the first element in the set.
end()	Returns an iterator to the theoretical element that follows the last element in the set.

Function	Description
rbegin()	Returns a reverse iterator pointing to the last element in the container.
rend()	Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.
crbegin()	Returns a constant iterator pointing to the last element in the container.
crend()	Returns a constant iterator pointing to the position just before the first element in the container.
cbegin()	Returns a constant iterator pointing to the first element in the container.
cend()	Returns a constant iterator pointing to the position past the last element in the container.
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.
insert(const g)	Adds a new element ‘g’ to the set.
iterator insert (iterator position, const g)	Adds a new element ‘g’ at the position pointed by the iterator.
erase(iterator position)	Removes the element at the position pointed by the iterator.
erase(const g)	Removes the value ‘g’ from the set.

Function	Description
clear()	Removes all the elements from the set.
key_comp() / value_comp()	Returns the object that determines how the elements in the set are ordered ('<' by default).
find(const g)	Returns an iterator to the element 'g' in the set if found, else returns the iterator to the end.
count(const g)	Returns 1 or 0 based on whether the element 'g' is present in the set or not.
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.
upper_bound(const g)	Returns an iterator to the first element that will go after the element 'g' in the set.
equal_range()	The function returns an iterator of pairs. (key_comp). The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
emplace()	This function is used to insert a new element into the set container, only if the element to be inserted is unique and does not already exist in the set.
emplace_hint()	Returns an iterator pointing to the position where the insertion is done. If the element passed in the parameter already exists, then it returns an iterator pointing to the position where the existing element is.
swap()	This function is used to exchange the contents of two sets but the sets must be of the same type, although sizes may differ.

Function	Description
operator=	The '=' is an operator in C++ STL that copies (or moves) a set to another set and set::operator= is the corresponding operator function.
get_allocator()	Returns the copy of the allocator object associated with the set.

Multiset:

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

Function	Definition
begin()	Returns an iterator to the first element in the multiset.
end()	Returns an iterator to the theoretical element that follows the last element in the multiset.
size()	Returns the number of elements in the multiset.
max_size()	Returns the maximum number of elements that the multiset can hold.
empty()	Returns whether the multiset is empty.
pair insert(const g)	Adds a new element 'g' to the multiset.
iterator insert (iterator position,const g)	Adds a new element 'g' at the position pointed by the iterator.
erase(iterator position)	Removes the element at the position pointed by the iterator.
erase(const g)	Removes the value 'g' from the multiset.

Function	Definition
clear()	Removes all the elements from the multiset.
key_comp() / value_comp()	Returns the object that determines how the elements in the multiset are ordered (‘<‘ by default).
find(const g)	Returns an iterator to the element ‘g’ in the multiset if found, else returns the iterator to end.
count(const g)	Returns the number of matches to element ‘g’ in the multiset.
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 if found, else returns the iterator to end.
upper_bound(const g)	Returns an iterator to the first element that will go after the element ‘g’ in the multiset.
multiset::swap()	This function is used to exchange the contents of two multisets but the sets must be of the same type, although sizes may differ.
multiset::operator=	This operator is used to assign new contents to the container by replacing the existing contents.
multiset::emplace()	This function is used to insert a new element into the multiset container.
multiset equal_range()	Returns an iterator of pairs. The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
multiset::emplace_hint()	Inserts a new element in the multiset.
multiset::rbegin()	Returns a reverse iterator pointing to the last element in the multiset container.

Function	Definition
multiset::rend()	Returns a reverse iterator pointing to the theoretical element right before the first element in the multiset container.
multiset::cbegin()	Returns a constant iterator pointing to the first element in the container.
multiset::cend()	Returns a constant iterator pointing to the position past the last element in the container.
multiset::crbegin()	Returns a constant reverse iterator pointing to the last element in the container.
multiset::crend()	Returns a constant reverse iterator pointing to the position just before the first element in the container.
multiset::get_allocator()	Returns a copy of the allocator object associated with the multiset.

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.

Function	Definition
map::insert()	Insert elements with a particular key in the map container –> O(log n)
map:: count()	Returns the number of matches to element with key-value 'g' in the map. –> O(log n)
map equal_range()	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.

Function	Definition
map erase()	Used to erase elements from the container –> O(log n)
map rend()	Returns a reverse iterator pointing to the theoretical element right before the first key-value pair in the map(which is considered its reverse end).
map rbegin()	Returns a reverse iterator which points to the last element of the map.
map find()	Returns an iterator to the element with key-value ‘g’ in the map if found, else returns the iterator to end.
map crbegin() and crend()	crbegin() returns a constant reverse iterator referring to the last element in the map container. crend() returns a constant reverse iterator pointing to the theoretical element before the first element in the map.
map cbegin() and cend()	cbegin() returns a constant iterator referring to the first element in the map container. cend() returns a constant iterator pointing to the theoretical element that follows the last element in the multimap.
map emplace()	Inserts the key and its element in the map container.
map max_size()	Returns the maximum number of elements a map container can hold –> O(1)
map upper_bound()	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
map operator=	Assigns contents of a container to a different container, replacing its current content.
map lower_bound()	Returns an iterator to the first element that is equivalent to the mapped value with key-value ‘g’ or definitely will not go before the element with key-value ‘g’ in the map –> O(log n)

Function	Definition
map emplace_hint()	Inserts the key and its element in the map container with a given hint.
map value_comp()	Returns the object that determines how the elements in the map are ordered (‘<‘ by default).
map key_comp()	Returns the object that determines how the elements in the map are ordered (‘<‘ by default).
map::size()	Returns the number of elements in the map.
map::empty()	Returns whether the map is empty
map::begin() and end()	begin() returns an iterator to the first element in the map. end() returns an iterator to the theoretical element that follows the last element in the map
map::operator[]	This operator is used to reference the element present at the position given inside the operator.
map::clear()	Removes all the elements from the map.
map::at() and map::swap()	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 the same type, although sizes may differ.

Multimap:

Multimap is similar to a map with the addition that multiple elements can have the same keys. Also, it is NOT required that the key-value and mapped value pair have to be unique in this case. One important thing to note about multimap is that multimap keeps all the keys in sorted order always. These properties of multimap make it very much useful in competitive programming.

Function	Definition
multimap::operator=	It is used to assign new contents to the container by replacing the existing contents.
multimap::crbegin() and multimap::crend()	crbegin() returns a constant reverse iterator referring to the last element in the multimap container. crend() returns a constant reverse iterator pointing to the theoretical element before the first element in the multimap.
multimap::emplace_hint()	Insert the key and its element in the multimap container with a given hint.
multimap clear()	Removes all the elements from the multimap.
multimap empty()	Returns whether the multimap is empty.
multimap maxsize()	Returns the maximum number of elements a multimap container can hold.
multimap value_comp()	Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default).
multimap rend	Returns a reverse iterator pointing to the theoretical element preceding to the first element of the multimap container.
multimap::cbegin() and multimap::cend()	cbegin() returns a constant iterator referring to the first element in the multimap container. cend() returns a constant iterator pointing to the theoretical element that follows the last element in the multimap.

Function	Definition
multimap::swap()	Swap the contents of one multimap with another multimap of same type and size.
multimap rbegin	Returns an iterator pointing to the last element of the container.
multimap size()	Returns the number of elements in the multimap container.
multimap::emplace()	Inserts the key and its element in the multimap container.
multimap::begin() and multimap::end()	begin() returns an iterator referring to the first element in the multimap container. end() returns an iterator to the theoretical element that follows the last element in the multimap.
multimap upper_bound()	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.
multimap::count()	Returns the number of matches to element with key-value ‘g’ in the multimap.
multimap::erase()	Removes the key value from the multimap.
multimap::find()	Returns an iterator to the element with key-value ‘g’ in the multimap if found, else returns the iterator to end.
multimap equal_range()	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.

Function	Definition
multimap insert()	Used to insert elements in the multimap container.
multimap lower_bound()	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.
multimap key_comp()	Returns the object that determines how the elements in the multimap are ordered (‘<‘ by default).

Unordered Sets:

An **unordered_set** is an unordered associative container implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the **unordered_set** take constant time **O(1)** on an average which can go up to linear time **O(n)** in the worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

Function Name	Function Description
insert()	Insert a new {element} in the unordered_set container.
begin()	Return an iterator pointing to the first element in the unordered_set container.
end()	Returns an iterator pointing to the past-the-end-element.
count()	Count occurrences of a particular element in an unordered_set container.
find()	Search for an element in the container.

Function Name	Function Description
clear()	Removes all of the elements from an unordered_set and empties it.
cbegin()	Return a const_iterator pointing to the first element in the unordered_set container.
cend()	Return a const_iterator pointing to a past-the-end element in the unordered_set container or in one of its buckets.
bucket_size()	Returns the total number of elements present in a specific bucket in an unordered_set container.
erase()	Remove either a single element or a range of elements ranging from start(inclusive) to end(exclusive).
size()	Return the number of elements in the unordered_set container.
swap()	Exchange values of two unordered_set containers.
emplace()	Insert an element in an unordered_set container.
max_size()	Returns maximum number of elements that an unordered_set container can hold.
empty()	Check if an unordered_set container is empty or not.
equal_range	Returns range that includes all elements equal to a given value.
operator=	Copies (or moves) an unordered_set to another unordered_set and unordered_set::operator= is the corresponding operator function.

Function Name	Function Description
hash_function()	This hash function is a unary function that takes a single argument only and returns a unique value of type size_t based on it.
reserve()	Used to request a capacity change of unordered_set.
bucket()	Returns the bucket number of a specific element.
bucket_count()	Returns the total number of buckets present in an unordered_set container.
load_factor()	Returns the current load factor in the unordered_set container.
rehash()	Set the number of buckets in the container of unordered_set to a given size or more.
max_load_factor()	Returns(Or sets) the current maximum load factor of the unordered set container.
emplace_hint()	Inserts a new element in the unordered_set only if the value to be inserted is unique, with a given hint.
== operator	The '==' is an operator in C++ STL that performs an equality comparison operation between two unordered sets and unordered_set::operator== is the corresponding operator function for the same.
key_eq()	Returns a boolean value according to the comparison. It returns the key equivalence comparison predicate used by the unordered_set.

Function Name	Function Description
operator!=	The != is a relational operator in C++ STL which compares the equality and inequality between unordered_set containers.
max_bucket_count()	Find the maximum number of buckets that unordered_set can have.

unordered_multiset:

The unordered_multiset in C++ STL is an unordered associative container that works similarly to an unordered_set. The only difference is that we can store multiple copies of the same key in this container.

It is also implemented using a hash table so the time complexity of the operations is O(1) on average which can go up to linear time O(n) in the worst case. Internally when an existing value is inserted, the data structure increases its count which is associated with each value. A count of each value is stored in unordered_multiset, it takes more space than unordered_set (if all values are distinct).

Due to hashing of elements, it has no particular order of storing the elements so all element can come in any order but duplicate element comes together.

Function Name	Function Description
insert()	Inserts new elements in the unordered_multiset. This increases the container size.
begin()	Returns an iterator pointing to the first element in the container or the first element in one of its buckets.
end()	Returns an iterator pointing to the position immediately after the last element in the container or to the position immediately after the last element in one of its buckets.
empty()	It returns true if the unordered_multiset container is empty. Otherwise, it returns false.

Function Name	Function Description
find()	Returns an iterator that points to the position which has the element val.
cbegin()	Returns a constant iterator pointing to the first element in the container or the first element in one of its buckets.
cend()	Returns a constant iterator pointing to the position immediately after the last element in the container or immediately after the last element in one of its buckets.
equal_range()	Returns the range in which all the elements are equal to a given value.
emplace()	Inserts a new element in the unordered_multiset container.
clear()	Clears the contents of the unordered_multiset container.
count()	Returns the count of elements in the unordered_multiset container which is equal to a given value.
size()	The size() method of unordered_multiset is used to count the number of elements of unordered_set it is called with.
max_size	The max_size() of unordered_multiset takes the maximum number of elements that the unordered_multiset container is able to hold.
swap()	Swaps the contents of two unordered_multiset containers.
erase()	Used to remove either a single element or, all elements with a definite value or, a range of elements ranging from the start(inclusive) to the end(exclusive).

Function Name	Function Description
bucket()	Returns the bucket number in which a given element is. Bucket size varies from 0 to bucket_count-1.
bucket_size()	Returns the number of elements in the bucket that has the element val.
reserve()	The reverse() function of unordered_multiset sets the number of buckets in the container (bucket_count) to the most appropriate to contain at least n elements.
max_bucket_count()	Returns the maximum number of buckets that the unordered multiset container can have.
load_factor()	Returns the current load factor in the unordered_multiset container.
max_load_factor()	Returns the maximum load factor of the unordered_multiset container.
bucket_count()	Returns the total number of buckets in the unordered_multiset container.
hash_function()	This hash function is a unary function that takes a single argument only and returns a unique value of type size_t based on it.
rehash()	Sets the number of buckets in the container to N or more.
key_eq()	Returns a boolean value according to the comparison.
emplace_hint()	Inserts a new element in the unordered_multiset container.

Function Name	Function Description
get_allocator	This function gets the stored allocator object and returns the allocator object used to construct the container.
operator =	The '=' is an operator in C++ STL that copies (or moves) an unordered_multiset to another unordered_multiset and unordered_multiset::operator= is the corresponding operator function.

unordered_map:

unordered_map is an associated container that stores elements formed by the combination of a key value and a mapped value. The key value is used to uniquely identify the element and the mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined. In simple terms, an ***unordered_map*** is like a data structure of dictionary type that stores elements in itself. It contains successive pairs (key, value), which allows fast retrieval of an individual element based on its unique key.

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

Methods/Functions	Description
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

Methods/Functions	Description
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 the 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
find()	Returns iterator to the element
empty()	Checks whether the container is empty in the unordered_map container
erase()	Erase elements in the container in the unordered_map container

unordered_multimap:

We have discussed unordered_map in our previous post, but there is a limitation, we can not store duplicates in unordered_map, that is if we have a key-value pair already in our unordered_multimap and another pair is inserted, then both will be there whereas in case of unordered_map the previous value corresponding to the key is updated by the new value that is only would be there. Even can exist in unordered_multimap twice.

Methods of unordered_multimap:

- begin()– Returns an iterator pointing to the first element in the container or to the first element in one of its bucket.
- end()– Returns an iterator pointing to the position after the last element in the container or to the position after the last element in one of its bucket.
- count()– Returns the number of elements in the container whose key is equal to the key passed in the parameter.
- cbegin()– Returns a constant iterator pointing to the first element in the container or to the first element in one of its bucket.
- cend()– Returns a constant iterator pointing to the position after the last element in the container or to the position after the last element in one of its bucket.
- clear() – Clears the contents of the unordered_multimap container.
- size()– Returns the size of the unordered_multimap. It denotes the number of elements in that container.
- swap()– Swaps the contents of two unordered_multimap containers. The sizes can differ of both the containers.
- find()– Returns an iterator which points to one of the elements which have the key k.
- bucket_size()– Returns the number of elements in the bucket n.
- empty()– It returns true if the unordered_multimap container is empty. Otherwise, it returns false.
- equal_range()– Returns the range in which all the element's key is equal to a key.
- operator=– Copy/Assign/Move elements from different container.
- max_size()– Returns the maximum number of elements that the unordered_multimap container can hold.
- load_factor()– Returns the current load factor in the unordered_multimap container.
- key_eq()– Returns a boolean value according to the comparison.
- emplace()– Inserts a new {key, element} in the unordered_multimap container.
- emplace_hint()– Inserts a new {key:element} in the unordered_multimap container.
- bucket_count()– Returns the total number of buckets in the unordered_multimap container.
- bucket()– Returns the bucket number in which a given key is.
- max_load_factor()– Returns the maximum load factor of the unordered_multimap container.

- rehash()– Sets the number of buckets in the container to N or more.
- reserve()– Sets the number of buckets in the container (bucket_count) to the most appropriate number so that it contains at least n elements.
- hash_function()– This hash function is a unary function that takes a single argument only and returns a unique value of type size_t based on it.
- max_bucket_count()– Returns the maximum number of buckets that the unordered multimap container can have.

Functors:

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.

Iterators:

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequences of numbers, characters etc. They reduce the complexity and execution time of the 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 *after* end position of the container.

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

4. next() :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.

5. prev() :- This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

6. 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.