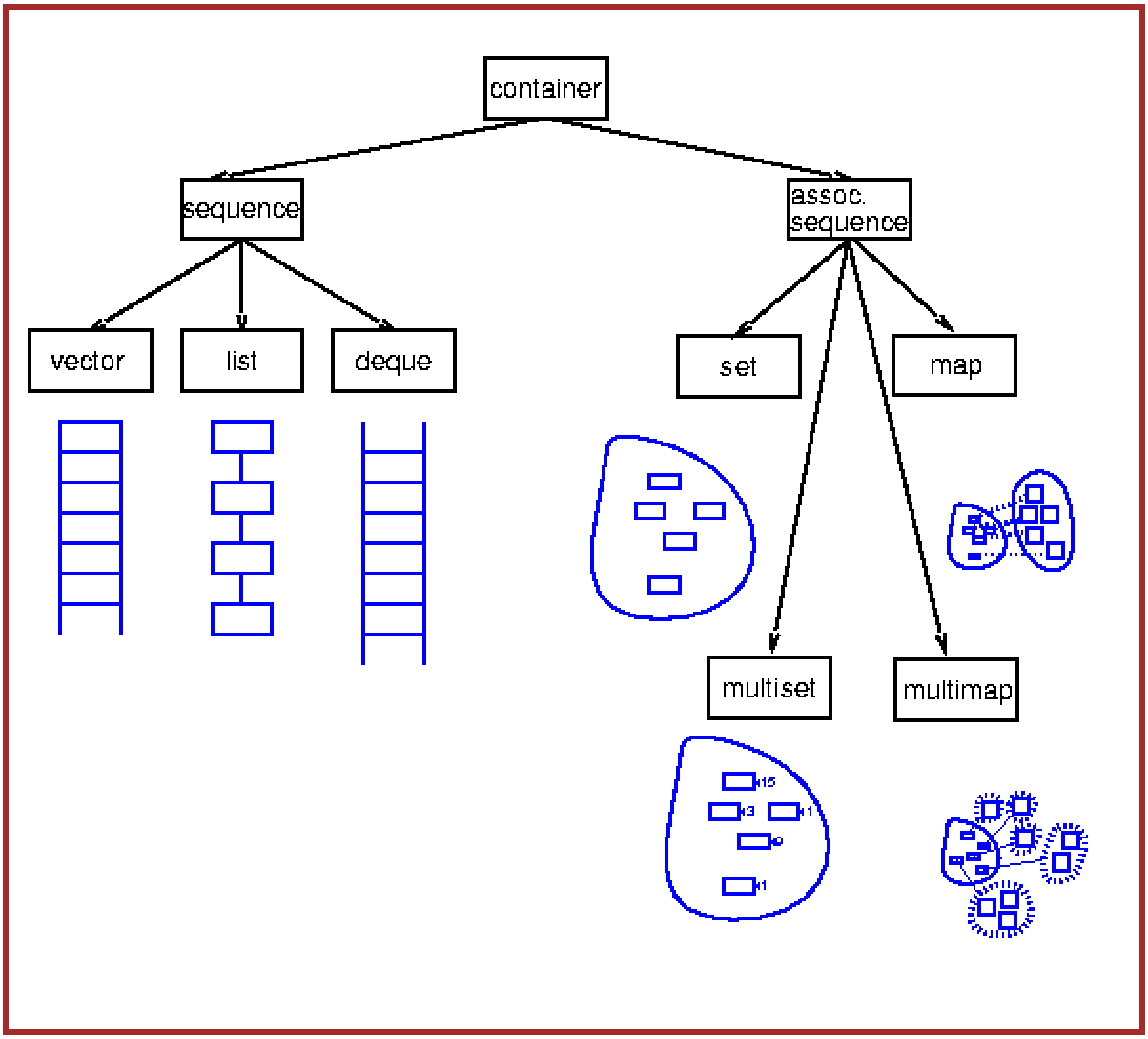
STL

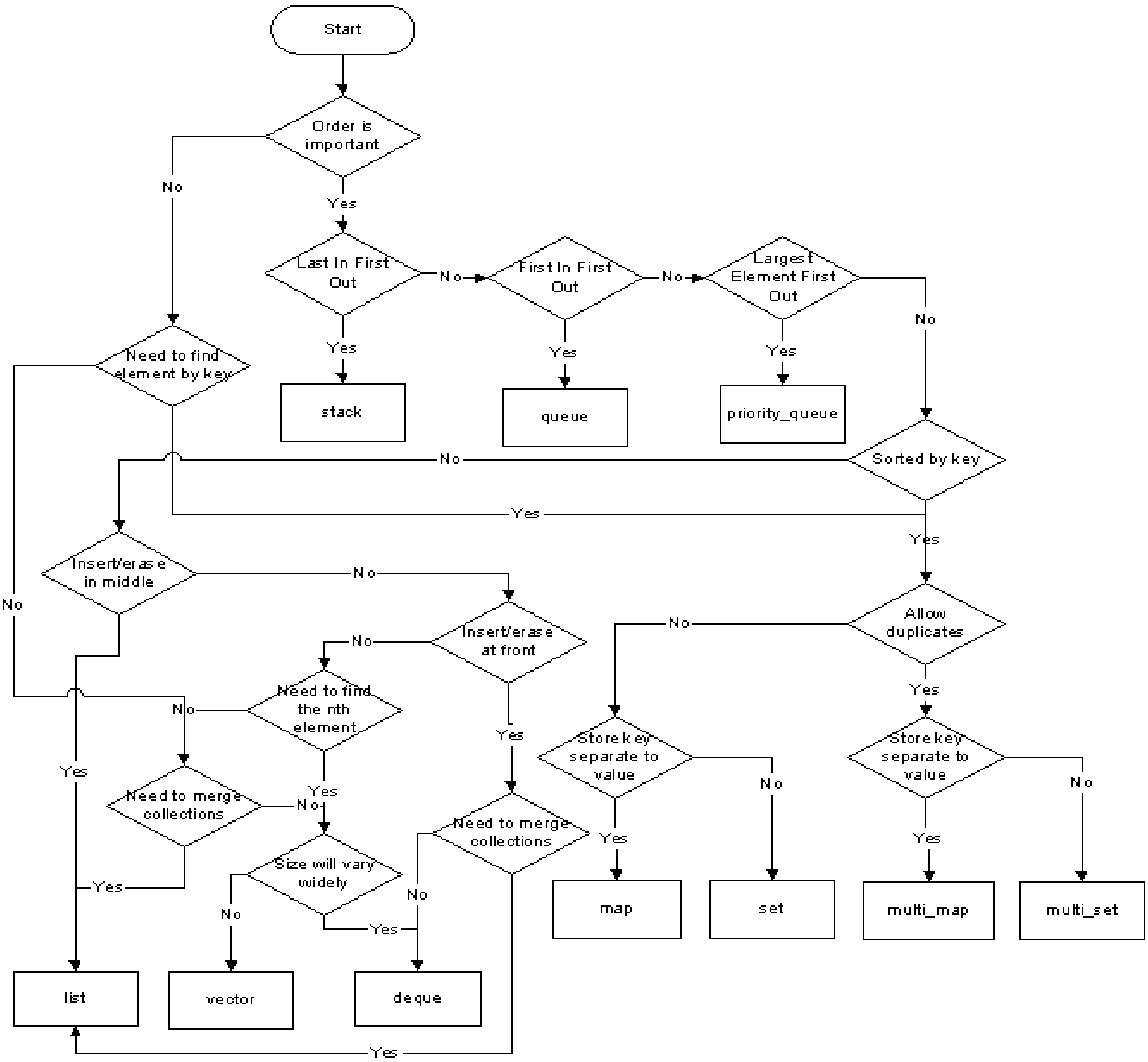
- STL (Standard Template Library)

At the core of the C++ Standard Template Library are following three well-structured components:

Component	Description
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.


container
sequence
assoc.
sequence
vector
list
deque
set
map
multiset
multimap

- [How can I efficiently select a Standard Library container in C++?]()



- **Vector**

  - The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets on regular pointers to elements.
  - Look this simple vector program.

```
#include <iostream>
#include <vector>

int main()
{
        // Create a vector containing integers
       std::vector<int> v = {7, 5, 16, 8};

       // Add two more integers to vector
       v.push_back(25);
       v.push_back(13);

       // Iterate and print values of vector
       for(int n : v) {
```

```
            std::cout << n << '\t';
        }
}
```
- O/p: 7 5 16 8 25 13

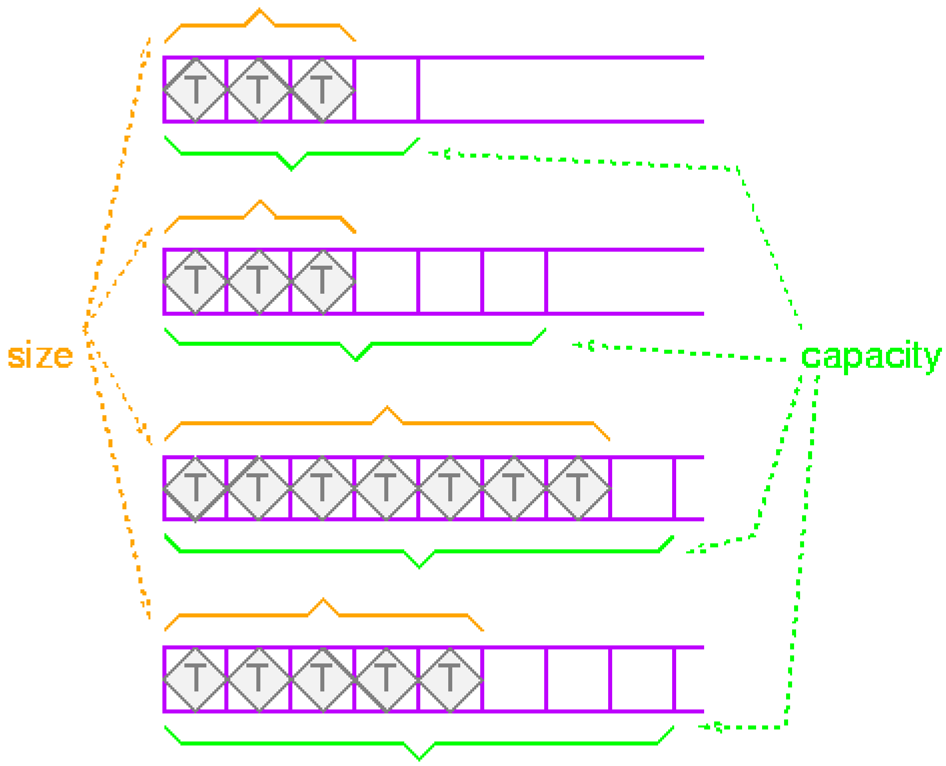
Vector Functions:

Size - returns the number of elements.

max_size - returns the maximum possible number of elements.

reserve - Requests that the vector capacity be at least enough to contain *n* elements. If *n* is greater than the current vector capacity, the function causes the container to reallocate its storage increasing its capacity to *n*.

Capacity - returns the number of elements that can be held in currently allocated storage



resize- Resizes the container so that it contains n elements.

void resize (size_type n, value_type val = value_type());

If n is smaller than the current container size, the content is reduced to its first n elements, removing those beyond (and destroying them).

If n is greater than the current container size, the content is expanded by inserting at the end as many elements as needed to reach a size of n. If val is specified, the new elements are initialized as copies of val, otherwise, they are value-initialized.

```
std::vector<int> v1;
v1.resize(1000); //allocation + instance creation
cout <<(v1.size() == 1000)<< endl;   //prints 1
cout <<(v1.capacity()==1000)<< endl; //prints 1

std::vector<int> v2;
v2.reserve(1000); //only allocation
cout <<(v2.size() == 1000)<< endl;   //prints 0 here size is 0
cout <<(v2.capacity()==1000)<< endl; //prints 1
```

push_back:

- Adds a new element at the end of the vector.
- Causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

pop_back:
Removes the last element in the vector, effectively reducing the container size by one.

erase:

Removes from the vector either a single element (position) or a range of elements ([first,last)).

This effectively reduces the container size by the number of elements removed, which are destroyed.

vectors use an array as their underlying storage, ***erasing elements in positions other than the vector end causes the container to relocate all the elements*** after the segment erased to their new positions. This is generally an inefficient operation.

E.g

```
std::vector<int> myvector;

// set some values (from 1 to 10)
for (int i=1; i<=10; i++) myvector.push_back(i);

// erase the 6th element
myvector.erase (myvector.begin()+5);

// erase the first 3 elements:
myvector.erase (myvector.begin(),myvector.begin()+3);
```

Difference b/w operator[] and at() To Access Vector In C++
1. Both are used to access elements in vector.
2. **operator[] don't do range checking, at() does range checking** before accessing
3. operator[] don't throw when it is **out of bound** (undefined behaviour), but at() throw if it is out of bound.
4. **operator[] faster** and **at() is slower** as compare to operator[]

- ## Stack

- Stacks are a type of container adaptor, specifically designed to operate in a **LIFO** context (last-in first-out)
- Stacks are ***implemented as containers adaptors***, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements.

The container shall support the following operations:
•empty
•size
•back
•push_back
•pop_back

Constructor:

explicit stack (const container_type& ctnr = container_type());

```
template<class T, class Container = std::deque<T>> class stack;
```

e.g

```
std::deque<int> mydeque (3,100);          // deque with 3 elements
std::vector<int> myvector (2,200);        // vector with 2 elements

std::stack<int> first;                    // empty stack
std::stack<int> second (mydeque);         // stack initialized to copy of deque

std::stack<int,std::vector<int> > third;  // empty stack using vector. Here stack is
                                          // using vector as underlying
container.
std::stack<int,std::vector<int> > fourth (myvector);
```

- ## deque

```
template < class T, class Alloc = allocator<T> > class deque;
```

acronym of double-ended queue

Internaly Imagine it as a vector of vectors. it can be implemented otherwise, like a **linked list of arrays**.

Modifiers:

assign	
	Assign container content (public member function)
push_back	
	Add element at the end (public member function)
push_front	
	Insert element at beginning (public member function)
pop_back	
	Delete last element (public member function)
pop_front	
	Delete first element (public member function)
insert	
	Insert elements (public member function)
erase	
	Erase elements (public member function)
swap	
	Swap content (public member function)
clear	
	Clear content (public member function)
emplace	
	Construct and insert element (public member function)
emplace_front	
	Construct and insert element at beginning (public member function)
emplace_back	
	Construct and insert element at the end (public member function)

Capacity:

size	
	Return size (public member function)
max_size	
	Return maximum size (public member function)
resize	
	Change size (public member function)

❖ Vector Vs. Deque

[(1) Vector Vs. Deque | LinkedIn]

• List

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

splice() function in C++ STL

The **list::splice()** is a built-in function in C++ STL which is used to transfer elements from one list to another. The splice() function can be used in three ways:

1. Transfer all the elements of list *x* into another list at some *position*.
2. Transfer only the element pointed by *i* from list *x* into the list at some *position*.
3. Transfers the range *[first, last)* from list *x* into another list at some *position*.

• Maps

Maps are associative containers that store elements formed by a combination of a **key value** and a mapped value.

Associative
Elements in associative containers are referenced by their ***key*** and not by their absolute position in the container.

Ordered

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

Unique keys
No two elements in the container can have equivalent *keys*.

Constructor:

empty (1)	`explicit map (const key_compare& comp = key_compare(), const allocator_type& alloc = allocator_type());`
range (2)	`template <class InputIterator> map (InputIterator first, InputIterator last, const key_compare& comp = key_compare(), const allocator_type& alloc = allocator_type());`
copy (3)	`map (const map& x);`

e.g:

```
std::map<char,int> first;

std::map<char,int> second (first.begin(),first.end());

std::map<char,int> third (second);
```

Insert:

```
  mymap.insert ( std::pair<char,int>('a',100) );

 // Second insert function version (range insertion):
  std::map<char,int> anothermap;
  anothermap.insert(mymap.begin(),mymap.find('c'));

  // showing contents:
  std::cout << "mymap contains:\n";
  for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

cbegin:

Returns a `const_iterator` pointing to the first element in the container.

Cend:

Returns a `const_iterator` pointing to the *past-the-end* element in the container.

```
e.g:
  std::map<char,int>::Iterator it = mymap.cbegin();
  for (auto it = mymap.cbegin(); it != mymap.cend(); ++it)
    std::cout << " [" << (*it).first << ':' << (*it).second << ']';
```

find:

Searches the container for an element with a key equivalent to k and returns an iterator to it if found, otherwise it returns an iterator to map::end.

```
e.g:
  it = mymap.find('b');
  if (it != mymap.end())
    mymap.erase (it);
```

`map::lower_bound` : will points returns an iterator pointing to the first element that is not less than key.

`map::upper_bound` : returns an iterator pointing to the first element that is greater than key.

e.g.

```
 mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;

  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)

  mymap.erase(itlow,itup);        // erases [itlow,itup)

 // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

o/p:

```
a => 20
e => 100
```

```
intmap[1]=10;
```

```
intmap[2]=20;
intmap[4]=40; // <<---both lower_bound(3)/upper_bound(3) will points to here
intmap[5]=50;

it1=intmap.lower_bound (3);
it2=intmap.upper_bound (3);
```

❖ Using User defined class objects as keys in std::map

By default std::map uses "**operator <**" as sorting criteria for keys. For default data types like int and std::string etc, operator < is available by default but for User defined classes operator < is not available by default.

Therefore, to use user defined class objects as keys in std::map we should have either,

•Default sorting criteria i.e. operator < defined for our Class.

•std::map should be assigned with an external sorting criteria i.e. comparator that can compare two objects

of your user defined class.

Lets understand by example,
Suppose our class is User that has id and name properties. To use this class as key in std::map we will overload operator <.

```
class User
{
	std::string m_id;
	std::string m_name;
public:
	User(std::string name, std::string id)
		:m_id(id), m_name(name)
	{}
	const std::string& getId() const {
		return m_id;
	}
	const std::string& getName() const {
		return m_name;
	}
	bool operator< (const User& userObj) const
	{
		if(userObj.m_id < this->m_id)
			return true;
	}
};

std::map<User, int> m_UserInfoMap;

m_UserInfoMap.insert(std::make_pair<User, int>(User("Mr.X", "3"), 100) );
m_UserInfoMap.insert(std::make_pair<User, int>(User("Mr.X", "1"), 120) );
m_UserInfoMap.insert(std::make_pair<User, int>(User("Mr.Z", "2"), 300) );

std::map<User, int>::iterator it = m_UserInfoMap.begin();
for(; it != m_UserInfoMap.end(); it++)
{
	std::cout<<it->first.getName()<<" :: "<<it->second<<std::endl;
}
```

O/P:
Mr.X :: 100
Mr.Z :: 300
Mr.X :: 120

As we can see in output above, std::map can contain User object with unique Ids only therefore there are two entries for 2 Mr.X objects.
Now suppose we want to change the sorting criteria of keys i.e for User objects, instead of comparing by ID we want to compare them by name property.
To achieve this we have two options either change the definition of operator < or by using external sorting criteria i.e. comparators. For example,

```
struct UserNameComparator
{
    bool operator()(const User & left, const User & right) const
    {
        return (left.getName() > right.getName());
    }
};


 std::map<User, int, UserNameComparator> m_UserInfoMap;

    m_UserInfoMap.insert(std::make_pair<User, int>(User("Mr.X", "3"), 100) );
    m_UserInfoMap.insert(std::make_pair<User, int>(User("Mr.X", "1"), 120) );
    m_UserInfoMap.insert(std::make_pair<User, int>(User("Mr.Z", "2"), 300) );

    std::map<User, int, UserNameComparator>::iterator it =
m_UserInfoMap.begin();
    for(; it != m_UserInfoMap.end(); it++)
    {
        std::cout<<it->first.getName()<<" :: "<<it->second<<std::endl;
```

```
}
```

O/P:

Mr.Z :: 300
Mr.X :: 100
As we can see in above output there is only 1 entry for name Mr.X because this time because we are comparing keys i.e User objects by name instead of Id.

Extra Note:
std::set, std::multiset, std::map and std::multimap are guaranteed to be ***ordered according to the keys*** (and the criterion supplied). When we print any my map or set using iterator It will print always in sorted order.

- ## Multimap

Multimaps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order, and where multiple elements can have equivalent keys.

You are allowed to insert multiple value with same key.

```
mymultimap.insert ( std::pair<char,int>('a',100) );
mymultimap.insert ( std::pair<char,int>('z',150) );

mymultimap.insert ( std::pair<char,int>('a',200) );
mymultimap.insert ( std::pair<char,int>('a',100) );

e.g:
int main()
{
    std::multimap<int, char> dict{
        {1, 'A'},
        {2, 'B'},
        {2, 'C'},
        {2, 'D'},
        {4, 'E'},
        {3, 'F'}
    };

    auto range = dict.equal_range(2);
    for (auto i = range.first; i != range.second; ++i)
    {
        std::cout << i->first << ": " << i->second << '\n';
    }
}
```

- ## Sets

- Sets are containers that store unique elements following a specific order.
- In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique.
- The value of the elements in a **set cannot be modified once in the container** (the elements are always ***const***), but they can be inserted or removed from the container.

Sets are typically implemented as **binary search trees**(Red-Black Tree).

Constructor:

empty (1)	`explicit set (const key_compare& comp = key_compare(), const allocator_type& alloc = allocator_type());`
range (2)	`template <class InputIterator> set (InputIterator first, InputIterator last, const key_compare& comp = key_compare(), const allocator_type& alloc = allocator_type());`
copy (3)	`set (const set& x);`

e.g:

```
std::set<int> first;                           // empty set of ints

int myints[]= {10,20,30,40,50};
std::set<int> second (myints,myints+5);        // range

std::set<int> third (second);                  // a copy of second

std::set<int> fourth (second.begin(), second.end());  // iterator ctor.
```

```
int myints[]= {10,20,30,40,50};
std::set<int> second (myints,myints+5);        // range
second.insert(10);
```

O/p: 10,20,30,40,50

Capacity:

empty	
	Test whether container is empty (public member function)
size	
	Return container size (public member function)
max_size	
	Return maximum size (public member function)

Modifiers:

insert	
	Insert element (public member function)
erase	
	Erase elements (public member function)
swap	
	Swap content (public member function)
clear	
	Clear content (public member function)
emplace	

	Construct and insert element (public member function)
emplace_hint	
	Construct and insert element with hint (public member function)

Operations:

find	
	Get iterator to element (public member function)
count	
	Count elements with a specific value (public member function)
lower_bound	
	Return iterator to lower bound (public member function)
upper_bound	
	Return iterator to upper bound (public member function)
equal_range	
	Get range of equal elements (public member function)

- ## Multiset

Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.

Multisets are typically implemented as binary search trees.

e.g:

```
int myints[]= {10,20,30,40,50};
std::multiset<int> second (myints,myints+5);        // range
second.insert(10);
```

```
O/p: 10,10,20,30,40,50
```

- ## Unordered Sets

An ***unordered_set*** is 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*** takes constant time **O(1)** on an average which can go up to linear time **O(n)** in worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.
The **unordered_set** can contain key of any type – predefined or user-defined data structure but when we define key of type user define the type, we need to specify our comparison function according to which keys will be compared.

❖ Sets vs Unordered Sets

Set is an ordered sequence of unique keys whereas ***unordered_set is a set in which key can be stored in any order***, so unordered. Set is implemented as a ***balanced search tree structure*** that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(log n) while for unordered_set, it is O(1).

❖ Methods on Unordered Sets:

For unordered_set many functions are defined among which most used are the size and empty for capacity, find for searching a key, insert and erase for modification.
The Unordered_set allows only unique keys, for duplicate keys unordered_multiset

```
// declaring set for storing string data-type
    unordered_set <string> stringSet ;

    // inserting various string, same string will be stored
    // once in set

    stringSet.insert("code") ;
    stringSet.insert("in") ;
    stringSet.insert("c++") ;
    stringSet.insert("is") ;
    stringSet.insert("fast") ;

    // now iterating over whole set and printing its
    // content
    cout << "\nAll elements : ";
    unordered_set<string> :: iterator itr;
    for (itr = stringSet.begin(); itr != stringSet.end(); itr++)
        cout << (*itr) << endl;
```

O/P: Order is not fix

```
All elements :

is

fast

c++

in

code
```

- ## unordered_map

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

```
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;
```

O/P:

```
Contribute 30

GeeksforGeeks 10

Practice 20
```

❖ 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.
The map is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of map operations is O(log n) while for unordered_map, it is O(1) on average.

❖ Difference bet insert and emplace:

- The primary difference is that insert takes an object whose type is the same as the container type and **copies** that argument into the container. emplace takes a more or less arbitrary argument list and **constructs** an object in the container from those arguments.
- The advantage of emplace is, it does **in-place** insertion and **avoids an unnecessary copy** of object.

```
e.g. 1
int main()
 {

   std::set<A> set;
   set.insert(A(10)); // This will costruct and the copy object
   set.emplace(20);  // This will just create object directly i set with
      //given params
 }
e.g. 2
 // declaring map
 multimap<pair<char, int>> ms;

 // using emplace() to insert pair in-place
 ms.emplace('a', 24);

 // Below line would not compile
 // ms.insert('b', 25);

 // using emplace() to insert pair in-place
 ms.emplace(make_pair('b', 25));
```

Algorithms

Sorting:

sort	
	Sort elements in range (function template)
stable_sort	
	Sort elements preserving order of equivalents (function template)
partial_sort	
	Partially sort elements in range (function template)
partial_sort_copy	
	Copy and partially sort range (function template)
is_sorted	
	Check whether range is sorted (function template)
is_sorted_until	
	Find first unsorted element in range (function template)
nth_element	
	Sort element in range (function template)

Modifying sequence operations:

copy	
	Copy range of elements (function template)
copy_n	
	Copy elements (function template)
copy_if	
	Copy certain elements of range (function template)
copy_backward	
	Copy range of elements backward (function template)
move	
	Move range of elements (function template)
move_backward	
	Move range of elements backward (function template)
swap	
	Exchange values of two objects (function template)
swap_ranges	
	Exchange values of two ranges (function template)
iter_swap	
	Exchange values of objects pointed to by two iterators (function template)
transform	
	Transform range (function template)
replace	
	Replace value in range (function template)
replace_if	
	Replace values in range (function template)
replace_copy	
	Copy range replacing value (function template)
replace_copy_if	

	Copy range replacing value (function template)
fill	
	Fill range with value (function template)
fill_n	
	Fill sequence with value (function template)
generate	
	Generate values for range with function (function template)
generate_n	
	Generate values for sequence with function (function template)
remove	
	Remove value from range (function template)
remove_if	
	Remove elements from range (function template)
remove_copy	
	Copy range removing value (function template)
remove_copy_if	
	Copy range removing values (function template)
unique	
	Remove consecutive duplicates in range (function template)
unique_copy	
	Copy range removing duplicates (function template)
reverse	
	Reverse range (function template)
reverse_copy	
	Copy range reversed (function template)
rotate	
	Rotate left the elements in range (function template)
rotate_copy	
	Copy range rotated left (function template)
random_shuffle	
	Randomly rearrange elements in range (function template)
shuffle	
	Randomly rearrange elements in range using generator (function template)

Min/max:

min	
	Return the smallest (function template)
max	
	Return the largest (function template)
minmax	
	Return smallest and largest elements (function template)
min_element	
	Return smallest element in range (function template)
max_element	
	Return largest element in range (function template)
minmax_element	
	Return smallest and largest elements in range (function template)

Non-modifying sequence operations:

all_of	
	Test condition on all elements in range (function template)
any_of	
	Test if any element in range fulfills condition (function template)
none_of	
	Test if no elements fulfill condition (function template)
for_each	
	Apply function to range (function template)
find	
	Find value in range (function template)
find_if	
	Find element in range (function template)
find_if_not	
	Find element in range (negative condition) (function template)
find_end	
	Find last subsequence in range (function template)
find_first_of	
	Find element from set in range (function template)
adjacent_find	
	Find equal adjacent elements in range (function template)
count	
	Count appearances of value in range (function template)
count_if	
	Return number of elements in range satisfying condition (function template)
mismatch	
	Return first position where two ranges differ (function template)
equal	
	Test whether the elements in two ranges are equal (function template)
is_permutation	
	Test whether range is permutation of another (function template)
search	
	Search range for subsequence (function template)
search_n	
	Search range for elements (function template)

- ## std::sort

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );
```

It is a function template. Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved.

```
std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

// sort using the default operator<
std::sort(s.begin(), s.end()); o/p:0 1 2 3 4 5 6 7 8 9

// sort using a standard library compare function object

std::sort(s.begin(), s.end(), std::greater<int>()); o/p: 9 8 7 6 5 4 3 2 1 0

// sort using a custom function object

struct {
	bool operator()(int a, int b)
	{
		return a < b;
	}
} customLess;
std::sort(s.begin(), s.end(), customLess);o/p:0 1 2 3 4 5 6 7 8 9

// sort using a lambda expression

std::sort(s.begin(), s.end(), [](int a, int b) {
	return b < a;
}); o/p: 9 8 7 6 5 4 3 2 1 0
```

- ## std::partial_sort

Rearranges elements such that the range [first, middle) contains the sorted middle - first smallest elements in the range [first, last). The order of equal elements is not guaranteed to be preserved. The order of the remaining elements in the range[middle, last) is unspecified.

```
std::array<int, 10> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

std::partial_sort(s.begin(), s.begin() + 3, s.end());
```

O/p:0 1 2 7 8 6 5 9 4 3

❖ Where can it be used ?

1. **Finding the largest element**: Since, with std::partial_sort, we can partially sort the container till whichever position we would like to. So, if we just sort the first position and use a function object , we can find the largest element, without having to sort the entire container.

```
// Using std::partial_sort
std::partial_sort(v.begin(), v.begin() + 1, v.end(),
                  greater<int>());

// Displaying the largest element after applying
// std::partial_sort
```

```
ip = v.begin();
cout << "The largest element is = " << *ip;
```

2. Finding the smallest element: Similar to finding the largest element, we can also find the smallest element in the container in the previous example.

```
// Using std::partial_sort
   std::partial_sort(v.begin(), v.begin() + 1, v.end());
```

❖ Point to remember:

- **std::sort() vs std::partial_sort():** Some of you might think that why are we using std::partial_sort, in place we can use std::sort() for the limited range, but remember, if we use std::sort with a partial range, then only elements within that range will be considered for sorting, while all other elements outside the range will not be considered for this purpose, whereas with std::partial_sort(), all the elements will be considered for sorting.
- partial_sort rearranges elements such that the range [first, middle) contains the sorted elements.
- The order of equal elements is not guaranted to be preserved.
- The order of remaining elements is unspecified.

• std::merge

```
template< class InputIt1, class InputIt2, class OutputIt >

OutputIt merge( InputIt1 first1, InputIt1 last1,
             InputIt2 first2, InputIt2 last2,
                OutputIt d_first );
template< class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt merge( InputIt1 first1, InputIt1 last1,
             InputIt2 first2, InputIt2 last2,
                OutputIt d_first, Compare comp );
```

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at d_first.

```
// merge

std::vector<int> dst;
std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(), std::back_inserter(dst));
```

O/P:

v1 : 0 1 3 4 4 5 5 8 8 9

v2 : 0 2 2 3 6 6 8 8 8 9
dst: 0 0 1 2 2 3 3 4 4 5 5 6 6 8 8 8 8 8 9 9

• std::min

```
template< class T >
const T& min( const T& a, const T& b );
```

```
template< class T >
constexpr const T& min( const T& a, const T& b );
```

Returns the smaller of the given values.

-Returns the smaller of a and b.

-Returns the smallest of the values in initializer list ilist.

```
std::cout << "smaller of 1 and 9999: " << std::min(1, 9999) << '\n'
          << "smaller of 'a', and 'b': " << std::min('a', 'b') << '\n'
          << "shortest of \"foo\", \"bar\", and \"hello\": " <<
std::min( { "foo", "bar", "hello" },
          [](const std::string& s1, const std::string& s2) {
              return s1.size() < s2.size();
          }) << '\n';
```

O/P:

1

a

foo

- ## std::min_element

Finds the smallest element in the range [first, last).

- Elements are compared using operator<.
- Elements are compared using the given binary comparison function comp.

e.g :

```
std::vector<int> v{3, 1, 4, 1, 5, 9};

std::vector<int>::iterator result = std::min_element(std::begin(v), std::end(v));            o/p:1
```

One of min version use std::min_element internally.

```
template< class T >
T min( std::initializer_list<T> ilist)
{
    return *std::min_element(ilist.begin(), ilist.end());
}
```

e.g:

```
std::min( { 9, 5, 3, 45, 21, 2, 14 }) o/p:3
```

- ## std::max

Returns the greater of the given values.

- Returns the greater of a and b.

- Returns the greatest of the values in initializer list ilist.

e.g:

For above example of std::min with use of std::max

O/P: 9999, b, hello

- ## std::max_element

Finds the greatest element in the range [first, last).

  - Elements are compared using operator<.
  - Elements are compared using the given binary comparison function comp.

e.g:

```
static bool abs_compare(int a, int b)
{
        return (std::abs(a) < std::abs(b));
}

        std::vector<int> v{ 3, 1, -14, 1, 5, 9 };
        std::vector<int>::iterator result;

        result = std::max_element(v.begin(), v.end()); O/P: 9

        result = std::max_element(v.begin(), v.end(), abs_compare); O/P:-14
```

One of max version use std::max_element internally.

```
template< class T >
T max( std::initializer_list<T> ilist)
{
        return *std::max_element(ilist.begin(), ilist.end());
}
```

e.g:

```
std::max( { 9, 5, 3, 45, 21, 2, 14 })
```

- ## std::minmax

Returns the lowest and the greatest of the given values.

  - Returns references to the smaller and the greater of a and b.
  - Returns the smallest and the greatest of the values in initializer list ilist.

- ### Return value

1-2) Returns the result of std::pair<const T&, const T&>(a, b) if a<b or if a is equivalent to b. Returns the result of std::pair<const T&, const T&>(b, a) if b<a.

e.g:

```
std::pair<int, int> bounds = std::minmax(std::rand() % v.size(),
                                         std::rand() % v.size());
```

• std::minmax_element

Finds the smallest and greatest element in the range [first, last).

-Elements are compared using operator<.

-Elements are compared using the given binary comparison function comp.

e.g:

```
std::vector<int> v = { 3, 9, 1, 4, 2, 5, 9 };

auto result = std::minmax_element(v.begin(), v.end()); O/P: Min=1 Max=9
```

• std::find, std::find_if, std::find_if_not

Returns the first element in the range [first, last) that satisfies specific criteria:

- find searches for an element equal to value
- find_if searches for an element for which predicate p returns true
- find_if_not searches for an element for which predicate q returns false

Return value

Iterator to the first element satisfying the condition or last if no such element is found.

e.g:

```
std::vector<int> v{ 0, 1, 2, 3, 4 };

auto result1 = std::find(std::begin(v), std::end(v), n1);
auto result2 = std::find(std::begin(v), std::end(v), n2);

if (result1 != std::end(v)) {
    std::cout << "v contains: " << n1 << '\n';
}
else {
    std::cout << "v does not contain: " << n1 << '\n';
}

bool IsOdd(int i)
{
    return i % 2;
}

it = std::find_if(vec.begin(), vec.end(), IsOdd);
```

- # std::count, std::count_if

Returns the number of elements in the range [first, last) satisfying specific criteria.

- counts the elements that are equal to value.
- counts elements for which predicate p returns true.

Return value

number of elements satisfying the condition.

e.g:

1.
```
int data[] = { 1, 2, 3, 4, 4, 3, 7, 8, 9, 10 };
std::vector<int> v(data, data + 10);

int target1 = 3;
int target2 = 5;
int num_items1 = std::count(v.begin(), v.end(), target1); //O/P: count: 2
int num_items2 = std::count(v.begin(), v.end(), target2); //O/P: count: 0
```

2. This example uses a lambda expression to count elements divisible by 3.

```
int num_items1 = std::count_if(v.begin(), v.end(), [](int i) {return i % 3 == 0; });
```

- # std::equal

Returns true if the range [first1, last1) is equal to the range [first2, last2), and false otherwise.

Return value

If the elements in the two ranges are equal, returns true. Otherwise returns false.

e.g:

1.
```
bool flag = std::equal(s.begin(), s.end(), v.begin(), v.end());
```

2.
```
bool is_palindrome(const std::string& s)
{
    return std::equal(s.begin(), s.begin() + s.size()/2, s.rbegin());
}
```

- # std::copy, std::copy_if

Copies the elements in the range, defined by [first, last), to another range beginning at d_first.

- Copies all elements in the range [first, last). The behavior is undefined if d_first is within the range[first, last). In this case, std::copy_backward may be used instead.
- Only copies the elements for which the predicate pred returns true.

Return value

Output iterator to the element in the destination range, one past the last element copied.

std::search

Searches for the first occurrence of the subsequence of elements [s_first, s_last) in the range [first, last]

- Elements are compared using operator==.
- Elements are compared using the given binary predicate p.

Diff between search and find:

The difference is that std::search searches for a *whole range* of elements within another range, while std::find_first_of searches for a *single* element from a range within another range.

std::reverse

Reverses the order of the elements in the range.

e.g:
```
std::vector<int> v({1,2,3});
    std::reverse(std::begin(v), std::end(v));
    std::cout << v[0] << v[1] << v[2] << '\n'; O/P:321
```

std::unique

Eliminates all but the first element from every consecutive group of equivalent elements from the range [first, last) and returns a past-the-end iterator for the new *logical* end of the range.

- Elements are compared using operator==.
- Elements are compared using the given binary predicate p.

e.g :

```
    // remove duplicate elements (normal use)
    std::vector<int> v{1,2,3,1,2,3,3,4,5,4,5,6,7};
    std::sort(v.begin(), v.end()); // 1 1 2 2 3 3 3 4 4 5 5 6 7
    auto last = std::unique(v.begin(), v.end());
    // v now holds {1 2 3 4 5 6 7 x x x x x x}, where 'x' is indeterminate
    v.erase(last, v.end());
```

O/P:
1 2 3 4 5 6 7

std::is_sorted

Checks if the elements in range [first, last) are sorted in non-descending order.

-Elements are compared using operator<.

- Elements are compared using the given binary comparison function comp.

Return value

true if the elements in the range are sorted in ascending order

e.g:

```
#include <iostream>
#include <algorithm>

int main()
{
```

```
    const int N = 5;
    int digits[N] = {3, 1, 4, 1, 5};

    for (auto i : digits) std::cout << i << ' ';
    std::cout << ": is_sorted: " << std::is_sorted(digits, digits+N) << '\n';

    std::sort(digits, digits+N);

    for (auto i : digits) std::cout << i << ' ';
    std::cout << ": is_sorted: " << std::is_sorted(digits, digits+N) << '\n';
}
```

O/P:

3 1 4 1 5 : is_sorted: 0

1 1 3 4 5 : is_sorted: 1

• std::is_sorted_until

Examines the range [first, last) and finds the largest range beginning at first in which the elements are sorted in ascending order.

Return value

The upper bound of the largest range beginning at first in which the elements are sorted in ascending order. That is, the last iterator it for which range [first, it) is sorted.

e.g:

```
int nums[N] = {3, 1, 4, 1, 5, 9};
int *sorted_end = std::is_sorted_until(nums, nums + N);
sorted_size = std::distance(nums, sorted_end);
```

Other STL notes

• std::lower_bound

Returns an iterator pointing to the first element in the range [first, last) that is *not less* than (i.e. greater or equal to) value.

• std::upper_bound

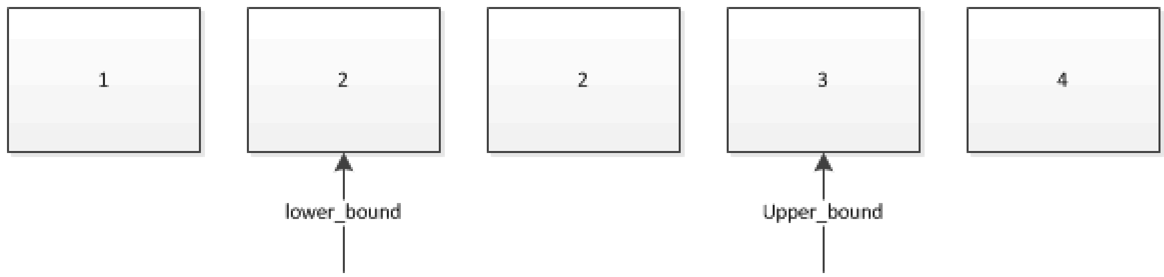
Returns an iterator pointing to the first element in the range [first, last) that is *greater* than value.

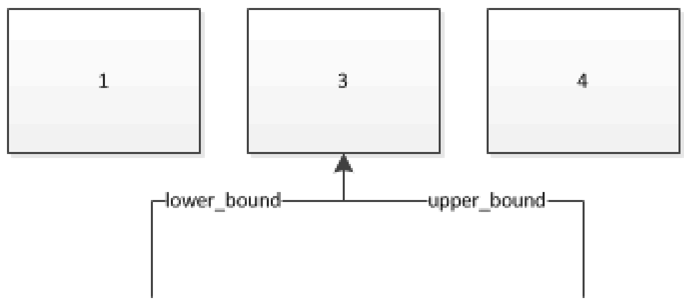
e.g:

```
std::vector<int> data = { 1, 1, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6 };

    auto lower = std::lower_bound(data.begin(), data.end(), 4); O/P: 4
auto upper = std::upper_bound(data.begin(), data.end(), 4); O/P: 4

        lower = std::lower_bound(data.begin(), data.end(), 2); O/P: 2
        upper = std::upper_bound(data.begin(), data.end(), 2); O/P: 3
```





- ## std::binary_search

Checks if an element equivalent to value appears within the range [first, last).
For std::binary_search to succeed, the range [first, last) must be at least partially ordered,

e.g:

```
std::vector<int> haystack {1, 3, 4, 5, 9};
        int needle = 5;
        if (std::binary_search(haystack.begin(), haystack.end(), needle)) {
                std::cout << "Found " << needle << '\n';
          } else {
                std::cout << "no dice!\n";
          }
```

O/P: Found

❖ Key Note of binary search

Binary search needs container to be in sorted order. If it is not it consider as it is sorted and do the searching with start, mid and end point.

```
std::vector<int> vec = { 4,1,8,3,12,2,5 };

if (std::binary_search(vec.begin(), vec.end(), 4)) {
    std::cout << "Found " << '\n';
}
else {
    std::cout << "no dice!\n";
}
```

- ## std::set_difference

Copies the elements from the sorted range [first1, last1) which are not found in the sorted range [first2, last2) to the range beginning at d_first.
The resulting range is also sorted.

Return value

Iterator past the end of the constructed range.

e.g:

```
std::vector<int> v1 {1, 2, 5, 5, 5, 9};
    std::vector<int> v2 {2, 5, 7};
    std::vector<int> diff;

    std::set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
                        std::inserter(diff, diff.begin()));
```

O/P: 1 5 5 9

- ## std::random_shuffle, std::shuffle

```
template< class RandomIt >
void random_shuffle( RandomIt first, RandomIt last );

template< class RandomIt, class URBG >
void shuffle( RandomIt first, RandomIt last, URBG&& g );
```

Reorders the elements in the given range [first, last) such that each possible permutation of those elements has equal probability of appearance.
e.g:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::random_device rd;
    std::mt19937 g(rd());

    std::shuffle(v.begin(), v.end(), g);

std::random_shuffle(v.begin(), v.end(), g);
```

- ## std::transform

```
template< class InputIt, class OutputIt, class UnaryOperation >

OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first,
        UnaryOperation unary_op );
```

std::transform applies the given function to a range and stores the result in another range, beginning at d_first. UnaryOperation is function objects.

Functor is a term that refers to an entity that supports operator () in expressions (with zero or more parameters), i.e. something that syntactically behaves as a function.

A function object that returns a Boolean value is a **predicate**.

A predicate *is a* specific kind *of functor: a functor that evaluates to a boolean value. It is not necessarily a value of bool type, but rather a value of any type with "boolean" semantics. The type should be implicitly convertible to bool though.*

e.g:

```
std::string s("hello");
    std::transform(s.begin(), s.end(), s.begin(),
                   [](unsigned char c) { return std::toupper(c); });
    std::cout << s;
```

- ## std::rotate

Performs a left rotation on a range of elements.

Specifically, std::rotate swaps the elements in the range [first, last) in such a way that the elementn_first becomes the first element of the new range and n_first - 1 becomes the last element.

e.g:

```
std::vector<int> v{2, 4, 2, 0, 5, 10, 7, 3, 7, 1};

 std::sort(v.begin(),v.end());
 // simple rotation to the left
    std::rotate(v.begin(), v.begin() + 1, v.end());

// simple rotation to the right
    std::rotate(v.rbegin(), v.rbegin() + 1, v.rend());
```

O/P:

before sort: 2 4 2 0 5 10 7 3 7 1

after sort: 0 1 2 2 3 4 5 7 7 10
simple rotate left : 1 2 2 3 4 5 7 7 10 0
simple rotate right: 0 1 2 2 3 4 5 7 7 10