STL (Standard Template Library)

- Array
- Vector (Dynamic Array)
- list (Doubly Link List)
- forward_list (Single Link list)
- Set
- multiset
- Map
- Multimap

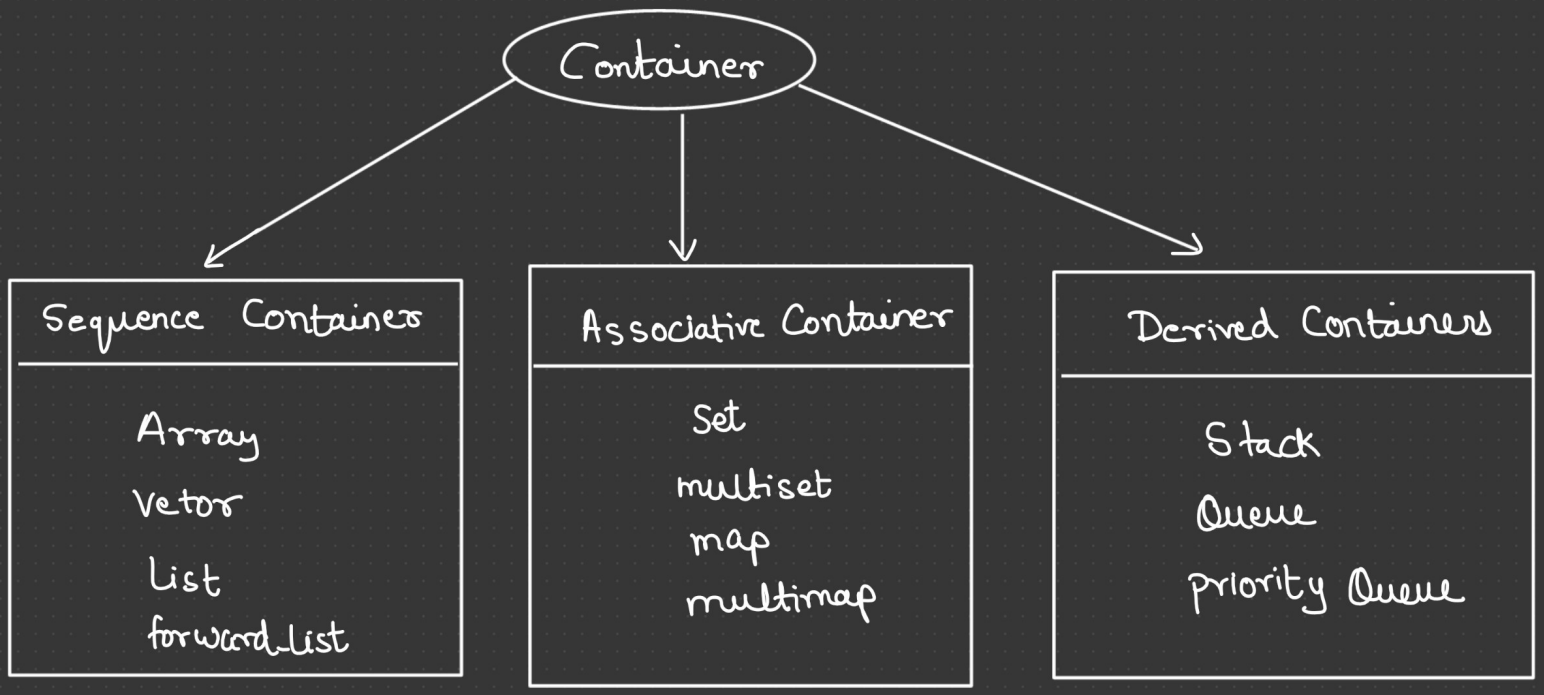
} Ordered / Unordered

- Stack
- Queue
- Priority Queue.

Components of STL :-

- Containers
- Iterators
- Algorithms.

Classification of Containers

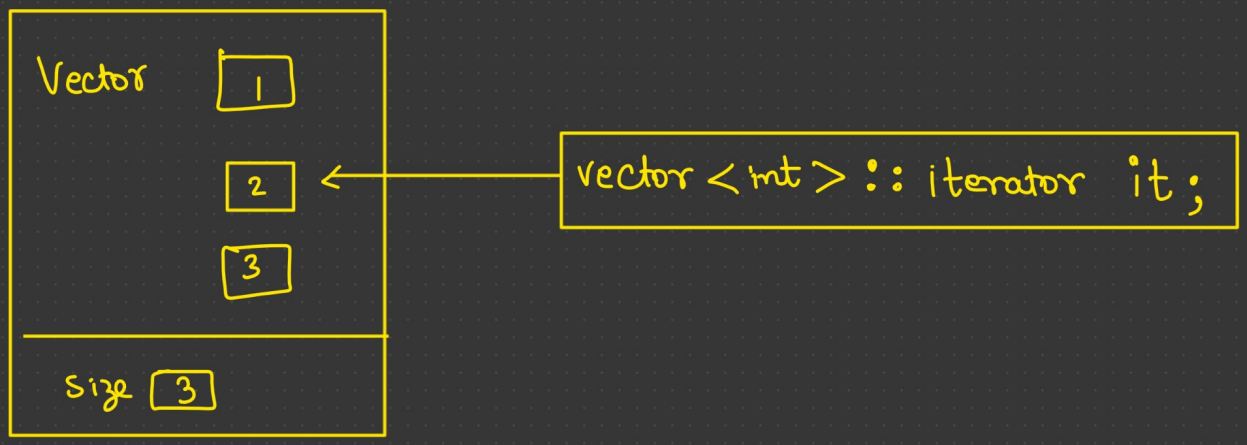


- Each container class contains a set of funtions that can be used to manipulate the contents

Iterator

* Iterators are pointer-like entities used to access the individual elements in a container.

* Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.



Array Class

```
template <class T>
class array
{
    ____
    ____
    ____
    ____
    ___
    ___
};
```

+

- at()
- get()
- operator[]
- front()
- back()
- size()
- max_size()
- swap()

Array <int, 10> a; ≈ int a[10];

Vector class

① `#include <vector>`

② `vector <int> v;`

③ `vector <int> :: iterator it;`

④
```
for( it = v.begin() ; it != v.end() ; it++)
{
    cout << *it << " ";
}
```

Size = 0 (No. of element present in array), capacity = 0 (Size of array)

size = 1 , capacity = 1

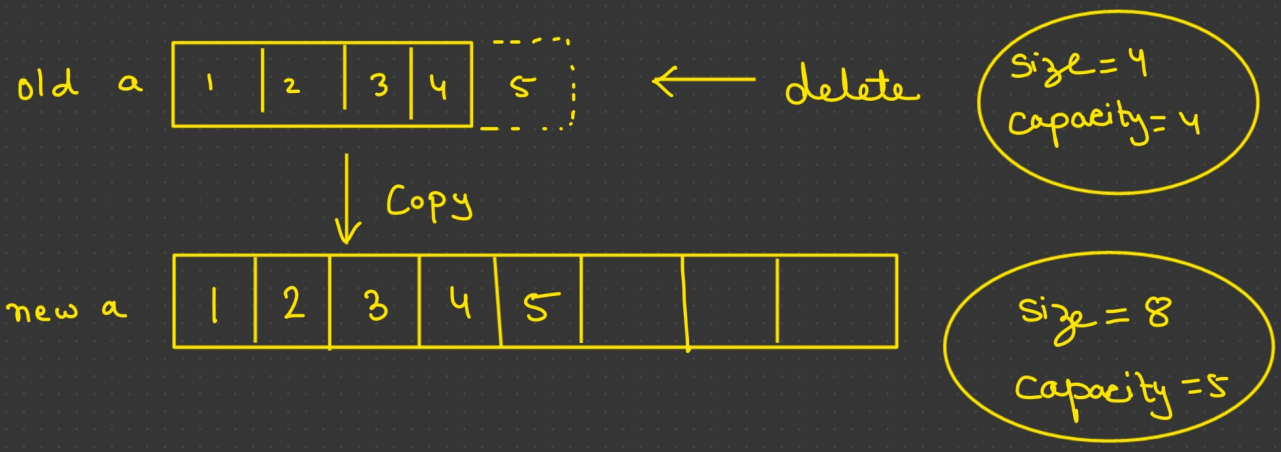
size = 2 , capacity = 2

size = 3 , capacity = 4

size = 4 , capacity = 4

size = 5, 6, 7, 8 , capacity = 8

size = 9 , capacity = 16


old a
1
2
3
4
5
delete
size = 4
capacity = 4
Copy
new a
1
2
3
4
5
size = 8
capacity = 5

List

- It is non-contiguous memory allocation.
- It is same as doubly linked list.
- It is slow in traversal as compared to vector.

- Insertion & Deletion is fast as compared to vector.

```
① #include<list>
② list<int> l1;
③ l1.push_back(5);
④ list<int>::iterator x;
⑤ for(x = l1.begin(); x != l1.end(); x++)
  {
    cout<<*x<<endl;
  }
```

Forward_list

- It is similar to singly linked list.
- We can move in forward direction only.
- Non-contiguous memory allocation.

① `#include <forward_list>`

② `forward_list <int> l1;`

③ `l1.assign({1,2,3,4,5,6});`

`l1.assign(5,10);`

`l1.insert_after(x, {7,8,9});` ← iterator

`l1.push_front(10);`
`l1.pop_front();`
`l1.remove(4);`

} But push_back() & pop_back() is not available.

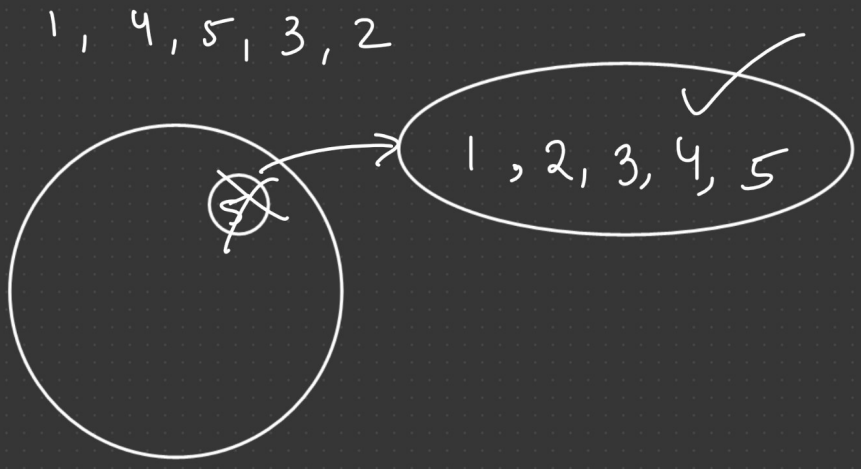
```
forward_list <int> l2;
l2.assign(l1.begin(), l1.end());
```

④ `forward_list<int>::iterator x;`

```
for(x = l1.begin(); x != l1.end(); x++)
{
    cout<<*x;
}
```

Set

- Values are stored in sorted order.



- Each element in set has to be unique.

1. `#include<set>`
2. `set <int> s;` ⟶ Ascending Order

```
s.insert(1);
s.insert(5);
s.insert(3);
```

3. `set <int>::iterator x;`

```
for(x = s.begin(); x != s.end(); x++)
    cout<<*x;
```

For Descending Order :-

```
set < int , greater<int> > s;
      ①          ②
```

Multiset

* Repeated element can be stored in it.

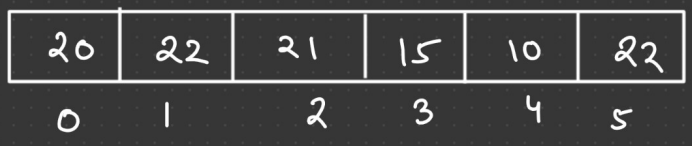
① `#include <set>`

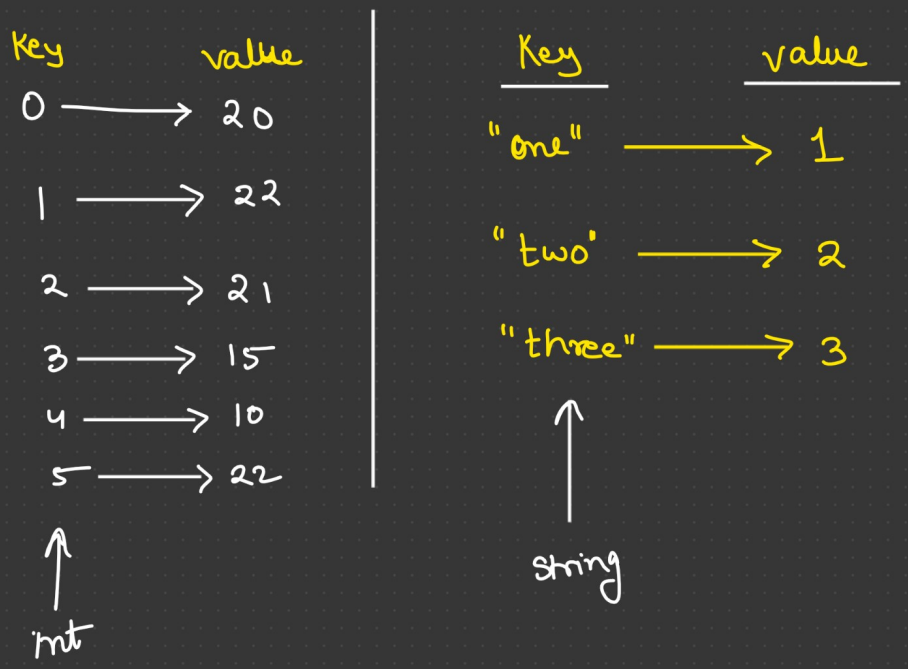
② `multiset <int> s;`

`multiset <int , greater<int>> s1;`

```
s.insert(10);
auto x = s.find(10);
s.erase(10);
```

Map





Map is a associative container in which all the element are stored in mapped fashion.

① #include <map>

② map < int, string > m;

m[key] = value;

m[1] = "prateek";

m[5] = "9555031137";

m[8] = "Bhopal";

1 → "prateek"

5 → "9555031137"

8 → "Bhopal"

- Map can be used for hashing technique.
- Keys of map will always be unique.
- If we insert more than one value at same key then only last value will be present (old value gets overwritten).
- Keys of the map are stored in sorted order.

```
m.erase(4);

auto it = m.find(4);

if( m.count(4) > 0)
{   cout<<"Element found";
}

m.insert( pair<int, string>(5, "Kanpur");
m.insert ({6, "Jabalpur"});

map<int ,string>:: iterator x = m.begin();

for(  ; x != m.end() ; x++)
{
  cout<< x->first <<"   "<< x->second;
}
```

multimap

- Multiple keys can be present.
- operator [ ] is not overloaded.

  ∴ we can't use `m[1] = "prateek";`

  But we can use `m.insert({1,"Prateek"});`

- `m.erase(4);` it will erase all the values mapped to the given key.
- keys will be in sorted ordered.

Unordered set

- Element will not be in sorted manner. It means element can be present in any order.
- Set is implemented as a balanced tree structure.

Ordered set insertion Time Complexity = $O(\log n)$.

Unordered set insertion Time Complexity = $O(1)$.

- Key will be unique.

1. `#include <unordered_set>`
2. `unordered_set <int> s;`

```
s.insert (2);
```