→ unordered_map -

It an associative container in C++ STL, which uses hashing internally. so, insertion, deletion and searching is O(1) operation in unordered_map.

Like map it also store key-value pair but unlike map it doesn't maintain any order.

declaration -

```
unordered_map<int, int> m;
```

- insert() -
  This function is used to insert pair in map.

```
m.insert({5, 7});
```

- [] operator -
  It perform two operation, if key is already there, then it can modify the value, and if key is not present then it will create a key-value pairs with given values.

```
m[key] = value;
```

- begin() -
- end() -
- find(key) -
  Takes key as an argument and returns iterator to that pair if present or iterator to m.end() if not present.

- count() -
  It also takes key as an argument and count occurance of that argument since, no duplicates are in unorded_map only returns 0 or 1.

- size() -
- erase() -

⟹ Algorithum in STL:-

→ Find:-

Find searches linearly in the given range of a container and if element is present it returns iterator to the element, otherwise returns iterator for the given last range (upper_bound element).

Syntax:-

```
auto it = Find (lower_band of container, upper_band
                of container, element);
```

ex. -

```
int main()
{
  vector<int> v = { 10, 20, 7, 30, 12, 9 };
auto it = find(v.begin(), v.end(), 10);
  if(it == v.end())
     cout << "not found";
  else
     cout << "found at" << it - v.begin();
}
```

Output :-

found at 0

```
int main(){
    vector<int> v={10, 12, 7, 13, 27, 19};
auto it= find(v.begin()+1, v.end(), 10);
    if(it ==v.end() )
        cout<<"not found";
    else
        cout<<"found at" << it - v.begin();
return 0;
}
```

Output :-

not found

we can use find to search only in given range, not in whole container as we have done in this example.

NOTE:- These are specific find functions for some containers like string, set, map, unordered set/map etc. For these, it is recommended to use those find functions.

Containers like vector, list, queue, stack, we use normal find() function.

→ lower_bound :-

It uses binary search in container and returns iterator to the first occurance to the element or first element grater than it in a given sorted range.

If element is not there but there is element grater than given element, it returns address to that grater element. And if given element is not in container and no other element is grater than that it returns iterator to the last element in the passed range.

Syntax:-

auto it = lower_bound (initial add, final add, element);

example:-

```
int main()
{
    vector<int> v={5, 7, 7, 7, 8, 9, 10};
auto it=lower_bound(v.begin(), v.end(), 7);
    cout<< *it <<endl;
it= lower_bound(v.begin(), v.end(), 10);
  cout<< *it <<endl;
  it= lower_bound (v.begin(), v.end(), 13);
  cout<< *it <<endl;
return 0;
}
```

output:- 7
11 // 10 is not there but 11 is here which is greater than 10
garbage value / 0 as it returns iterator to the end() which is not part of vector.

We can also use lower_bound to find first occurance of the element if it is there.

ex. -

```
int main()
{
    vector<int> v = {7, 10, 10, 12, 13, 15};
auto it = lower_bound(v.begin(), v.end(), 10);
cout << it - v.begin() << endl;
return 0;
}
```

Output :-

1

If we search for '11' in this programme it will return iterator to the '12' which can cause logical error, in this case we use upper_bound(), if lower_bound <= upper_bound then only element is available, and if lower_bound == upper_bound then element is not present.

We can also use lower_bound to binary search in sorted array.

If given number is not present in array lower_bound can do two things, it returns iterator to the last element of range (in case of whole array arr+n) or returns iterator to an element grater than given element.

ex.-

```
int main()
{
    int arr[] = {7, 8, 8, 8, 9, 11, 13};
int n = sizeof(arr)/sizeof(arr[0]);
auto it = lower_bound(arr, arr+n, 8);
  if((it == (arr+n)) || (*it != 8))
        cout<< "not found";
  else
        cout<< "found";
return 0;
}
```

For random access iterator, time complexity is O(logn)

but non-random access iterator, time complexity is high. So, for non-random access iterator it is not recommended also, for set and map, they have their own lowerbound function which works in logarithmic time, so, we should use them.

But for vector and array, lower-bound is best.

→ Upper_bound :-

It returns iterator to the first greater element to the given element in the sorted range.

Syntax :-

auto it = upper_bound(initial add, end add, ele.);

example:-

```
int main()
{
    vector<int> v={10,11,11,13,17,20};
auto it=upper_bound(v.begin(),v.end(),11);
    cout<<*it<<endl;
auto it=upper_bound(v.begin(),v.begin()+4,30)
    cout<<*it<<endl;
return 0;
}
```

Output:-

13

17 // as there is no element greater than given element, it returns end add.

By using upper_bound and lower_bound we can count occurance of a given number.

no. of occurance = upper_bound - lower bound.

⇒ is_permutation() :-

is_permutation() function checks if, two containers are in permutation or not.

Two containers are in permutation if they contain same elements but order may be different.

It takes 3 parameters,

begin and end iterator/add of one container and begin iter/add of second container.

Syntax :-

is_permutation(v1.begin(), v1.end(), v2.begin());

NOTE :- iterators pass here should be forward iterator or higher.

also, for associative container like Ordered set / unordered map it checks only for key.

It also work in case of multiple occurance of same element.

ex.-

```
int main()
{
    vector<int> v1 = { 7, 10, 30, 27, 9 };
    vector<int> v2 = { 10, 7, 27, 9, 30 };

    if (is_permutation(v1.begin(), v1.end(), v2.begin()))
        cout << "Yes" << endl;
    else
        cout << "No" << endl;

    return 0;
}
```

output :-

Yes

→ max_element and min_element :-

max_element and min_element function returns iterator to the maximum and minimum element in a container respectively.

both takes begin and end add. as argument.

Syntax :-

```
auto it1 = max_element(v.begin(), v.end());
auto it2 = min_element(v.begin(), v.end());
```

ex :-

```
int main(){
    vector<int> v = {10, 90, 7, 30, 47, 23};
auto it1 = max_element(v.begin(), v.end());
auto it2 = min_element(v.begin(), v.end());
cout<< *it1 << " " << *it2 ;
return 0;
}
```

Output :- 90 2

```
int main()
{
    int arr[] = {5, 8, 7, 2, 1, 12};
    cout<<*max_element(arr, arr+6)<<" ";
    cout<<*min_element(arr, arr+6)<<" ";
}
```

Output :- 12 1

We can use max_element and min element to find max and min element according to user defined criteria.

For example, we have a vector of 2-D points, and we have to find two points with max and min x-coordi. On this condition we use user defined campare function-

ex-

```
#include<vector>
#include<iostream>
#include<algorithm>
using namespace std;
```

```
struct point{
    int x,y;
    point(int i, int j){x=i; y=j;}
};

bool mycomp(point p1, point p2){
    return p1.x < p2.x;
}

int main()
{
    vector<point> v = {{5,4}, {2,300}, {90,10}};
    auto it = max_element(v.begin(), v.end(), mycomp);
    cout<<(*it).x<<" "<<(it->y)<<endl;
    it = min_element(v.begin(), v.end(), mycomp);
    cout<<it->x<<" "<<(*it).y<<endl;
    return 0;
}
```

Output 90 10
2 300

Time complexity - O(n)

In mycompare function

$$P_1.x < P_2.x$$

here,

'<' implies natural order

and,

'>' implies reverse of natural order.

Instead of using structure to store point we can use pairs as well.

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<utility>

using namespace std;

bool mycomp(pair<int,int> P1, pair<int,int>P2)
{
    return P1.first < P2.first;
}

int main(){
  vector<pair<int,int>> v={{7,5},{30,2},{2,25}};
```

```
auto it = max_element(v.begin(), v.end(), mycomp);
cout << it -> first << " " << it -> second << endl;
it = min_element(v.begin(), v.end(), mycomp);
cout << it -> first << " " << it -> second << endl;
return 0;
}
```

Output :-

30 2
2 25

→ Count :-

Count() function gives occurance of the given element in a container in linear time.

Syntax :-

```
int it = count(begin add, end add, element);
```

ex. -

```
int main()
{
    vector<int> v = {5, 7, 10, 5, 13, 7, 9, 13, 103};
    cout << count(v.begin(), v.end(), 10) << endl;
    cout << count(v.begin(), v.end(), 20);
}
```

Output:- 2
0

count() function can be used for containers like vector, queue, string, array, stack etc., but containers like map and set, it is recommended to use their count function as this count function has T.C. of O(n) but their count function works better.

→ binary search():-

binary search() return True/False according to, if given number present in container or not. For this to work, container must be sorted.

Syntax:-

binary_search(v.begin(), v.end(), x);

Binary search can be used to search container using user defined condition.

Syntax:-

binary_search(v.begin(), v.end(), x, compare_fun);

Ex. -

```
int main(){
  vector<int> v={10,20,30,40,50};

if(binary_search(v.begin(),v.end(),20)==true)
      cout<<"found";

  else
      cout<<"not found";

return 0;
}
```

Output :-

found,

Searching a container of points of 2-D coordinate using x-coordinate.

```
bool mycomp(pair<int,int> p1, pair<int,int> p2)
{
   return p1.first < p2.first;
}

int main()
{  vector<pair<int,int>> v={{2,30},{7,7},{3,30}};
   sort(v.begin(), v.end());
if(binary_search(v.begin(),v.end(),{2,30},mycomp))
      cout<<"found";
```

```
else
    cout<<"not found";
return 0;
}
```

Output:-
found

If you are using used defined function to search in binary search, we solely used that function. In previous example according to that function, binary search only checks for first element of pair not second one. So, if you check for {2,29}, instead of {2,30} it will show found.

Instead of pair in previous example we can use structure too.

```
Structure point{
    int x, y;
    point(int i, int j){x=i; y=j;}
};

bool myComp(point p1, point p2){
    return p1.x < p2.x;
}
```

```
int main()
{
    vector<point> v={{3,7},{0,5},{22,5},{22,
    sort(v.begin(), v.end(), mycomp);
    Point P(22,5);
if(binary_search(v.begin(), v.end(), P, mycomp))
        cout<<"found";
    else
        cout<<" not found";
    return 0;
}
```

Output :- found;

For random access iterator T.C. is O(logn), for non-random access iterator T.C. is O(n).

→ rotate() :-

This function used to rearrange the element of containers such that given element from middle became first element and element that were left of it put at right side of containers

ex. –

vector<int> v = {5, 7, 12, 2, 9, 13, 20}

if given element is 2 then vector become

{2, 9, 13, 20, 5, 7, 12}

It takes 3 argument begin add, add of element and end add.

Syntax:-

rotate(begin add, add of given ele., end add);

ex.-

```
int main()
{
    vector<int> v={2,30,8,17,47,9,13};
    rotate(v.begin(), v.begin()+3, v.end());
    for(auto x:v)
        cout<<x<<" ";
    return 0;
}
```

output:- 17 47 9 13 2 30 8