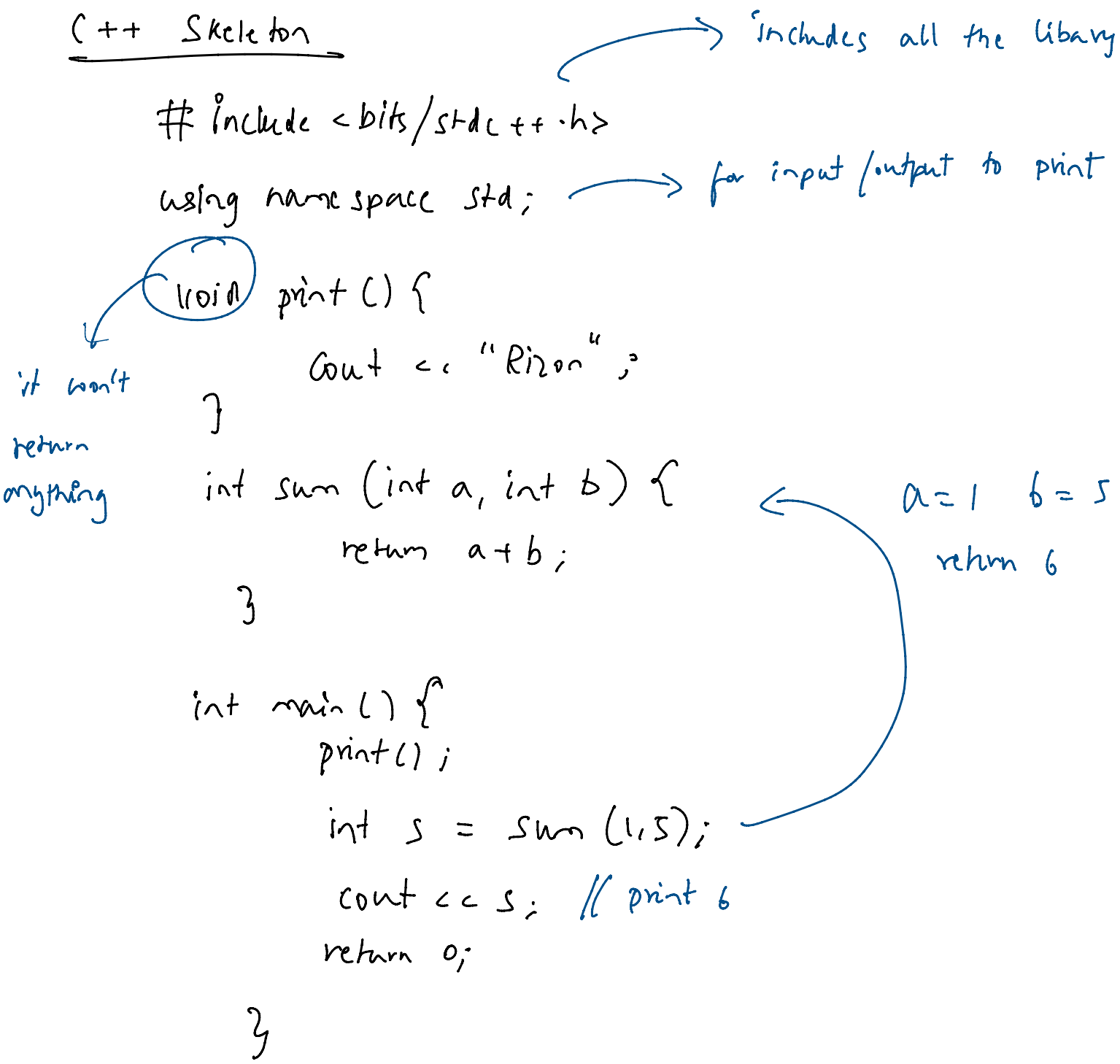
1. C++ Skeleton

C++ Skeleton



2. Types of STL

Types of STL:

* Algorithm
* Container ⟶ we will learn this first
* Function
* Iterators

```
void pairs () {        → in place of int there can be other datatype depends on its requirement.
    pair<int, int> p = {1, 3}; // to store two integer

    cout << p.first << " " << p.second;
```

1 3

```
// to store more than 2 pair
    pair<int, pair<int, int>> p = {1, {3, 4}}  // 3 variables

    cout << p.first << " " << p.second.second << " " << p.second.first;
```

1 first second 3 4

```
// to declare pairs of arrays
    pair<int, int> arr[] = { {1, 2}, {2, 5}, {5, 1} }
                             0       1       2  → index

    cout << arr[1].second; // 5
```

4. Vectors

04 June 2022 06:44 PM

Vector is a container which is dynamic in nature, you can always increase the size.

```
int a[5] = {-, -, -, -, -}
           0  1  2  3  4
```

↑ You can't modify this array

```
void explainVector() {

    vector<int> v; // declaration - empty container {}

    v.push_back(1); // => {1}

    v.emplace_back(2); // similary to push back it dynamically increase it size.
                       // {1, 2}

    vector<pair<int, int>> vec; // vector of pair

    v.push_back({1,2}); // {1,2}

    v.emplace_back(1,2); // {1,2}

    // Container of size 5 of element 100
    vector<int> v(5, 100); // {100, 100, 100, 100, 100}
                           //  0    1    2    3    4
                           // ↑ Size

    vector<int> v(5); // {0, 0, 0, 0, 0} Garbage Value
```

emplace_back is faster than push back.

```
vector <int> v1 (5, 20);   // {20, 20, 20, 20, 20}

// to copy into another vector
vector <int> v2 (v1);      // {20, 20, 20, 20, 20}
```

How to access element in a vector ?

v → {20, 10, 15, 5, 7}
0 1 2 3 4

1st way

v[1] ⇒ 10

v[3] ⇒ 5

2nd way → iterator

Syntax:

```
vector <int> :: iterator it = v.begin();
```

- vector <int> : data type
- it : variable
- v.begin() : points directly on the memory

```
it tt;

cout << *(it) << " "; //10

it = it + 2; // shift it by 2 position

cout << *(it) << " "; // 5
```

v → {20, 10, 15, 5, 7}
0 1 2 3 4
begin

printing the memory address, not the element.

in order to access the element in C++ we use *.

*(v.begin()) = 20

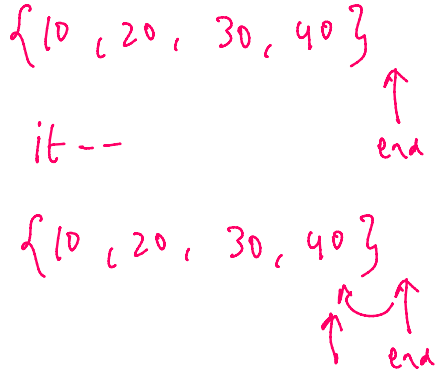
* (v.begin()) = 20

it ++ // it move to next memory

* (v.begin()) = 10

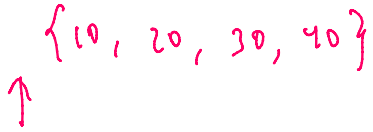
END

Vector <int> :: iterator it = v.end()



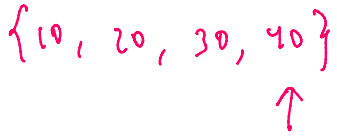
REND : (Reverse END)

Vector <int> :: iterator it = v.rend()

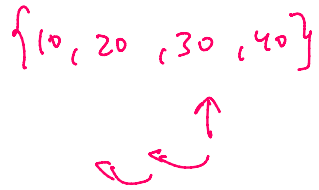


RBEGIN : (Reverse begin)

Vector <int> :: iterator it = v.rbegin()

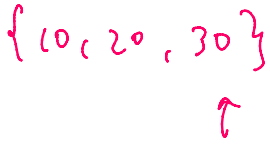


it++ ⇒

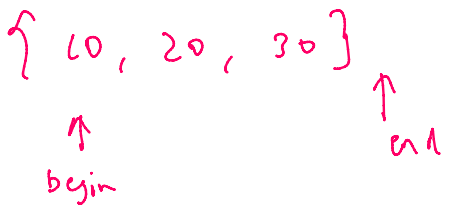


↳ in reverse way it move.

cout << v.back() << " "; // 30



// print using for loop




```
for (vector<int>::iterator it = V.begin() != V.end(); it++){
        cout << *(it) << " ";
}
```

Output ⇒ 10 20 30

// More short way to use

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

Output ⇒ 10 20 30

// More More Shorted way

```
for (auto it : V) {
    cout << it << " ";
}
```

Output ⇒ 10 20 30

Deletion in Vector:

{10, 20, 12, 23}

```
V. erase (V. begin() + 1);   // { 10, 12, 23 }
```

{ 10, 20, 12, 23, 35 }

```
V. erase ( V.begin() + 2, V.begin() + 4);   // { 10, 20, 35 } [start, end)
```

↑ not included

Insertation in vector:

```
vector < int > V (2, 100);   // {100, 100}
V. insert (V. begin(), 300);   // { 300, 100, 100}
V. insert ( V. begin() + 1, 2, 10)   // { 300, 10, 10, 100, 100}

vector <int> copy (2, 50);   //{50, 50}
V. insert (V. begin(), copy. begin(), copy. end());   // { 50, 50, 300, 10,
                                                          10, 100, 100 }
```

{10, 20}

```
V. size ();   // 2 (How many element there in vector)
```

{10, 20}

```
V. pop_back();   // { 10 }
```

V1 → {10, 20} V2 → {30, 40}

v1.swap(v2); v1 → {30, 40} v2 → {10, 20}

v.clear(); // erase the entire vector

cout << v.empty(); // {1} → False {} → true

5. List

```
list<int> ls;

ls.push_back(2);      // {2}

ls.emplace_back(4);   // {2,4}

ls.push_front(5);     // {5,2,4}

ls.emplace_front(1);  // {2,4}

// rest function same as vector
```

```
Void explainDeque {

    deque <int> dq;

    dq.push_back (1);          // {1}

    dq.emplace_back (2);       // {1, 2}

    dq.push_front (4);         // {4, 1, 2}

    dq.emplace_front (3);      // {3, 4, 1, 2}

    dq.pop_back();             // {3, 4, 1}

    dq.pop_front();            // {4, 1}

    dq.back();

    dq.front();

}
```

Stack ⇒ LIFO (Last In Front Out)

```
Void explainStack() {
    stack<int> st;
    st.push(1);      // {1}
    st.push(2);      // {2,1}
    st.push(3);      // {3,2,1}
    st.push(3);      // {3,3,2,1}
    st.emplace(5);   // {5,3,3,2,1}
    cout << st.top(); // print 5 "* * st[2] is invalid **"
    st.pop()         {3,3,2,1}
    st.top()         // 3
    st.size()        // 4
    st.empty()       // False
    Stack<int> st1, st2;
    st1.swap(st2)
}
```

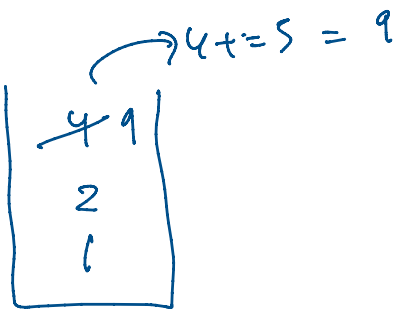
} T.C ⇒ O(1)

8. Queue

08 June 2022 09:53 AM

FIFO ⇒ First In First Out

```
void explainQueue(){

    queue<int> q;

    q.push(1);   // {1}
    q.push(2);   // {1,2}
    q.push(4);   // {1,2,4}

    q.back() t=5

    cout << q.back(); //print 9

    // {1,2,9}
    cout << q.front(); //print 1

    q.pop(); // {2,9}

    cout << q.front(); //print 2

}
```



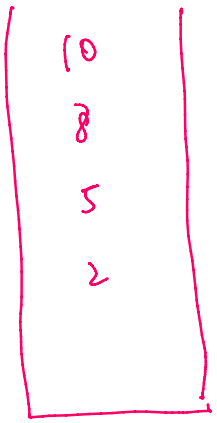
9. Priority Queue

08 June 2022 10:00 AM

(Max Heap)

Priority Queue : Largest element stays at the top.

```
void expPQ() {
    priority_queue<int> pq;
    pq.push(5);      // 5
    pq.push(2);      // {5, 2}
    pq.push(8);      // {8, 5, 2}
    pq.emplace(10);  // {10, 8, 5, 2}

    cout << pq.top();  // print 10
    pq.pop();          // {8, 5, 2}
    cout << pq.top();  // print 8
```



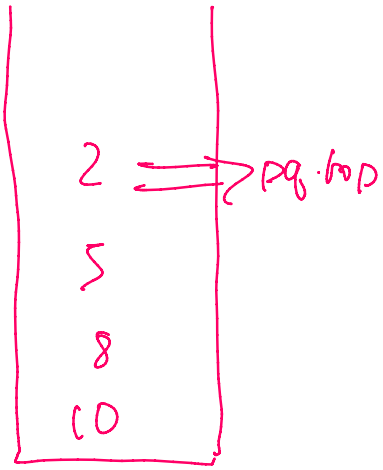
(Min Heap)

Minimum Heap Syntax: For smallest element

```
priority_queue<int, vector<int>, greater<int>> pq;
pq.push(5); // 5
pq.push(2); // 2, 5
```

→ You need to add this for smallest element

```
pq.push(8); // {2, 5, 8}
pq.emplace(10); {2, 5, 8, 10}
cout << pq.top(); //prints 2
```



T.C

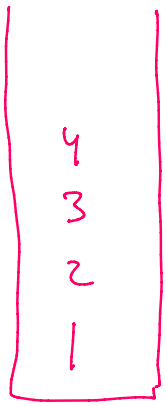
push → log n

top → O(1)

pop → log n

Set : It store everything in sorted order and store unique elements.

```
void explainSet() {
    Set<int> St;
    St.insert(1);    // {1}
    St.emplace(2);   // {1,2}
    St.insert(2);    // {1,2}
    St.insert(4);    // {1,2,4}
    St.insert(3);    // {1,2,3,4}
```



// functionality of insert in vector can also be used, that only increase efficiency.

↓

// {1, 2, 3, 4, 5}

```
auto it = St.find(3);   // return an iterator which points to the '3'.
```

↓

// {1, 2, 3, 4, 5}

```
auto it = st.find(6); // if the element is not in the set it
                      will always return st.end() i.e. the
                      iterator that points to right after the
                      end.

// {1, 4, 5}

st.erase(5); // erase 5 // take logarithmic time

int cnt = st.count(1). // '1' occurence if it does'nt
                       have then '0'.

auto it = st.find(3);

st.erase(it); // it take constant time.

// {1, 2, 3, 4, 5}

auto it1 = st.find(2);

auto it2 = st.find(4);

st.erase(it1, it2); // after erase {1,4,5} [first, last)

// lower_bound() and upperbound() iworks in the same
way as in vector it does.

// This the syntax

auto it = st.lower_bound(2);

auto it = st.upper_bound(3);

}
```

In set everything happen in $(\log N)$

Binary Search

Q: Check if X exists in sorted array or not?

A[] = {1, 4, 5, 8, 9}

a a+n

```
bool res = binary_search (a, a+n, 3);
```

check '3' exist or not

o/p ⇒ false

```
bool res = binary_search (a, a+n, 4);
```

o/p ⇒ True

lower_bound functions

a[] = {1, 4, 5, 6, 9, 9}

```
int ind = lower_bound (a, a+n, 4) - a;
```

to get index we do '- a'

⇒ 1 (Index)

```
int ind = lower_bound (a, a+n, 7) - a;
```

⇒ 4 (Index)

```
int ind = lower_bound (a, a+n, 10) - a;
```

⇒ 6 (Index)

lower bound STL returns the first occurence of the element if it occurs and if it doesn't occurs then it return the iterator pointing to the element which is the immediate next greater of the given element.

Syntax:

Starting value

```
int ind = lower_bound(a.begin(), a.end(), a) - a.begin();
```

Upper Bound:

```
int ind = upper_bound(a.begin(), a.end(), a) - a.begin();
```

a[] = {1, 4, 5, 6, 9, 9}

```
int ind = upper_bound(a, a+n, 4) - a;
int ind = upper_bound(a, a+n, 7) - a;
int ind = upper_bound(a, a+n, 10) - a;
```

Q: Find the first occurrence of a X in a sorted array. If it doesn't exits, print -1.

X=4 (1)

A[] = {1, 4, 4, 4, 4, 9, 9, 10, 11}

A[7] = {1, 4, 4, 4, 4, 9, 9, 10, 11}

X=2 (-1) X=12 (-1)

```
int ind = lower_bound(a, a+n, X) - a;
if(ind != n && a[ind] == X) cout << ind;
else cout << -1;
```

X = 4 // 1

X = 2 // -1

X = 12 // -1

Q: Find the last occurrence of a X in a sorted array. If it doesn't exists, print -1.

A[7] = {1, 4, 4, 4, 4, 9, 9, 10, 11}

```
int ind = upper_bound(a, a+n, X) - a;
ind--;
if(ind >= 0 && a[ind] == X) cout << ind;
else cout << -1;
```

edge case

X = 4 // 5

X = 2 // -1

X = 0 // -1

Q: Find the largest nos. smaller than X in a sorted array. If it doesn't exists, print -1.

A[7] = {1, 4, 4, 4, 4, 9, 9, 10, 11}

```
int ind = lower_bound(a, a+n, X) - a;
ind--;
if(ind >= 0) cout << a[ind]
else cout << -1;
```

X = 4 // 1

X = 2 // 1

X = 1 // -1

```
void explainMultiSet() {
    // Everything is same as set only stores duplicates elements.
    multiset<int> ms;
    ms.insert(1); // {1}
    ms.insert(1); // {1,1}
    ms.insert(1); // {1,1,1}
    ms.erase(1); // all 1's erased.
    int cnt = ms.count(1);

    // only a single one erased.
    ms.erase(ms.find(1));
    ms.erase(ms.find(1), ms.find(1)+2);
}
```

```
Void explainUSet () {
    unordered_set <int> st;   // Same as set only one
}                             thing is NOT 'sorted'
                              but unique.
```

```
// lower_bound and upper_bound function
// does not works, rest all functions are same
// as above,  it does not stores in any
// particular order it has a better complexity
// than set in most cases, except some when collision happens
```

T.C $\Rightarrow O(1)$

Worst case $\Rightarrow O(N)$

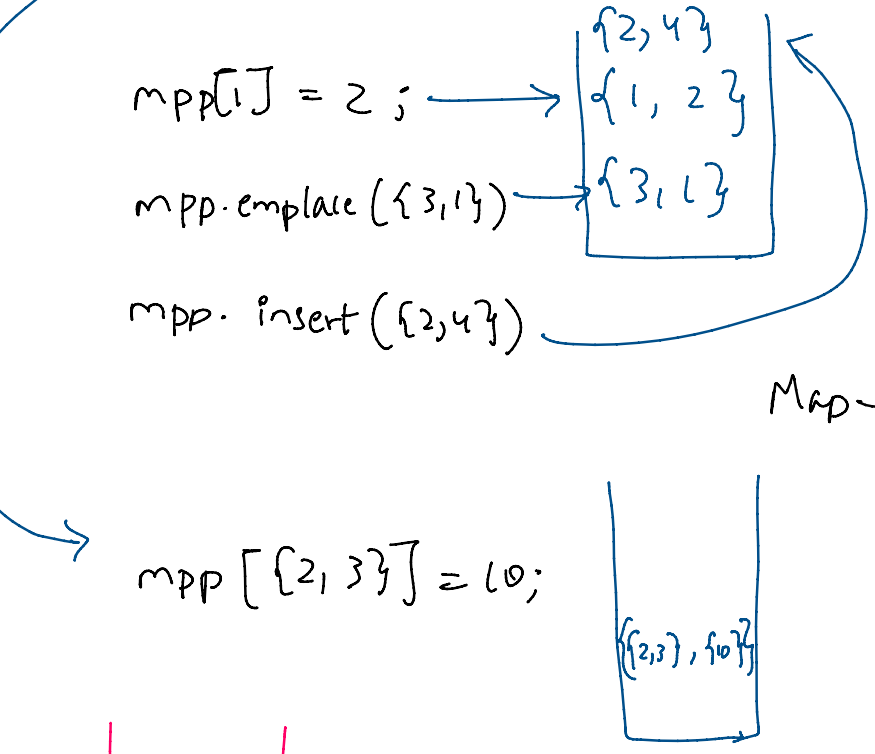
13. Map

16 June 2022 05:10 PM

{ key, value >

* Map is key, value pairs.
* The key are unique.
* The value can have duplicates.

```
void explainMap() {

    map<int, int> mpp;
    map<int, pair<int,int>> mpp;   // key is one int, value is two int
    map<pair<int, int>, int> mpp;  // key is two int, value is one int

    mpp[1] = 2;
    mpp.emplace({3,1});
    mpp.insert({2,4});

    mpp[{2,3}] = 10;
```

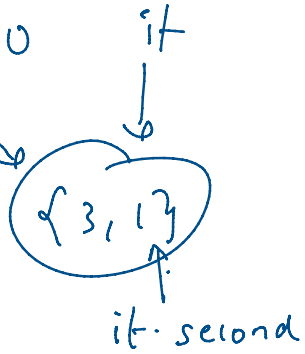


Map → Store Unique key in stored order of key.

{ {1, 2}, {2, 4}, {3, 1} }

{ {1, 2}, {2, 4}, {3, 1} }

```
for (auto it : mpp) {
        cout << it.first << "    " it.second << endl; // 1 2
}                                                    // 2 4
                                                     // 3 1

cout << mpp[1]; // 2

cout << mpp[5]; // null or 0

auto it = mpp.find(3);
cout << *(it).second // 1
```




```
auto it = mpp.find(5); //mp.end()

// This is syntax
auto it = mpp.lower_bound(2);
auto it = mpp.upper_bound(3);
}
```

14. MultiMap and Unordered Map

16 June 2022 05:23 PM

MultiMap similar to map only thing differ is you can store duplicates key everything in sorted order.

{1, 2} {1, 3}

Unordered Map : contains unique keys but not in sorted order.

Map: T.C $\Rightarrow$ $O(\log n)$

Unordered Map : T.C $\Rightarrow$ $O(1)$

15. Extra

16 June 2022 05:28 PM

```cpp
void explainExtra() {


    sort(a, a + n);
    sort(v.begin(), v.end());

    sort(a+2, a+4);

    sort(a, a+n, greater<int>);

    pair<int,int> a[] = {{1,2}, {2, 1}, {4, 1}};

    // sort it according to second element
    // if second element is same, then sort
    // it according to first element but in descending

    sort(a, a+n ,comp);

    // {4,1}, {2, 1}, {1, 2}};


    int num = 7;
    int cnt = __builtin_popcount();

    long long num = 165786578687;
    int cnt = __builtin_popcountll();


    string s = "123";

    do {
        cout << s << endl;
    } while(next_permutation(s.begin(), s.end()));
```

sort(a+2, a+4)

{1, 5, 4, 3}

descending order ⇒ {1, 5, 3, 4}

only this part will be sorted

Comparator:

Own way of sorting.

```cpp
bool comp(pair<int,int> p1, pair<int,int> p2) {
    if(p1.second < p2.second) return true;
    if(p1.second > p2.second) return false;
    // they are same

    if(p1.first > p2.first) return true;
    return false;
}
```

{ p1 } { p2 }