STL Overview

Memo

```
#define PI 3.14159265358979323846

/* int dx[] = {1,-1,0,0} , dy[] = {0,0,1,-1}; */ // 4 Direction
/* int dx[] = {1,-1,0,0,1,1,-1,-1} , dy[] = {0,0,1,-1,1,-1,1,-1}; */ // 8 Direction
/* int dx[] = {1,-1,1,-1,2,2,-2,-2} , dy[] = {2,2,-2,-2,1,-1,1,-1}; */ // Knight Direction
/* int dx[] = {2,-2,1,1,-1,-1} , dy[] = {0,0,1,-1,1,-1}; */ // Hexagonal Direction
```

2d Vector

```
vector<vector<int>> v2(2, vector<int> (4, -10));
```

v2 is a vector with size 2. Inside this vector it has another vector of `int`, let's call it `vin`. the size of `vin` is 4 and each elements of `vin` is initialized to the value -10. The bottom line, v2 is a 2×4 2D-matrix, 2 rows and 4 columns with each elements assigned the value -10

Custom Sort Structure:

```
// Created by Dipta Das on 8/07/17.
struct Info {
    int id, marks, weight;
    void assignValue (int id, int marks, int weight) {
        this->id = id; this->marks = marks; this->weight = weight;
    }
    bool operator < (const Info a) const {
        return this->weight < a.weight;
    }
};
bool compareInfo (Info i1, Info i2) {
    return i1.marks < i2.marks;
}
int main ()
{
//    freopen("input", "r", stdin);
    Info info[3];
    info[0].assignValue(2, 20, 100);
    info[1].assignValue(3, 50, 30);
    info[2].assignValue(1, 40, 50);

    // Use either of the follwing two ways.
    // sort(info, info+3, compareInfo);
    // sort(info, info+3);
}
```

Custom Sort Vector:

```
bool compareUsingMapId (pair <int, pair<int, int> > m1, pair <int, pair<int, int> > m2) {
    // the parameters are actually of type of the elements inside the vector we want to sort
    return m1.first < m2.first; // id vs id
//    return m1.second.first < m2.second.first; // marks vs marks
//    return m1.second.second < m2.second.second; // total vs total
}
int main ()
{
    vector < pair <int, pair <int, int> > > mymap;
    mymap.push_back(make_pair(2, make_pair(20, 100)));
    mymap.push_back(make_pair(1, make_pair(40, 50)));
    mymap.push_back(make_pair(3, make_pair(50, 30)));
```

```
    sort(mymap.begin(), mymap.end(), compareUsingMapId);

    return 0;
}
```

Map:

```
int main()
{
    map <string, int> mymap;
    map <string, int>::iterator it;
    mymap["Abul"] = 43;
    mymap["Dabul"] = 10;
    mymap.insert(make_pair("Babul", 20));
/*  now the map is sorted itself based on the keys
    Abul    43
    Babul   20
    Dabul   10 */
    for(it = mymap.begin(); it != mymap.end(); it++) {
        cout << it->first << " " << it->second << endl;
    }
}
```

Priority Queue:

```
int main()
{
    // queue and priority_queue takes dublicate elemnts
    priority_queue <int> myq;
    vector <int>::iterator it;
    myq.push(50); myq.push(20); myq.push(100);
    myq.push(100);
    // remember your can't iterate through queues, so you'll have to use while loop
    priority_queue <string> myq2;
    myq2.push("Abullll");
    myq2.push("Dabul");
    myq2.push("Babul");
    /*
    remember::::::::
    priority_queue sorts string in the opposite direction of the
    lexicographical order.
    Dabul
    Babul
    Abullllll
    */
}
```

Queue:

```
int main()
{
    queue <int> myq;
    // queue and priority_queue takes dublicate elemnts
    cout << "At the back: " << myq.back() << endl;
    cout << "queue.size() = " << myq.size() << endl;
    cout << "queue.empty() = " << myq.empty() << endl;
    cout << "At the front: " << myq.front() << endl;
}
```

Set:

```
int main()
{
    set <string> names;
    set <string>::iterator it1;
    /*
        set sorts the elements in lexicographical order
        No duplicate elements are allowed! if the element is identical to one of the existing element of t
    */
    for (it1 = names.begin(); it1 != names.end(); it1++) {
        cout << *it1 << endl;
    }
    cout << endl;
    return 0;
}
```

```
int main()
{
    set <int> myset;
    myset.insert(3); // 3
    myset.insert(1); // 1 3
    myset.insert(7); // 1 3 7

    set<int>::iterator it;
    it = myset.find(7); // 'it' now points to 7, it's very fast: O(log(n))

    /* one very important thing: the value of elements in set/multi-set can not be modified. like, *it = 1

    /*
    the insert() functions returns a pair of values. [shown below]
    the second value returns a boolen to indicate whether the insertion is
    successful or not.
    the first element of the pair is an iterator.
    if the insertion is completed then the iterator points to the inserted
    element. if the insertion is not completed the iterator points to the
    duplicate value already existing in the set.
    */
    pair <set<int>::iterator, bool> ret;
    ret = myset.insert(3); // no element inserted
    if (ret.second == false) {
        it = ret.first; // therefore, 'it' now points to 3 in myset
    }

    myset.insert(it, 9); // 1 3 7 9
    /*
    then why use iterator in insert() function?
    well it's because iterator is given as a hint as to where to position 9 if I can give a good hint then
    */

    myset.erase(it); // removes 3 from the set // 1 7 9
    myset.erase(7);  // removes 7 from the set /// 1 9
    const bool is_in_myset = s.find(10) != s.end();
    // don't try to use distance function with the iterator s.find() returns, or you'll get segmentation f
}
```

Vector:

```
vector<int> v = { 1, 2, 3, 4, 5 };
v.back(); // returns the last element
v.erase(v.end()-1); // remove last element
v.erase(v.begin());
```