Stl.cpp

```cpp
////////////**********  STL  **********///////////
/////// Stanndard Template Library in C++: ////////

#include <bits/stdc++.h>
using namespace std;

int main() {
    /// C++ STL mainly divided into 4 parts.
    /*
    1. Algorithms
    2. Containers.
    3. Functions.
    4. Iterators.
    */

    /*Now, first we are going to learn about containers.
    And to learn about containerm first we have to learn about pairs. Lets see what are Pairs. Pair is a part of utility library:

    lets say, we want to store couple of integers. So, here we can use pairs.
    */



   /////////////////     -:PAIR:-     /////////////////

    pair<int, int> p = {1, 3}; // declaration of pair , here in the place of int, the datatype can be anything double, float, char, string etc.
    cout << p.first << " " << p.second << endl; // output

    //// Now, for multiple variables we can use the nested  property of pair.
    pair<int, pair<int, int>> nestedP = {1, {4, 6}}; // declaration of nested pair
    cout << nestedP.first << " " << nestedP.second.first << " " << nestedP.second.second << endl; // ouput of nested_pair.

    /* Till now, we have learnt about integer array or character array or array of any particular datatypes. But now, we also can declare/use an array of pair.*/

    pair<int, int> arr[] = {{1, 2}, {3, 4}, {5, 6}, {7, 8}}; // array pair declaration
    cout << arr[1].second << endl;




    /* SO, now the first container we are going to learn about Vectors. */
    ////////////////      -:VECTORS:-    /////////////////

    /* Vector container is dynamic in nature. So, whenever we want to increase the size of the vector we can. But, in array we can't do it as it has constant size.
    -> If there is a requirement, where we don't know the size of any particular data structure, that's the perfect position where we can use Vector.
    */

    // Declaration of vector

```

```cpp
    vector <int> v;               // it creates an empty container.
    v.push_back(1);               // {1}
    v.emplace_back(2);            // {1, 2}
    /// Note: genrally, emplace_back() is faster than push_back().

    // Declaration of a vector of pair datatype.

    vector <pair<int, int>> vec;
    vec.push_back({1,5});
    vec.emplace_back(6,7);
    // *-> here,in emplce_back() we don't have to give the curly braces as emplace_back automatically assumes it as a pair. That's how emplace_back() is different from push_back().

    //// We can declare a vector of any particular size and with particular elements.
    vector <int> v1(5, 100);        // {100, 100, 100, 100, 100}

    /// If, we want to declare without 100, we also can do that but in that case there will be 0 or some garbage value.

    vector <int> v2(5);             // {0, 0, 0, 0, 0} <-here 0 can be garbage value.

    // Lets know how we can copy one vector container to other.
    vector <int> v3(5, 20);         // v3[] = {20, 20, 20, 20, 20}
    vector <int> v4(v3);            // v4[] = {20, 20, 20, 20, 20} <- remember, it is an another container with same value.
    // here, after all these declaration also we can use push_back() or emplace_back()



    //// Lets see how we can access elements in a Vector.
    /// here we are going to learn about Iterators.

    ////////////////////   -:ITERATORS:-    ////////////////////

    vector <int> v5 = {10, 20, 20, 14, 52, 62};
    /// Declaration of Iterator:
    vector<int>::iterator it = v5.begin(); // it is pointing to first address  of v5[] vector.

    /*
    v.begin() -> address of memory.
    *(v.begin()) -> value at that address.
    */

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

    it = it + 2; // shifted by 2 position.
    cout << *(it) << " "; // 14

    // We also have iterators apart from begin. // v.end(),  v.rbegin() -> reverse begin,   v.rend() -> reverse end

    vector<int>::iterator it1 = v5.end();
    // Note: It will not point to the address of the last element.
    // -> It will point to the address which is right after the address of the last element.
    // -> And, then if we do "it1 --" then it will point to the address of the last element.

```

```
104     cout << v5.back() << endl; // the element in the last index. // 62
105
106     ///// Lets see how we can print a vector using for loop.
107     for (vector<int>::iterator it2 = v5.begin(); it2 != v5.end(); it2 ++) {
108         cout << *(it2) << " ";
109     } //output: 10 20 20 14 52 62
110
111     // now, in the position of vector<int>::iterator we can use auto which will
    automatically detects the data type.
112     for (auto it = v5.begin(); it != v5.end(); it ++) {
113         cout << *(it) << " ";
114     }
115
116     /// For Each loop to print vector:
117     for (auto it: v5) {
118         cout << it << " "; // here "it" will automatically iterate over the each
    element of the vector.
119     }
120
121
122     ///// Lets see, Deletion of a Vector.
123     v5.erase(v5.begin() + 1); // here we are pointing to the address of 20 to delete
    it.
124     for (auto it: v5) {
125         cout << it << " ";  // now the vector v5 = {10, 20, 20, 14, 52, 62} is
    resuffled to {10, 20, 14, 52, 62}
126     }
127     // But in this way we can delete one element, now lets see how we can delete couple
    of element.
128     /// syntax: vector.erase (starting address, end address after the element)
129     v5.erase(v5.begin() + 1, v5.begin() + 3); // this will delete 20 and 14.
130     for (auto it: v5) {
131         cout << it << " "; // v5[] = {10, 52, 62}
132     }
133
134
135     /// Insert Function: (how to insert in a vector)
136     vector <int> v6(2, 200);                // {200, 200}
137     // -> now, lets see how to insert an element in the beginning
138     v6.insert(v6.begin(), 100);             // {100, 200, 200}
139     // lets see how to insert multiple element.
140     v6.insert(v6.begin() + 1, 2, 150);      // {100, 150, 150, 200, 200}
141     // for (auto it: v6) {
142     //     cout << it << " ";               // {100, 150, 150, 200, 200}
143     // }
144
145     //// lets see how we can insert one vector into another vector.
146     vector <int> copy(3, 500);              // {500, 500, 500}
147     v6.insert(v6.begin(), copy.begin(), copy.end());  // {500, 500, 500, 100, 150, 150,
    200, 200}
148     // for (auto it: v6) {
149     //     cout << it << " ";     //  500 500 500 100 150 150 200 200
150     // }
151
152
153     ///////////   : Checking the Size of Vector :   /////////
154     //// lets see how to check the size of any vector.
155     cout << v6.size();          // 8
156
157
```

```cpp
    ///////////    : Popping Element from Vector : ///////////////

    ///// Lets see how to pop element from the end side of any vector.
    //now, v6[] = {500, 500, 500, 100, 150, 150, 200, 200}
    v6.pop_back();   // last 200 will be popped.
    for (auto it: v6) {
        cout << it << " ";     //  500 500 500 100 150 150 200
    }
    cout << v6.size();         // 7

    ////////////   : Swap two Vector :  //////////////

    vector <int> v7 = {25, 35};
    vector <int> v8 = {50, 30};
    v8.swap(v7);
    // or, v7.swap(v8);
    /// now, v7 = {50, 30}  and v8 = {25, 35}

    ////////////   :Clear the vector :    //////////////
    v6.clear(); //// it will erase the whole vector



    return 0;
}


void explainList() {
    ////////// ******** : LIST : ********////////////
    //->now, we are going to learn about another container LIST. It is also dynamic in nature. The difference between list and vector is that list has front operations.

    list <int> ls;

    ls.push_back(10);
    ls.emplace_back(20);


    // In vector, we have to use insert, but here we can use push_front() and emplace_front().
    ls.push_front(5);
    ls.emplace_front(7);

    for (auto it: ls) {
        cout << it << " ";   //  ls = {7, 5, 10, 20}
    }

    /// Rest functions are same as vector.
    // begin, end, rbegin, rend, clear, insert, size, swap

}



void explainDeque() {
    deque<int> dq;

    dq.push_back(12); // {12}
    dq.emplace_back(24); // {12, 24}
```

```cpp
    dq.push_front(10); // {10, 12, 24}
    dq.emplace_front(5); // {5, 10, 12, 24}

    dq.pop_back(); // {5, 10, 12}
    dq.pop_front(); // {10, 12}

    cout << dq.back(); // 12
    cout << dq.front(); // 10

    // rest functions are same as vector
    // begin, end, rbegin, rend, clear, insert, size, swap
}



void explainStack () {
    // Stack follows LIFO/FILO method.

    stack <int> st;
    st.push(1); // {1}
    st.push(2); // {2, 1}
    st.push(3); // {3, 2, 1}
    st.push(4); // {4, 3, 2, 1}
    st.emplace(5); // {5, 4, 3, 2, 1}

    cout << st.top(); // print 5 // here "**** st[2] ****" is invalid.
    // st.top() just show which element is in the top of the stack.

    // now popping elements from the top of the stack
    st.pop(); // now, st is looking like {4, 3, 2, 1}

    cout << st.top();  // 4
    cout << st.size(); // 4
    cout << st.empty(); // this will show false, as it has 4 elements.

    /// We alse can swap stack. Lets see how...
    stack <int> st1, st2;
    st1.swap(st2);

    /// here complexity is O(1) as everything happens in constant time.
}



void explainQueue() {
    /// Queue follows LILO/FIFO method.

    queue<int> q;

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

    q.back() += 5;   // the last element has been added with 5. {1, 2, 9}

    cout << q.back(); // print 9
    // now, q is {1, 2, 9}
    cout << q.front(); // prints 1
    q.pop(); // {2, 9}
```

```
274
275     cout << q.front(); // prints 2
276
277     // size swap empty same as stack.
278 }
279
280
281
282 void explainPQ() {
283     // the largest element will stay at the top
284     // -> this is also known as Maximum Heap or Max Heap
285
286     priority_queue <int> pq;
287
288     pq.push(5);     // {5}
289     pq.push(2);     // {5, 2}
290     pq.push(8);     // {8, 5, 2}
291     pq.emplace(10);     // {10, 8, 5, 2}
292
293     cout << pq.top();    // print 10
294     pq.pop();            // pop 10 , so now pq = {8, 5, 2}
295     cout << pq.top();    // print 8
296
297     // so, here there are mainly three functions: push, pop, top
298     // size, swap, empty functions are same as others.
299
300     // Minimum Heap -> means we want to put the smallest element in the top.
301     // -> this is also known as Minimum Heap or Min Heap
302
303     priority_queue<int, vector<int>, greater<int>> pq1;
304     pq1.push(5);        // {5}
305     pq1.push(2);        // {2, 5}
306     pq1.push(8);        // {2, 5, 8}
307     pq1.emplace(8);        // {2, 5, 8, 10}
308
309     cout << pq1.top();      // print 2
310
311     // push and pop => O(log(n)) , top => O(1)
312 }
313
314
315
316 void explainSet () {
317     // Now, here it is Set Container. Lets see how its actually work.
318     // NOTE: 01_It stores EVERYTHING in SORTED order and stores UNIQUE
319     set<int> st;
320     st.insert(1); // {1}
321     st.insert(2); // {1, 2}
322     st.insert(2); // {1, 2}
323     st.insert(4); // {1, 2, 4}
324     st.insert(3); // {1, 2, 3, 4}
325
326     // Functionality of insert in vector can be used also, that only increases the
    efficiency.
327     // begin(), end(), rbegin(), rend(), size(), empty() and swap() are same as those
    of above.
328     // {1, 2, 3, 4}
329     auto it = st.find(3); // it will return an iterator which points to 3.
330
```

```
    // {1, 2, 3, 4}
    auto it = st.find(6); // as 6 is not in the set, therefore it will return st.end().
    // st.end() means an interator that will point right after the last element.


    int cnt = st.count(1); // 1 -> as set holds only unique elements, so all items count will be only 1.
    int cnt = st.count(32); // 0 ->if the element is not present, then it will show 0.

    // {1, 2, 3, 4}
    st.erase(3); // {1, 2, 4} -> it will delete 3 and will maintain the sorted order.

    // We can also erase using iterator like in vector.
    //  {1, 2, 3, 4, 5}
    auto it1 = st.find(2);
    auto it2 = st.find(4);
    st.erase(it1, it2); // {1, 4, 5}

    // lower_bound() and upper_bound() function works in the same way as in vector it does.
    auto it = st.lower_bound(2);
    auto it = st.upper_bound(3);

    // other functions size(), empty(), swap() everthing is similar to vector.

    // in set everything happens in log(n) time complexity.
}


void explainMultiSet () {
    // Similar as set but it obeys only one conditions that is sorted. but elements are not unique. We can insert duplicate elements also.

    multiset <int> ms;
    ms.insert(1); // {1}
    ms.insert(1); // {1, 1}
    ms.insert(1); // {1, 1, 1}
    ms.insert(1); // {1, 1, 1, 1}

    ms.erase (1); // all 1's are erased.

    int cnt = ms.count(1);

    // only a single one erased
    ms.erase(ms.find(1));

    // rest all functions are same as set.
}


void explainUnorderedSet() {
    unordered_set <int> st;
    // -> It stores unique.
    // -> It doesn't store in sorted order.
    // -> lower_bound() and upper_bound() function does not works, rest all functions are same.
    // -> It does not stores in any particular order.
    // -> It has a better complexity than set in most cases, except some when collision happens.
}
```

```cpp


void explainMap () {
    // Map stores unique keys in sorted order.

    // map <key, value> var_name;
    map<int, int> mpp; // first int is key & second int is value.

    // map<int, pair<int, int>> mpp;  //  first int is key & second pair is value.


    mpp[1] = 2; // on the key 1 it stores 2
    mpp.insert({3, 1}); // on the key 3 it stores 1
    mpp.insert({2, 4}); // on the key 2 it stores 4


    map<pair<int, int>, int> mp; // first pair is key & second int is value.
    mp[{2, 3}] = 10; // stores 10 in the pair {2, 3}

    for (auto it: mpp) {
        cout << it.first << " " << it.second << endl;
    }
    // the output will look like this:
        // 1 2
        // 2 4
        // 3 1

    cout << mpp[1]; // 2
    cout << mpp[5]; // 0 or NULL

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

    auto it = mpp.find(5); // as 5 is not present it will point to mpp.end()

    // Syntax of lower_bound and upper_bound
    auto it = mpp.lower_bound(2);
    auto it = mpp.upper_bound(3);

    // erase, swap, size, empty functions are same as above.

    // Map works in logarithmic time. O(logn)
}



void explainMultiMap () {
    // everything is same as map but it can store multiple same keys but in sorted order.
    // only map[key] cannot be used here.
}


void explainUnorderedMap () {
    // it will have unique keys but in sorted order.
    // Unordered Map works in Constant time O(1). in worst case it goes O(n)
}

```

```cpp

//////////////////   -: IMPORTANT ALGORITHMS :-  ///////////////////

bool comp (pair<int, int> p1, pair<int, int> p2) {
    // sort it according to the second element.
    // if second element is same, then sort according to first element but in descending.

    if (p1.second < p2.second) return true;
    if (p1.second > p2.second) return false;
    // if they are same:
    // for the last condition
    if (p1.first > p2.first) return true;
    return false;
}

void explainAlgorithms () {
    // lets say there is an array, and you have to sort it using STL.
    //So, in c++ we can use sort(a, a+n)
    int arr[] = {1, 5, 3, 7, 2};
    sort(arr, arr + 5); // sort(starting_iterator, end_iterator) {1, 2, 3, 5, 7}

    // For sorting of vector:
    vector <int> v = {7, 4, 3, 6, 9};
    sort(v.begin(), v.end()); //  {3, 4, 6, 7, 9}

    // lets see how to sort in some particular position.
    int arr2[] = {10, 41, 21, 23, 51, 1};
    // here we just want to sort from 23 to 1.
    sort(arr + 3, arr + 6); // {10, 41, 21, 1, 23, 51}

    // Lets say we want to sort them in descending order.
    int arr3[] = {4, 87, 2, 14, 51, 23};
    sort(arr3, arr3 + 6, greater<int>()); // {87, 51, 23, 14, 4, 2}

    // Now, we want to sort in any other fashion or in other way.

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

    // sort it according to the second element.
    // if second element is same, then sort according to first element but in descending.
    // means, if a[] = {{1, 2}, {2, 1}, {4, 1}} , then after sorting, a[] = {{4, 1}, {2, 1}, {1, 2}} to do this we have to use:

    sort (a, a+3, comp); // comp is a bollean function. (comp is before this function)
    // now, a[] = {{4, 1}, {2, 1}, {1, 2}}
    // if comp returns false only then it will sort otherwise they are in correct order.

    int num = 7;
    int cnt = __builtin_popcount(num); // count set bits , for 7 => 0...0111 so, cnt = 3.

    // now for long long how to count set bits.
    long long num1 = 1564651654557;
    int cnt = __builtin_popcountll(num1); // count set bits , for 7 => 0...0111 so, cnt = 3.

    //////// Next-Permutation.
```

```cpp
    //// Lets know how to find next_permutation.
    // for 123 -> 123 132 213 231 312 321

    string s = "123";
    do {
        cout << s << endl;
    } while(next_permutation(s.begin(), s.end()));
    // for 123 -> 123 132 213 231 312 321
    // for 231 -> 231 312 321
    // if we want the whole permutation for 231, we can sort before printing them.

    string s1 = "231";
    sort(s1.begin(), s1.end());
    do {
        cout << s1 << endl;
    } while(next_permutation(s1.begin(), s1.end()));
    // now for 231 -> 123 132 213 231 312 321


    ////// Max Element:
    // lets say we want the maximum element of an array.
    int arr[] = {1, 2, 352, 2125, 41, 25, 54, 65};
    int maxi = *max_element(arr, arr + 8); // 2125
    // *max_element (start_address, end_address)

    // similarly, for minimum element,
    int mini = *min_element(arr, arr + 8); // 1


}
```