DIVYANSHU
BANSAL

STL

Q.) why do we write #include<bits/stdc++.h>

Ans.) We write #include<bits/stdc++.h> because it saves our time as it include all libraries of c++.
But in interview we should not write it instead write individual libraries.

Q.) what is the use of namespace std ?

Ans]
```
#include<bits/stdc++.h>
using namespace std;
namespace ray {
        int val = 50;
        int getval(){
                return val*10;
        }
}
int main(){
        double val=10.0;
        cout<< val<<endl;   // prints 10.0
        cout<< ray::val<<endl;  // prints 50
        cout<< ray::getval<<endl;  // prints 50*10
}
```

STL commands

STRINGS

1.) append(): inserts additional characters at the end of the string. (can also be done using '+' or '+=' operators). It's time complexity is O(N) where N is the size of the new string.

2.) begin(): Returns an iterator pointing to the first character. Its time complexity is O(1).

3.) clear(): Erases all the contents of the string and assign an empty string ("") of length zero. Its time complexity is O(1).

4.) compare(): Compares the value of string with the string passed in the parameter and returns an integer accordingly. Its time complexity is O(N+M) where N is the size of first string and M is the size of the 2nd string

5.) copy(): copies the substring of the string in the string passed as parameter and returns the number of characters copied. Its time complexity is O(N) N= size of copied string.

6.) empty(): Returns a boolean value. True if the string is empty and false if the string is not empty. Its time complexity is O(1).

7.) end(): Returns an iterator pointing to a position which is next to the last character. Its time complexity is O(1).

8.) erase(): deletes a substring of the string. Its time complexity is O(N) where N is the size of the new string.

9.) find(): Searches the string and returns the first occurance of the parameter in the string. Its time complexity is O(N) where N is the size of the string.

10.) length(): Returns the length of the string, Its time complexity is O(1).

11.) insert(): inserts additional characters into the string at a particular position. Its time complexity is O(N) where N is the size of the new string.

12.) size(): Returns the length of the string. Its time complexity is O(1).

13.) substr(): Returns a string which is the copy of the substring. Its time complexity is O(N) where N is the size of the substring.

Containers:

Array

```
array< int, 3> arr;        // {0,0,0}
```

data type — size of the container

Max. size of the array in global scope

→ 10^7 - int, double, char

```
int arr[10000000];
```

→ 10^8 - bool

Max. size of the array in main

→ 10^6 - int, double, char

→ 10^7 - bool

Struct

create a data type where you can store (string, int, double, char)

```
Struct node{
    string str;
    int num;
    double doub;
    char x;
  node(str_, num_, doub_, x_){
    str = str_;
    num = num_;
    doub = doub_;
    x = x_;
  }
};
main(){
  node *ray = new node("steiner", 79, 91.0, ' ');
```

Or `node ray = node("steiner", 79, 91.0, ' ');`

→ `array<int, 5> arr = {1};` // {1, 0, 0, 0, 0}

→ `array<int, 5> arr;`
`arr.fill(10);` // {10, 10, 10, 10, 10}

// here fill() function is used to fill the values in array

→ `arr.at(index);` // return the value of element of that particular index.

```
for(int i=0; i<5; i++){
    cout<<arr.at(i)<<" ";
}
```
// prints all elements of array

iterators

An iterator is an object that points to some element in a range of elements (such as an array or a container) and has a ability to iterate through those elements using a set of operators (with at least increment (++) and dereference (*) operators).

{10, 20, 30, 40}

begin(): points to the first element of an array.
→ points at 10.

end(): points to the place just after last element.
It points after 40.

r.begin(): points at the last element. i.e. points at 40.

r.end(): points to the place just before first element. Eg; r.end() will point just before 10 not 10.

Different ways of printing elements of an array using iterator

```
array<int, 5> arr = {1, 2, 3, 4, 5};
→ for(auto it: arr.begin(); it != arr.end(); it++){
      cout << *it << " ";
  }
→ for(auto it : arr.end()-1; it >= arr.begin(); it--){
      cout << *it << " ";
  }
→ for each loop
  for(auto it : arr){
      cout << it << " ";
  }
```

Imp Note: Instead of it: in for loop it will be it =

→ string s = "xhegewe";
for (auto c : s){
cout << c << " "; // x h e g e w e
}

// size
→ cout << arr.size();

// front
→ cout << arr.front(); // arr.at(0);

// end
→ cout << arr.back(); // arr.at(arr.size() - 1);

Vector

Vectors are sequence containers that have dynamic size. In other words, vector are dynamic arrays.
Just like arrays vectors elements are placed in contagious storage location so they can be ~~accepted~~ accessed and traversed using iterators. To traverse the vector we need the position of the first and last element in the vector which we can get through begin(), and end() or we can use indexing from 0 to size().

→ In array we need to define its size
Eg: int arr[50];

→ But in case of vectors you can dynamically increase and decrease ~~the~~ its size.

```
vector<int> arr;   // { } its an empty vector now.
```

classifying its a vector — its data type — name of the vector

→ `cout << arr.size() << endl;` // prints 0

How to add elements in vector?

→ push_back()

```
arr.push_back(0);   // {0}
arr.push_back(2);   // {0,2}
cout << arr.size() << endl;   // 2
```

How to remove ~~the~~ an element from vector?

→ pop_back()

```
arr.pop_back();   // {0}
```

* it pops out the element which was last inserted.

```
cout << arr.size();   // 1
```

→ `arr.push_back(0);` // {0,0}

→ `arr.push_back(2);` // {0,0,2}

→ `vec.clear();` // erases all elements of your vector at once.

→ vector<int> vec1(4,0); // {0,0,0,0}
↳ it creates vector of size 4 and fills it with all zeros.

→ vector<int> vec2(4,10); // {10,10,10,10}

How to copy entire vector into a new vector

→ vector<int> vec3(vec2.begin(), vec2.end());

// [ ) it will include the begin() but not end()

Simple way to copy

→ vector<int> vec3(vec2);

Simple operations

→ vector<int> ray;

ray.push_back(1);

// You can also use ray.emplace_back(1) instead of
// push_back() as it also takes lesser time complexity
// which will be helpful for competitive programing.

ray.push_back(3);
ray.push_back(2);
ray.push_back(5); // {1,3,2,5}

Copying elements of vector between certain range

→ vector<int> ray1(ray.begin(), ray.begin()+2);
// {1,3}

// ray.begin()+2 will point at 2 at 2nd index
// but it will not include it.

Defining 2-D vectors

A 2D vector is a vector of the vector.

Eg: vector <vector <int>> vect;

→ In case of normal vector we initialize it as

1.) vector <datatype> vector_name;

→ Now in case of 2D vector we need to create vector of datatype vector.
we simply replace datatype with vector <int>

1.) vector <vector <int>> variable_name;

Initialization and declaration - (from gfg)

```
// include <bits.stdc++.h>  ☺ <bits/stdc++.h>
using namespace std;
int main(){
 vector <vector<int>> vec
       {
          {1,2,3},        // initialization and
          {4,5,6},        // declaration of values
          {7,8,9}         // of 2-D vector
       };
 for (int i=0; i<vec.size(); i++)
       {
          for (int j=0; j<vec[i].size(); j++)
          {
             cout << vec[i][j] << " ";
          }
          cout << endl;
       }
 return 0;
}
```

```
vector<vector<int>> vec;
vector<int> ray1;
ray1.push_back(1);
ray1.push_back(2);

vector<int> ray2;
ray2.push_back(10);
ray2.push_back(20);

vector<int> ray3;
ray3.push_back(19);
ray3.push_back(24);
ray3.push_back(27);

vec.push_back(ray1);   // ray1, ray2 and ray3
vec.push_back(ray2);   // are inserted into vec
vec.push_back(ray3);   // a 2D vector
```

/* three vectors ray1, ray2 and ray3 are created.
→ value 1 and 2 pushed to ray1
→ value 10 and 20 pushed to ray2
→ value 19, 24 and 27 pushed to ray3
*/

Outputting Elements

```
for(auto vectr: vec){
    for(auto it: vctr){
        cout << it << " ";
    }
    cout << endl;
}
```

/* This nested loop iterates through each vector within vec.
The inner loop iterates through each element of the vectors and prints them, followed by a space.
After printing all elements of a vector, it adds a newline to move to the next line. */

Alternate Method of printing them

```
for( int i=0; i< vec.size(); i++){
    for(int j=0; j< vec[i].size(); j++){
        cout<< vec[i][j]<<" ";
    }
    cout <<endl;
}
```

// /* 'i' iterates through the outer loop ('vec') and j iterates through the inner vectors */

// define 10X20

1.) Vector initialization

→ `vector<vector<int>>vec(10, vector<int>(20,0));`

This line initializes a 2D vector named 'vec' with dimensions 10 rows and 20 columns all initialized with '0'.

2.) Pushing an empty row

→ `vec.push_back(vector<int>(20,0));`

It adds another row to 'vec' and it contains 20 element all initialized as 0.

3.) Output the size

→ `cout<<vec.size()<<endl; // prints 11`

This prints the size of vector 'vec' which is 11. Initially it was 10X20 2D vector but after insertion of 1 new row, in total their are 11 rows now.

4.) Modifying a specific element

→ vec[2].push_back(1);

This adds a new element at the end of 3rd row (vec[2]) of the 2D vector.

5.) Array of vectors

→ vector<int> arr[4];
arr[1].push_back(0);

This declares an array 'arr' containing 4 vectors of int. Then, it pushes '0' into the second vector (index '1') of the array 'arr'.

3-D vector initialization

vector<vector<vector<int>>> vec(10, vector<vector<int>>(20, vector<int>(30,0)));

→ This line initializes 3-D vector of 10 layers, each layer containing 20 rows and each row having 30 elements, all initialized to '0'.

Lecture-2 Of STL

Set

→ Sets are containers which store only unique values.

→ Sets stores everything in linearly ascending order.

Q.) given n elements, tell me the no. of unique elements

arr[] = {2, 5, 2, 1, 5}

3 unique elements {2, 5, 1}

→ we can find this using set and output will be {1, 2, 5} as set stores only unique and in linearly ascending order.

```
→ set<int> st;          // declaration
  int n;                // size of set
  cin >> n;
  for(int i=0; i<n; i++){   // loops n times, taking user
      int x;                // input 'x' and inserting each
      cin >> x;             // 'x' into the set using
      st.insert(x);         // 'st.insert(x)'.
  }
```

* Set inherently maintains unique elements, so repeated insertions of the same value won't create duplicates.

```
→ cout << st.size();  // prints the size of set
```

Erasing elements from set

→ st → {1, 2, 5}

st.erase(st.begin()); // st → {2, 5}

// this removes the first element from the set

// st.begin() is an iterator which is pointing at first element

→ st.erase(st.begin(), st.begin()+2); // {5}

// it remove element from 'st.begin()' up to, but not including, 'st.begin()+2.

→ st.erase() operates in logarithmic time complexity (O(log n)) for sets, where 'n' represents the no. of elements in set.

→ st.erase(5) // deletes 5 from set {1, 2}

// used when we need to remove a particular element.

More operations on set

→ set<int> st = {1, 5, 7, 8};

auto it = st.find(7); // It searches for element '7' // in the set.

// takes log(n) time complexity

→ auto it = st.find(9); // It searches for 9 in set

// If the element is not present in set, it will be equal to 'st.end()'.

Insertion in set

```
→ st.emplace(6);  // or st.insert(6)
cout << st.size(); // prints 1
```

```
→ set<int> st;
st.insert(5);
st.insert(5);
```

How to iterate through a set

1.)
```
for(auto it = st.begin(); it != st.end(); it++){
    cout << *it << " ";
}
```

2.)
```
for(auto it : st){
    cout << it << endl;
}
```

→ First loop uses explicit iterators, while the second loop utilizes a more concise range based for loop.

deleting the entire set

```
st.erase(st.begin(), st.end());
```

Unordered set

→ Unordered set is same as set ~~set~~ it contains unique elements but not in sorted or ascending order like sets.

→ Unordered set has lower time complexity than set.

In general → O(1)

In worst case - O(n)

→ In competitive programming try to use unordered set instead of set until required.

→ If TLE occurs switch to set.

It has same operations like set

unordered_set<int> st;
st.insert(2);
st.insert(3);
st.insert(1);

Multiset

Multiset doesnot contain unique elements i.e. it can also have duplicates but in sorted order.

→ multiset<int>ms;
ms.insert(1);
ms.insert(1);
ms.insert(2);
ms.insert(2);
ms.insert(3); // ms.emplace() can also be used

// st→{1,1,2,2,3}

→ ms.erase(2) // all the instances of 2 will be erased

→ auto it=ms.find(2); // returns an iterator pointing to // first element of 2

→ ms.clear(); // deletes the entire set

→ ms.erase(ms.begin(), ms.end()); // deletes the entire set

Iterating through multiset

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

for(auto &it: st){
cout<<it<<endl;
}

→ st.count(2); // counts the occurance of 2 in multiset

Erasing specific elements

ms.erase(ms.find(2));

ms.erase(ms.find(2), ms.find(2)+2);

→ erases the first occurance of 2 in the multiset

→ erases a range of elements from the first occurance of '2' up to not including ms.find(2)+2.

Map

In maps each element has a key value and a mapped value. No two mapped values can have the same key values but two different key values can have the same mapped value.

Map initialization and key value insertion

```
map<string, int> mpp;        // mpp is a map with
                             // keys as strings and
                             // value as integer
mpp["ray"] = 27;
mpp["himu"] = 31;
mpp["praneer"] = 31;
mpp["sandeep"] = 67;
mpp["tanik"] = 89;
mpp["ray"] = 29;
```

Key value, mapped value

/* The map stores unique keys. Hence, when a key like "ray" is reassigned, it updates the existing value rather than creating a new key-value pair */

Erasing Elements

```
mpp.erase("ray");      // mpp.erase(key)
mpp.erase(mpp.begin()); // mpp.erase(iterator)
                        // removes first element

mpp.clear();  // entire map is cleaned up
```

Finding elements

```
auto it = mpp.find("ray"); //pts. to where ray lies
auto it = mpp.find("simran"); //points to end as she doesn't
                              //exist
```

Checking if map is empty or not

```
if(mpp.empty()){
    cout<<"Yes it is empty";
}
```

→ mpp.empty() returns 'true' if the map is empty otherwise it returns 'false'.

Counting instances of ~~element~~ key

```
mpp.count("ray"); //always returns 1 as it stores only
                  //1 instance of ray
```

Iterating through the map

```
for(auto it : mpp){
    cout<<it.first<<" "<<it.second<<endl;
}
for(auto it=mpp.begin(); it!=mpp.end(); it++){
    cout<<it->first<<" "<<it->second<<endl;
}
```

```
pair<int, int> pr;
pr.first = 1;
pr.second = 10;
```

Unordered Map

In unordered map elements are not stored in any order. It has O(1) time complexity but in worst case it can go upto O(n)

```
unordered_map<int, int> map;  // declaration
```

→ All the functions are same as maps

→ `unordered_map<pair<int, int>, int> mpp;`

Pair class

→ `pair<int, int> pr={1,2};` // declaration of pair
`pair<pair<int, int> int> pr={{1,2}, 2};` // Nested pair

Accessing and printing elements of pair

→ `pair<pair<int, int>, pair<int, int>> pr={{1,2},{2,4}};`
`cout << pr.first.first;` // output: 1
`cout << pr.second.second;` // output: 4

Containers using pairs

```
vector<pair<int, int>> vec;
set<pair<int, int>> st;
map<pair<int, int>, int> mpp;
```

Multimap usage

→ `multimap<string, int> mpp;`
`mpp.emplace("ray", 2);`
`mpp.emplace("ray", 5);`

/* in this case ray is used as the key for both entries. It allows multiple values for the same key, so both '2' and '5' are associated with key "ray" */

Map
↳ unique
↳ ordered

Unordered Map
↳ unique
↳ unordered

Multimap
↳ Not unique
↳ ordered

Stack

A stack is a Last-in, First-out (LIFO) Data structure.

// declaration of stack
→ stack<int> st; // lifo ds

st.push(2);
st.push(4);
st.push(3);
st.push(1);

Basic operations of stack
→ push / emplace
→ pop
→ top
→ size
→ empty

// push adds element to top of stack

→ cout << st.top(); // prints 1
~~cout~~ st.pop(); // deletes the last entered element
cout << st.top(); prints 3
st.pop();
cout << st.top(); // prints 4

/* st.top() - returns the top element of stack
st.pop() - removes the top element from the stack
*/

→ bool flag = st.empty(); /* returns true if stack is empty else false */

→ while (!st.empty()){
st.pop();
}

/* The while loop continously pops element from the stack until the stack becomes empty. */

→ cout << st.size() << endl; // no. of elements in stack

→ stack<int> st;
if(!st.empty()){
cout << st.top() << endl;
}

/* if you write st.top() for empty stack it will throw run time error, so it is necessary that you first check whether stack is empty or not.
*/

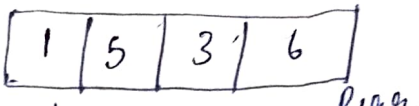
Queue

Queue is data structure which works on FIFO principle.

Queue Operations

→ push
→ front
→ rear
→ pop
→ size
→ empty

→ queue<int> q; // declaration of queue
q.push(1);
q.push(5);
q.push(3);
q.push(6);



cout << q.front(); // prints 1
~~cout~~ q.pop(); // deletes the front element i.e. 1
cout << q.front(); // prints 5

→ while (!q.empty()){
q.pop();
}
/* All operations have linear time complexity i.e. O(n) */

→ queue<int> q;
for(int i = 0; i < 10; i++){
q.push(i);
}
/* initializes a new queue 'q' and uses a loop to push integer from '0' to '9' into the queue. */

Priority Queue

Operations: → push → pop → top → size → empty

→ It stores elements in a certain priority either ascending or descending

→ Priority queue uses concept of heapify from heap sort.

Declaration and initialization of Priority Queue

```
→ priority_queue<int> pq;
   pq.push(1);   // inserts elements into priority queue 'pq'
   pq.push(5);
   pq.push(2);
   pq.push(6);

   cout<< pq.top(); // prints 6
   pq.pop();
   cout<< pq.top(); // prints 5
```

/* This creates a priority queue 'pq' of integers by default as max heap.

→ pq.top(): retrieves the element with the highest priority.

→ pq.pop(): removes element with highest priority.

Priority Queue of Pairs

~~→ priority_queue<int, int>~~

```
→ priority_queue<pair<int,int>> pq;
   pq.push({1, 5});
   pq.push({1, 6});
   pq.push({1, 7});
```

How to retrieve min. value

```
priority_queue<int> pq;
pq.push(-1);          // pq.push(-1*el);
pq.push(-5);
pq.push(-2);
pq.push(-6);
cout<<-1*pq.top()<<endl; //prints 1
```

Min Priority Queue

```
priority_queue<int, vector<int>, greater<int>> pq;
pq.push(1);
pq.push(5);
pq.push(2);
pq.push(6);
```

1	2	5	6

```
cout<<pq.top()<<endl; //prints 1
```

Min Priority Queue with pairs

```
priority_queue<pair<int, int>, vector<pair<int,int>>,
                greater<pair<int,int>>> pq;
```

Dequeue

Deque provides flexible insertion and deletions from both the ends.

operations

- push_front(): inserts at front
- push_back(): inserts at back
- pop_front()
- pop_back()

→ size() → empty() → clear()

→ at(): Accesses the element at a specified position.

iterators

→ begin()
→ end()
→ rbegin()
→ rend()