STL
Standard Template Library

#include <bits/stdc++.h> → this contain all the library.

4 parts
① Algorithms ② Containers ③ Functions ④ Iterators

Pairs

It is a part of utility library.

pair <int, int> p;
- int → data type of 1st element
- int → data type of 2nd element.

to access 1st element in a pair we use pair.first f^n. (p.first here)
& for second we use pair.second. (p.second here)
as our pair name is p.

Initializing pair;

pair<int, int> p = {1, 3};

Nesting of pairs

Here suppose we have to store 1, 3, 4 into a pair

pair<int, pair<int, int>> p1
= {1, {3, 4}};
{ - , {-, -} }

- p1.first → 1
- p1.second.first → 3
- p1.second.second → 4

```
pair<int, int> arr[] = {{1,2}, {2,5}, {5,1}};
                        0      1      2
```

Pairs can be treated as a data type.

- arr[1].second.

Vectors → It is a container which is dynamic in nature. When we do not know about the size of the data structure then we use vector.

Declaration of vector

```
vector<int> v;
```

```
v.push_back(1);   ⟶ Inserts 1 in the vector.
v.emplace_back(2);
```

Generally emplace_back is faster than push_back.

Declaration of vector of pair

```
vector<pair<int, int>> vec;
vec.push_back({1, 2});
vec.emplace_back(1, 2);
```

While inserting a pair we have use curly braces if we are using push_back and curly braces are not necessary while using emplace_back because it automatically assumes then a pair

vector<int> v(5, 100); → Initialize with
(size)
{100, 100, 100, 100, 100}
of vector containing 5 instances of 100

vector<int> v(5); → {-, -, -, -, -}
some garbage value

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

Copying a vector into another vector

vector<int> v2(v1);

v_2 = {20, 20, 20, 20, 20}.

Accessing elements in a vector

Using iterator

Iterator points at the memory address. So iterator is a pointer.

Syntax:-

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

- vector → data structure
- int → the data type
- it → name of iterator
- v.begin() → It points directly to the memory means pointing v.begin will give the address of first element of a vector.

*(v.begin())

↓

It gives value at the memory/address at v.begin().

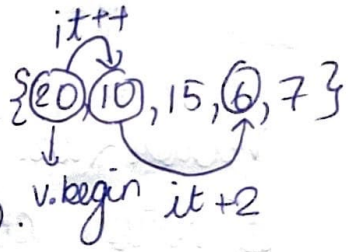
vector<int>::iterator it = v.begin();

it++;

cout << *(it) << " "; O.P → 10.

it = it+2

cout << *(it) << " "; O.P → 6.



v.begin is address of the first element of the vector

Other iterators

end, rend (reverse end) & rbegin (reverse begin)

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

vector<int>::~~int~~ iterator it = v.rend();

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

v = {10, 20, 30, 40} ↑

v.end will not point to 40.
end points to a memory element which is right after the last element.

v.rend() (Never Used)

↑ {10, 20, 30, 40}

v.rbegin (Never Used).

{10, 20, 30, 40}
↑

it ++ it comes to 30.

```
cout << V[0] << " " << v.at(0);
cout << v.back() << " ";
```

{10, 20, 30}

Points the last element in a vector.

Printing the entire vector

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

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

→ Here auto automatically assigns it to a vector iterator it automatically even defines the data type

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

→ it on v.

It iterates on the data type not an iterator here. We don't use value at (*) operator here.

ex- auto b = "raj" Here b will be automatically string.

→ a will be automatic

ex- auto a = 5 int.

Here the compiler knows 5 is an integer so it automatically assigns the data type.

Erasing elements from a vector

. erase (iterator) Used for deleting a single element

↓

location of the address which we want to delete.

Ex → v. erase(v.begin()+1); {10,20,30,40}

It deletes 20. New v = {10,30,40}.

Deleting multiple elements at the same time

. erase (start element's address, end address)

↓

It is after the element which we want to delete.

Ex - v = {10,20,30,40,50}

Suppose we want to delete 20,30 & 40

So,

v. erase (v.begin()+1, v.begin()+4);

at the end we have to give address of the next element.

it is like [Begin, end).

Inserting into a vector

```
vector<int> v(2,100); // {100,100}
v.insert(v.begin(),300); // {300,100,100};
```

- `v.begin()` → Address where we want to insert
- `300` → Value which we want to insert.

ex – {10, 20, 30, 40} (5)

If we want to insert 5 after 10 we will use

```
v.insert(v.begin()+1, 5);
```

Inserting multiple elements

```
v.insert(v.begin(),2,10)
```

- `v.begin()` → Address where we want to insert.
- `2` → No. of elements we want to insert
- `10` → The number we want to insert.

So doing this on v = {40, 20, 30}

will make v = {10, 10, 40, 20, 30}.

For inserting a vector into another vector:

$v = \{30, 10, 10, 100, 100\}$

```
vector<int> copy (2, 50); // {50, 50}
v.insert (v.begin(), copy.begin(), copy.end());
```

New $v = \{50, 50, 30, 10, 10, 100, 100\}$.

This inserts the entire vector copy into the vector v.

If we want to insert a portion of the vector,

```
v.insert (v.begin(), copy.begin(), copy.end());
```

- v.begin() → index where we want to add copy vector.
- copy.begin() → The elements from which we want to insert.
- copy.end() → The element after the element which we wanted to insert because [start, end)

v.size() → gives the number of elements in a vector.

v.pop_back() → removes the last element from the vector.

v.swap(v2);

It swaps the entire vectors

ex → initially

$v_1 = \{10, 20\}$

$v_2 = \{30, 40\}$

$v_1.swap(v_2)$;

$v_1 = \{30, 40\}$ $v_2 = \{10, 20\}$.

v.erase → clears the entire vector

cout << v.empty(); It is a boolean func^n

Returns 1 if v is empty
else returns 0.

Lists

It is similar to vector.

The only difference is it gives front operations as well. It is a container & dynamic in nature.

```
list<int> ls; → declaration
ls.push_back(2); // {2}
ls.emplace_back(4); // {2,4}
ls.push_front(5); // {5,2,4}
ls.emplace_front(6); // {6,5,2,4}
// rest are same as vector.
// begin, end, rend, rbegin, clear
// insert, swap and size.
```

an insert in a vector is more costlier (in terms of Time complexity) as compared to push front in a list.

Deque

exactly similar to list & vector.

```
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(); → shows the last element

dq.front(); → shows the first element.

// rest same as vector.

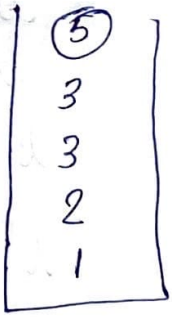
// begin, end, rbegin, rend, clear, insert, size & swap.

Stack

Stack has LIFO

↳ last in first out

```
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(); ~~delete~~ prints the top element of the stack. it does nothing to the top element.
5 in our case.

Index based accessing is not allowed in stack.

st.pop(); {3,3,2,1}

push, pop, top } all are O(1) in time complexity

cout<< st.top(); //3
cout<< st.size(); //4
cout<< st.empty(); False

stack<int> st1, st2;
st1.swap(st2);

Queue

Similar to stack FIFO
↳ First In First Out. (Imagine queue as a line of people for buying ticket)

queue<int> q;
q.push(1); //{1}
q.push(2); //{1,2}
q.emplace(4); //{1,2,4}
q.back() += 5 → ~~changes~~ adds 5 to the last element

4+5→9.
2
1

All the operations are happening in O(1) Time complexity.

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

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

// size swap empty same as stack.

Priority Queue

The elements which has the largest value stays at the top.

```
// Max heap
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(); // prints 10.
pq.pop(); // Removes the greatest element.
cout << pq.top(); // prints 8.
```

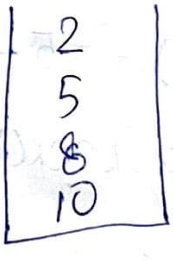
A tree is maintained inside.
Data is not stored in linear fashion

push, top, pop } main functions.

push → O(log n)
pop → O(log n)
top → O(1)

// Minimum heap → The priority queue which stores minimum element at the top.

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



Set

Stores everything in a sorted order and stores unique values only.

A tree is maintained


```
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}
// {1,2,3,4,5}
auto it = st.find(3);
```

→ It won't add 2 as it stores only unique elements.

It stores 3 ~~of~~ before 4 as it stores everything in a sorted order.

↙ It returns an iterator which points to the 3. (points to the address)

```
// {1,2,3,4,5}
auto it = st.find(6)
```

The 6 is not in the set.
So when the element is not in the set it always returns ~~set~~ st.end().
it points to right after end. {1,2,4,5,6}

```
int count = st.count(1);
```

It can ~~have~~ give only 1 or 0.
0 when it doesn't exist
1 when it is present and only 1 because set contains only unique elements.

```
st.erase(5); // erases 5 & maintains the sorted order
```

→ O(log n) Time Complexity.

{1,2,4,6}

we can also erase by giving address of the iterator

```
auto it = st.find(3);
st.erase(it);
```

→ it takes O(1) 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).

In set everything happens in dlog n) time complexity.

// lower_bound() & upper_bound() functions work in the same way as they work in vectors.

Syntax

```
auto it = st.lower_bound(2);
auto it = st.upper_bound(3);
```

size, empty, swap everything is similar to vector, begin, end etc.

~~Hello~~ MultiSet

It can stores multiple occurance unlike set.

Rest all functions are same as set.

It stores in sorted manner.

```
multiset<int> ms;
ms.insert(1); // {1}
ms.insert(1); // {1,1}
ms.insert(1); // {1,1,1}
ms.erase(1); // all 1's are erased.
int count = ms.count(1); → Returns no of 1's.
ms.erase(ms.find(1));
```

If we want to erase 2 1's

```
ms.erase(ms.find(1), ms.find(1)+2);
```

↓ It finds 1 and ↓ deletes 2 of them.

But if we say erase address it only erases that portion

Here ~~word~~ we erase the iteration to delete just one occurance of 1 instead of saying erase element as it deletes all the ~~se~~ 1's.

Unordered Set

Everything same as set.
It does not store in sorted order.
It only stores unique elements.

All the operations have Time complexity O(1)

All the functions are same only the lower bound and upper bound functions do not work.

For worst case — Time complexity O(N)
Happens in once in a millennium

Map → It is a container

Map stores everything in respect of {key, value}

Keys & Values

Keys must be unique

Key can be any data type

Values can be same.

Value can also be of any data type.

Map stores unique keys in sorted order.

Map works in O(log N) time complexity.

```
map<int, int> mpp;
```
- int (1st) → data type of key
- int (2nd) → data type of value

```
map<int, pair<int, int>> mpp;
```
- key is 1 integer
- value is a pair of 2 integer

```
map<pair<int, int>, int> mpp;
```
- Here key is a pair of 2 int
- Value is 1 integer

```
mpp<int,int>mpp;
mpp[1]=2;
  ↳key  ↳value
mpp.emplace({3,1});
mpp.insert({2,4});
```

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

These three lines will be storing like this as it should be in sorted order.

mpp[{2,3}]=10; → It is for map<pair<int,int>,int>mpp;

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

Output (In the sorted order of key)

For each loop

```
for(auto it : mpp)
{ cout << it.first << " " << it.second << endl;
}
```

[{1,2} {2,4} {3,1}]

```
cout << mpp[1]; // it says 2.
cout << mpp[5]; // 0 or null (it doesn't exist).
auto it = mpp.find(3);   // ← It gives iterator to 3,1.
cout << *(it).second;
auto it = mpp.lower_bound(2);
auto it = mpp.upper_bound(3);
auto it = mpp.find(5);
```

∵ 5 is not in the map it will point to mpp.end().

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

~~auto~~ Multimap — similar to map.

It can store duplicate keys.

everything in sorted order.

Unordered map

It can not store duplicate keys.

It will not store in sorted order.

Almost in every case unordered map works in O(1) time complexity.

In worst case → O(N) Time complexity (Happens very rarely)

Algorithms

sort(a, a+n); //For arrays
→ position of starting iterator
→ position of last iterator [start, end).

sort(v.begin(), v.end());

a+2 a+4

{1, 3, 5, 2}

sort(a+2, a+4);

{1, 3, 2, 5}.

If I want to sort this portion

For sorting in descending order

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

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

// Sorting it in order of increasing 2nd element.
and if second element is same sort in
according to first element but in descending.

sort (a, a+n, comp);

↳ Here comp is a self-written boolean comparator.

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

bool comp (pair<int, int> p1, pair<int, int> p2)
{
If (p1.second < p2.second)
return true;
If (p1.second > p2.second)
return false;
If (p1.second = p2.second)
{ If (p1.first > p2.first)
return true;
else
return false;
}

whenever writing comp just think in terms of 2 pairs only.

```
int num = 7;
int count = __builtin_popcount();
```
↗ It returns the no of set bits (1).

```
long long num = 165765786578 69;
int count = __builtin_popcountll();
```
↙ If its long long.

```
string s = "123";
sort(s.begin(), s.end());
do {
    cout << s << endl;
} while (next_permutation(s.begin(), s.end()));
```
↳ It returns all the possible permutation.

s = 123
s = 132
s = 213
s = 231
s = 312
s = 321
s → Null.

when there are no more permutations left the while loop cond^n becomes false.

If s was "231"
s = 231
s = 312
s = 321
Null.

So if we want to have all the permutations we should start from the sorted string.

```
int maxi = *max_element(a, a+n);
```
↓ It gives max element of an array/vector

```
int mini = *mini_element(a, a+n);
```