C++

STL Standerd Template Library.

1. Algorithm
2. Containers
3. ~~functions~~
4. Iterators

containers

before moving to containers, Let's understand pairs:

pair → utility library. (class template)

① pair <int, int> p = {1, 3}
first second

nested pairs

② pair <int, pair <int, int>> p = {1, {1, 2}}
first second
second. first
second. second

array pairs

③ pair <int, int> arr[] = {{1, 2}, {3, 4}, {5, 6}}
arr[]
first second
first second
arr[0]
first second
arr[2]

How to access?

① cout << p.first << p.second;
1 3

② cout << p.first << p.second.first << p.second.second;
1 2

③ cout << arr[0].first << arr[0].second;
1 2

Vector

Date
Page

First container we are going to learn is : Vector

⇒ **Vector:** Class template / container

int a[5] = {-, -, -, -, -};
You can't add one more element
So this is static array
but if you want to use array in dynamic way
you can use vector. (dynamic array);

* vector<int> arrname;
  ↳ creates an empty container.

v.push_back(-);
v.emplace_back(-);

vector → v

vector<int> v;
v.push_back(1); → {1}
v.emplace_back(2); → {1,2}

Both functions are used to add the element in a vector by increasing the size of vector. (dynamically)

faster & better

* **vector of pairs**

vector vec

vector<pair<int int>> vec; → {}

vec.push_back({4,5}); → {4,5}
vec.emplace_back(4,5); → {4,5}

* vector<int> v1(5,100); → vector v1 {100,100,100,100,100};
* vector<int> v2(5); → vector v2 {0,0,0,0,0}

depends on compiler — what value will be there.

* vector <int> temp(v1);
copies vector v1 into temp vector

How to Access?

You can access vector elements same like an array.

- v.at(-);
- v.begin();
- v.end();
- v.rend();
- v.rbegin();

for ex. v[0] or v.at(0)

v[0] ↑ best

v.at() ↑ function not much used

Other ways

imagine you have vector v {20,16,15,6,7}

* use of iterator

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

{20,16,15,6,7}

i↑

cout << *(i) << " "; → 20

i++;

{20,16,15,6,7}

↑ i

cout << *(i) << " "; → 16

i = i+2;

{20,16,15,6,7}

↑ i

cout << *i << " "; → 6

Iterator functions – returns pointers so to access value of those pointers you have to use * (dereference operator)

first value of vector: *(v.begin());

last value (element) of vector: *(v.end() - 1);

reverse

first from end element: *(v.rbegin());

last from end element: *(v.rend() - 1);

→ points to element past the last element, which is one position after last element.

for loop

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

(or)

```
for (auto i = v.begin; i != v.end(); i++)
{ cout ---------- }
```

v.back()
↑
direct access to last element (member function)

for each loop

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

for i in v

Erase

iterator

✓ v.erase(v.begin()+1);

✓ {10, 20, 30, 40, 50}

↑ erase

v.erase(-)
v.erase(-,-)

✓ {10, 30, 40, 50}

✓ v.erase(v.begin(), v.begin()+2)

start iterator — end iterator

✓ {10, 30, 40, 50}

↑ erase

✓ {50}

Insert

(position)

v.insert(v.begin(), 3, 70)

↑ begin for 3 times 70 value hoga.

v.insert(-,-,-)

{70, 70, 70, 50}

for example we have one vector → temp.

{10, 10, 10, 20}

v.insert(v.begin(), temp.begin(), temp.end()-1);

{10, 10, 10, 20, 70, 70, 70, 50}

v.size() → size

v.pop_back() → deletes last element

List / Dequeue

v.swap(v1); v → {30,20} ⇒ v = {20,40}
v1 → {20,40} ⇒ v1 = {30,20}

v.clear() → empty vector (erases entire)

v.empty() → true / false

⇒ List:

exactly similar as vector
it gives you front operations as well

list<int> ls;

... push_back() ✓ → {1,5,6,9}
... emplace_back() ✓ → {1,5,6,9,15}
new { push_front(4) → {4,1,5,6,9,15}
emplace_front(3) → {3,4,1,5,6,9,15}

ls.push_front(-)
ls.emplace_front(-)
} since insertion from front end is cheaper in list; these functions are there.

⇒ Dequeue: Doubly ended queue.

deque<int> dq; You can delete and add from both ends.

insertion: dq.push_back(5) / dq.emplace_back(5) same as vector
dq.push_front(5) / dq.emplace_front(5) of list

deletion: dq.pop_back(); | dq.front() } accessing elements
dq.pop_front(); | dq.back() } size, swap

other functions as it is :- begin, end, rbegin, rend, clear, insert...

Difference

vector dynamic array	list LinkedList	deque Doubly ended queue
element access: random access O(1)	sequential access O(n)	Random access O(1)
Insertion/deletion (front) Inefficient O(n)	efficient O(1)	efficient O(1)
Insertion/deletion (back) efficient O(1)		
Memory contiguous	requires more memory due to pointers	contiguous, but segmented memory
Access v[i], at() front(), back()	front() back()	d[i], at(), front(), back()
Insert/remove push-back() pop-back()	push-front() → u push-back() → u pop-front() → u pop-back() → u	
Size size(), resize(), capacity()	size(), resize()	size(), resize()
Iterators begin(), end()	begin(), end()	begin(), end()

Stack / Queue

⟹ **Stack:** (LIFO)

stack <int> st;

O(1)
- st.push(1) ✓ } adding element
- st.emplace(1) ✓ }
- st.top() → gives top element
- st.pop() → top removed, top--
- st.size() → gives no. of stack elements
- st.empty() → pop everything

swap function same like that dynamic array & list (vector) & dequeue.

⟹ **Queue:** (FIFO)

queue <int> q;

O(1)
- q.push(1); — 1
- q.push(2); — 1 | 2
- q.emplace(4); — 1 | 2 | 4
- q.back() += 5; — 1 | 2 | 9
- q.front(); → (1)
- q.pop() — 2 | 9
- q.front(); → (2)

size & swap functions are as it is

string: lexicographically → Hello ⓒ > Hello ⓔ → H comes first
chronologically...

⟹ Priority queue:

element with Highest value stays first.

Max heap

```
priority_queue <int> pq;        pq
pq.push(5)                      5
pq.push(2)                      5|2
pq.push(8)                      8|5|2
pq.emplace(10)                  10|8|5|2
pq.top() ⟶ (10)
pq.pop()                        8|5|2
pq.top() ⟶ (8)
```

size, swap is same as ....
empty

Min heap

element with lowest value stays first

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

```
pq.push(2)          2
  "    (5)          2|5
       (8)          2|5|8
       (1)          1|2|5|8
```

pq.top() ⟶ (1)

push } O(log n)
pop }

O(1) → top()

terminal: insert(position, element)
↳ push_back → append(last)
emplace_back → same but better...

⇒ set → sorted, unique } everything

```
set<int> s;
s.insert(4);          4
s.emplace(4);         4
s.insert(3);          3 4
s.insert(3);          3 4
```

s.find(3); → returns an iterator
s.find(6); → returns last element

```
s.erase(3);           4
```

iterator

s.count(element) → 1 or 0

You can also give an iterator to erase

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

for some set st.

terminal: erase(start, end)

```
s.lower_bound(2);
s.upper_bound(3);
```

} return

O(log n) } everything

Multiset / Unordered Set

⇒ Multiset ← unique ✗
sorted ✓
same like set

```
multiset<int> ms;
ms.insert(1)
 "   "   "
 "   "   "
```

{1,1,1}.

//ms.erase(1) → deletes every occurances of 1.

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

- erase address → that portion
- erase value → all occurances will be deleted

⇒ unordered set ← unique ✓
unsorted ✓
(NOT sorted)

all operations are same
lowerbound & upperbound (won't work).

O(1)
↓ can go till
O(N)
once in a millenium

MAP

Date
Page

map :

key, value pair

unique & sorted

key → value

```
map<int,int> m;
map<int,pair<key,key>> mp;
map<pair<int,int>,int> mpp;
```

(or)

(or)

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

another declaration

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

all other functions as it is

multimap :

keys can be duplicate

same as map

unordered map :

keys can be unordered but unique.

all the containers → pair

1. vector
2. list
3. deque
4. stack
5. queue
6. priority queue

7. set
8. multiset
9. unordered set

10. map
11. multimap
12. unordered map

some Algorithm to remember

sort(a, a+n) / sort(v.start(), v.end());

start end

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

sorting in my way

Imp

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

// sorting → second element (desc)
// elements same/duplicate
// sorting → first element (desc)

sort(a, a+n, comp);

comparator

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;
}

* int num = 7; 111
int count = __builtin_popcount();
// How many ones?
cout >> count; → (3)
same

* if num is long long num = 16785788 67
__builtin_popcountll()

* string s = "123";
// sort first → sort() func.
do { cout << s << endl;
} while (next_permutation(s.start(), s.end()));

s = (123) ✓ → print s
s = (132) ✓ → " "
s = (213) ✓ → " "
✓ → " "
: : :
NULL

checks next permutation

* int maxi = *max_element(a, a+n);
↑
Iterator
same
*min_element(a, a+n);