

Notes Made By

- RITI Kumari

1. Pairs (to store value in form of pairs)

pair <int, int> p = {1, 2};

C++ cout << p.first << " " << p.second;

STL o/p → 1 3

pairs of pairs

pair <int, pair<int>> p = {1, {3, 2}};

cout << p.first << p.second.first << p.second.second;

O/p → 1 3 2.

1	3 2
---	-----

p.first p.second.

pair of arrays.

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

{ cout << arr[1].second;

O/p → 4

1 2	3 4	5 6
0	1	2

## \* Vectors

Array - doesn't provide dynamic allocation. After declaring the size, the size can't be increased or decreased.

`vector < int > v;` // It defines a null vector.

`v.push_back(1);`

↓  
inserts an ~~new~~ element in the empty  
vector

`v.emplace_back(2);`

1

size = 1

1 2

size = 2

## Vector of pairs

`vector < pair < int, int > > vec;`

`vec.push_back({1, 2});` // push-back uses curly bracket

`vec.emplace_back(1, 2);` // In emplace\_back curly braces is not used but in case of pairs of pairs

Declare a vector of given size with every element having some value.

`vector < int >(5);` // A vector of size 5 is created

{0, 0, 0, 0, 0};

`vector < int > v(5, 20);`

{20 20 20 20 20};

## copying a vector

```
vector<int> v2(v1);
```

- \* Iterators → pointing to a given address (location).  
Iterators can be pushed front or back using  
`it++ & it--`:

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

```
it++;
```

```
cout << *it << " "; // O/P = 20.
```

↓  
to access the element at given itr

10	20	30	40
----	----	----	----

↑      ↑      ↑  
it    it+1    it+2

10	20	30	40
----	----	----	----

↑      ↑      ↑      ↑  
v.rend() v.begin()      v.end() v.begin()  
(points to      points to      memory address  
memory      end)      after 40  
address before  
10)

If you don't want your iterator to change on  
`it++` we use

`v.cbegin()`  
↓  
constant

Accessing elements without iterator.

Indexing is similar as that on array.

$v[0]$        $v.at(0);$

for the last element we ~~can't~~ use

101201301

$cout << v.back(); << " ";$  Qp → 30

Ways to print the vector

- 1) 

```
for (vector<int>::iterator it = v.begin() ; it != v.end() ;  
      it++) {  
    cout << *(it) << " ";
```
  - 2) 

```
for (auto it = v.begin() ; it != v.end() ; it++) {  
    cout << *it << " ";
```
  - 3) 

```
for (auto it : v) {  
    cout << it << " ";
```
- It moves through the element & not the iterator.  
 auto = automatically converts to the given datatype  
 erase

`vector_name.erase ( iterator )`

10	20	30	40
----	----	----	----

v. erase( v. begin() )

$$\%P = 10$$

for a range

10	20	12	23	35
----	----	----	----	----

`v.erase(v.begin() + 2, v.begin() + 4);`

$$\%p = 10, 20, 30$$

## 11 Inseed functions

helps in inserting element at a given place.

Vector<int> v(2, 100); { 100, 100 }

`v.insert(v.begin(), 300); { 300, 100, 100 }`

```
v.insert(v.begin() + 1, 2, 100);
```

$$\{ 300, 10, 10, 100, \\ 100 \},$$

11 Inserting a copy vector in vector v.

```
vector<int> copy(2,50); // {50, 50}
```

v.visit (v.begin(), copy.begin(), copy.end());

11 {50, 50, 300, 10, 10, 100, 100}

Size

`cout << v.size();`      O/p → { 0, 20 }  
                                  O/p → 2

Erasing the last element

10	20
----	----

v.pop\_back(); {10}

Swapping 2 vectors.

v<sub>1</sub> = {10, 20}

v<sub>2</sub> → {30, 40}

v<sub>1</sub>.swap(v<sub>2</sub>);

v<sub>1</sub> = {30, 40}

v<sub>2</sub> → {10, 20}

Erase the entire vector

v.clear();

cout << v.empty(); // checks if an array is empty or not.

\* list → stores the element in the dynamic fashion as vector does. You can push at front but in vector only at back.

list<int> ls;

ls.push\_back(2);

// {2}

push-front()

ls.emplace\_back(4);

// {2, 4}

pop-front()

ls.push\_front(5);

// {5, 2, 4}

push-back()

ls.emplace\_front(6);

// {5, 2, 4}

pop-back()

11 rest functions are same as vector

begin	clear	swap
end	insert	
rbegin	erase	
rend	size	

- \* Deque - deque is a container where elements could be pushed front as well as in the back.

Access the front element directly by dq.front();

dq  
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}

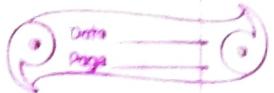
dq.back(); // last element

dq.front(); // first element

Rest func are same as vector

(begin, end, rbegin, rend, ~~clear~~, insert  
size, swap)

- \* Stack (LIFO) last in first out.



`push / pop operatn's` (It pushes elements in front)

`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();`

// points 5

No random access is allowed : `st[5]` → invalid

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

`cout << st.size(); // 4`

`cout << st.empty(); // Stack is empty or not`

`stack < int > st1;`

`Stack < int > st2;`

`st1.swap(st2);`

It doesn't have iterators like begin, end, ~~first~~ etc.

Queue  $\rightarrow$  FIFO (first in first out)

- 1) Same as stack
- 2) limited functionalities
- 3) dynamic in size
- 4) But it doesn't stores the last inserted element at the top rather it stores the first inserted element at the top.

queue<int> q;

q.push(1);      || {1}

q.push(2);      || {1, 2}

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

last element

cout << q.back();    || 4

front element

cout << q.front();    || 1      (q.top() doesn't work)

size, swap, empty are same as stack.

Priority queue  $\rightarrow$  Stores the element in a sorted fashion. (In descending order)

Max<sup>m</sup> heap

greater the value, at the top.

priority-queue < int > pq;

```

pq.push(5);    // {5}
pq.push(2);    // {5, 2}
pq.push(8);    // {6, 5, 2}
pq.push(10);   // {10, 6, 5, 2}
cout << pq.top(); // 10

```

```

pq.pop();      // {6, 5, 2}
cout << pq.top(); // 6

```

size swap empty function are same as stack & queue.

\* Min-heap  $\rightarrow$  ascending order (min<sup>m</sup> element at 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 element in a sorted order & unique elements only.

```

set < int > st;
st.insert(1); // {1}

```

st::emplace(2);  $\{1, 2\}$ .

st::insert(2); → not takes this 2 as it has taken.

st::insert(4);  $\{1, 2, 4\}$

st::insert(3);  $\{1, 2, 3, 4\}$ .

begin(), end(), subbegin(), rend(), size(),  
empty() & swap() are same as those of  
above

$\{1, 2, 3, 4, 5\}$ .

auto it = st::find(3);

but when we find the element which is not  
present in set, it would point <sup>after</sup> ~~to~~ the  
end element.

like

auto it = st::find(8);

1 1 2 1 3 | 4 | 5 ↑

if st::find(x) != st::end()  
element is present in set.

$\{1, 4, 5\}$

st::erase(5); it erases 5 O(logn)

int cnt = st::count(1);

if exists  $\text{cnt} = 1$  else  $\text{cnt} = 0$ .

To delete range of elements.

{1, 2, 3, 4, 5}.

auto it1 = st.find(2);

auto it2 = st.find(4);

st.erase(it1, it2);

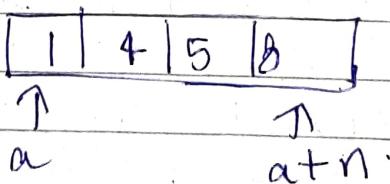
After erase {1, 4, 5} [first, last);

\* lower\_bound() & upper\_bound() functions

They work in the same way as in vector it does.

Check if x exists in sorted array array or not

bool res = binary\_search(a, a+n, 3);



- \* Multiset  $\rightarrow$  Similar as set but allows to store duplicate elements (but in sorted order)

`multiset < int > ms;`

`ms.insert(1); {1}`

`ms.insert(1); {1,1}`

`ms.insert(1); {1,1,1}`

`ms.erase(1);` || all the 1's get erased

only a single one erased.

`ms.erase(ms.find(1));`

After 3's : `ms.erase(ms.find(1), ms.find(1) + 2);`

To find the count the no in given set.

`int ont = ms.count(1)`

Rest all function are same as set.

- \* Unordered\_set  $\rightarrow$  same functionality as set
  - 1) Stores unique elements
  - 2) Doesn't store in sorted order
  - 3) Lower bound Upper bound is not applicable

\* Map - Stores elements in key & value pair.

{key, value}  
↓      ↓  
name   city

map <int, int> mpp ;

map <int, pair<int, int>> mpp ;

map <pair<int, int>, int> mpp ;

mpp[1] = 2 ;   // to initialise value

mpp.insert ({2, 4});

O/P

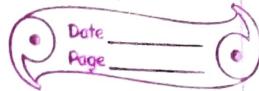
{1, 2}                  // stored in sorted  
{2, 4}                  order.

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

To access the value at given key.

cout << mpp[1];      // 2 .

map -  $O(\log n)$



find function works the same.

```
auto it = mpp.find(3);  
cout << *it << endl;
```

points to the position where 3 is found  
then prints the value.

lower bound & upper bound works as  
same.

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

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

\* Multimap - Stores a given key twice or  
more times (multiple key value pair.)

\* Unordered map - Stores elements all given  
 $O(1)$  key value pair in unsorted  
manner. But collisions could take  
place.

## \* Sort Algorithm

`sort(a, a+n);` // Sort in ascending order

for a given range

`sort(a+2, a+4);` [first, last)

`sort(a, a+n, greater<int>)` // Sort in descending order.

## \* Comparators

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

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

`sort(a, a+n, comp);`

Sorting it acc<sup>n</sup>. to 2nd element

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

- 1) If second element is same, then sort it according to the first element but in descending order.

\* built-in\_popcount() - Counts the no. of set bits in binary representation of a no.  
It only works for integers.

```
int num = 7; // 1111
```

```
int cnt = __builtin_popcount();
```

```
long long num = 165786578687;
```

```
int cnt = __builtin_popcount();
```

\* next\_permutation - It gives the next dictionary order. ↴ (returns true or false)

```
string s = "123";
```

```
do {
```

```
cout << s << endl;
```

```
} while (next_permutation(s.begin(), s.end()));
```

1) 123

1) 132

1) 213

1) 231

1) 312

1) 321

max element in a given range

```
int maxi = * max_element(a, a+n);
```