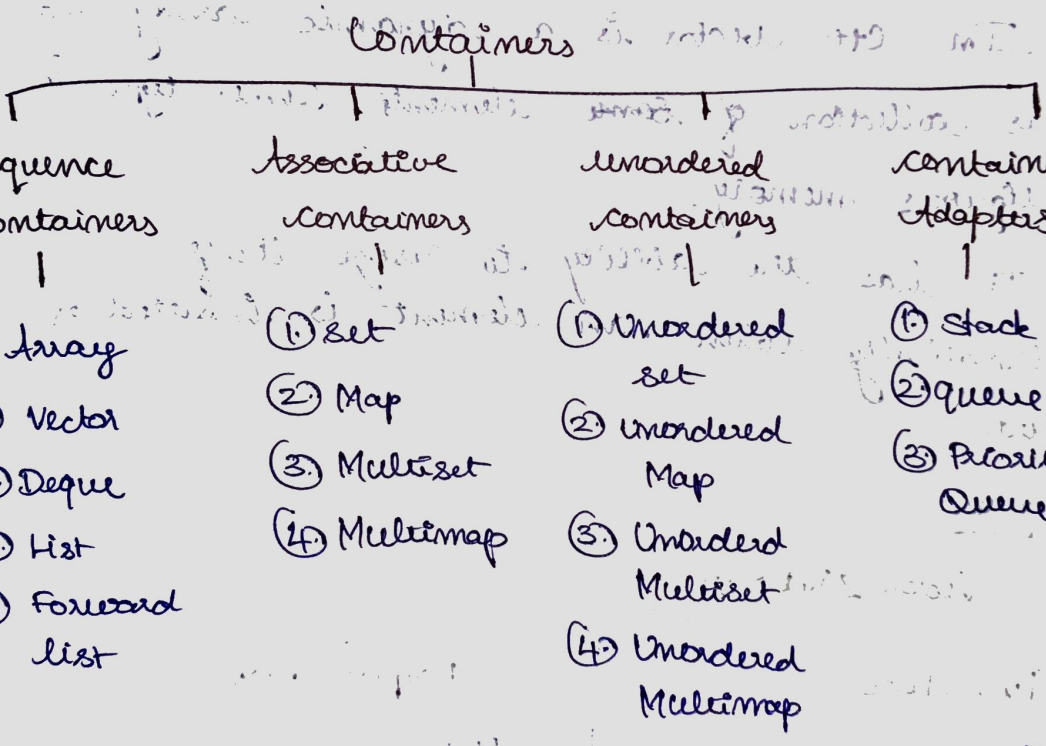
Standard Library Templete in C++

1. Algorithms
2. Containers
3. Function
4. Iterators

Containers:

* containers are powerfull tools in C++ for managing collections of data efficiently.



Pairs: [Utility library]

```
Void explainpair()
{
    pair<int, int> p = {1, 3};
```

```
cout << p.first << " " << p.second;
pair<int, pair<int, int>> p = {1, {3, 4}};
cout << p.first << " " << p.second.second
     << " " << p.second.first;
pair<int, int> arr[] = {{1,2}, {2,5}, {5,1}};
                        0      1      2
                        (Index)
cout << arr[1].second;
}
```

Vectors:

* In C++ vector is a dynamic array that stores collection of ~~same~~ elements / same type in contiguous memory.
* It has the ability to resize itself automatically when an element is inserted or deleted.

Syntax:

```
Vector<int> v;
```

Push_back	Emplace_back
* Slower	* Faster
* Syntax: Push_back (Value_to_Insert);	* Syntax: Emplace_back (Value to Insert)
* It accept the only object of the type if the constructor accept more than one arguments.	* It accept arguments of the constructor of the type.

```
void explainVector()
{
    Vector<int> v;
    v.push_back(1);
    v.emplace_back(2);

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

diff bt Push_back & Emplace_back

Vector <int> V(5,100); ⇒ {100, 100, 100, 100, 100} = [5]
(0, 1, 2, 3, 4)

Vector <int> V(5); ⇒ {0, 0, 0, 0, 0}

Vector <int> V1(5,20); {20, 20, 20, 20, 20}

Vector <int> V2(V1); V2 = V1

Iterators in C++:

* It is an object like a pointer to points to an element inside the container.
* we can use iterators to move through the contents of the container.

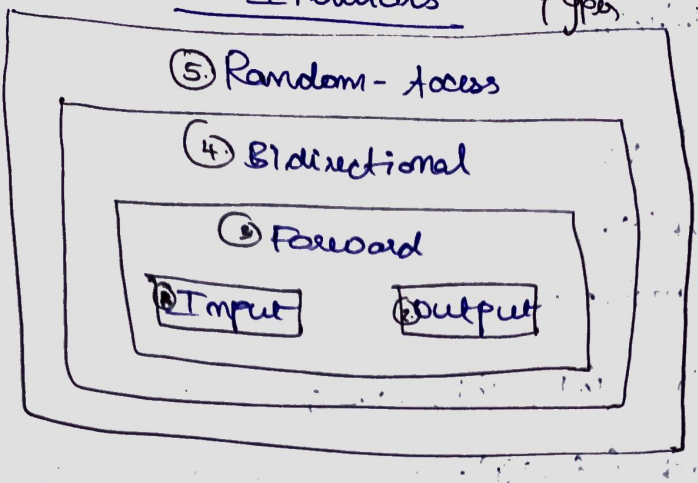
~~Vector<~~

Syntax:

CType :: Iterator IteratorName;

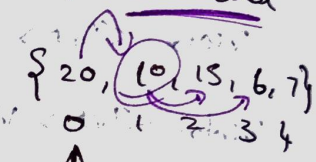
C-Type — container Type

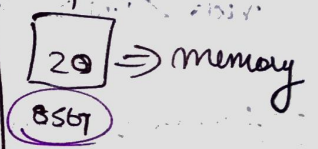
Iterators Types




```
Vector<int>::iterator it V.begin();
it++;
cout << *(it) << " ";
it = it + 2;
cout << *(it) << " "; // It print 6
```

Eg: Iterator





* ⇒ Indicate memory

V.begin() ⇒ it Points the memory address

it ++ ⇒ 10

```
Vector<int>::iterator it = V.end();
```

Eg: {10, 20, 30, 40} ↑

It Print the value after 40

```
Vector<int>::iterator it = V.rend();
```

(reverse end)

Eg: {10, 20, 30, 40}

↑ {40, 30, 20, 10}

↑

V.rend() It Print value before 10

```
Vector<int>::iterator it = V.rbegin();
```

Eg: {10, 20, 30, 40}

↑

V.rbegin() will print value 40

v.begin();

it ++

{10, 20, 30, 40}

It will increment to next.

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

cout << v.back() << " ";

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

(auto) is a keyword that automatically takes the datatype.

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

Erase():

v.erase(v.begin()+1); {10, 20, 12, 23}

⇒ {10, 12, 23}

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

{10, 20, 30, 40, 50} ⇒ [10, 20, 50]

Insert function:

```
Vector<int> v(2,100);
```

Output:

{100, 100}

```
v.insert(v.begin(), 300)
```

Output:

{300, 100, 100}

```
v.insert(v.begin()+1, 2, 10)
```

Output:

{300, 10, 10, 100, 100}

copy():

```
Vector<int> copy(2,50)
```

Output:

{50, 50}

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

Output:

{50, 50, 300, 10, 10, 100, 100}.

Size():

```
cout<<v.size();
```

Pop_back():

```
v.pop_back();
```

Eg: {10, 20}

```
v.pop_back();
```

{10}

swap:

Eg: $V_1 \rightarrow \{10, 20\}$

$V_2 \rightarrow \{30, 40\}$

```
V1.swap(V2);
```

output:

$V_1 = \{30, 40\}$

$V_2 = \{10, 20\}$

clear():

It erases the entire Vector

```
V.clear();

cout<<V.empty();
```

1 Vector can be empty.

List:

* List containers implements a doubly linked list in which each element contains the address of next and Previous element in the list.

* It stores data in non-contiguous memory hence providing fast insertion and deletion once the position of the element is known.

* It is similar to Vector.

Syntax:

```
list<T> l;
```

T - Type of element in the list

L - Name assigned to the list

Eg:

```
Void explainlist()
{
    list < int > ls;
    ls. push_back (2);
```

Output:

{2}

```
ls. Push_front (4);
```

Output:

{4, 2}

```
ls. emplace_back (5);
```

Output:

{4, 2, 5}

```
ls. Impleace_front(); {2, 4}
// rest functions same as vector
// bigin, end, rbegin, rend, clear, insert, size, sleep.
```

Deque:

* Deque container provides fast insertion and deletion at both ends.

* Deque - stands for Double Ended Queue.

* It is a special type of the queue where the insertion and deletion operations are possible at both the ends in constant time complexity.

```
Void explainDeque()
{
    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();
    dq.front();

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

Syntax for deque:

```
deque<T> dq;
```

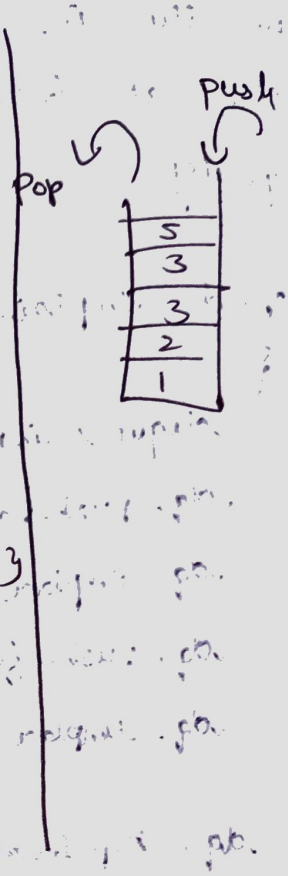
T → Type of the element

dq → name assigned to the deque.

Stack: [Last In First Out] LIFO


```
Void explainStack()
{
    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(); => Print 5
    st.pop();        => delete 5
    cout << st.top(); => Print 3
    cout << st.size(); => 4
    cout << st.empty; => False

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



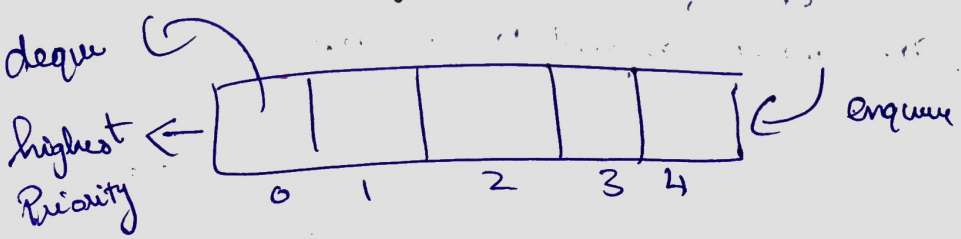
Syntax:

```
stack<int> st;
```

Queue: [First In First Out] FIFO

Syntax:

```
queue<T> q;
T → datatypes
q → name assigned
```



```
void explain_Queue()
{
    queue <int>q;
    q. Push(1);      ⇒ {1}
    q. Push(2);      ⇒ {1,2}
    q. emplace(4);   ⇒ {1,2,4}
    q. back() += 5   ⇒ 9 ⇒ {1,2,9}
                        ↳ +5
    cout << q. front(); ⇒ 1
    cout << q. front(); ⇒ Print 1
    q. pop();        {2,9}
    cout << q. front(); ⇒ Print -2
// Size, swap, empty same as stack
```

Priority Queue:

* In C++, Priority Queue is a type of queue in which there is some priority assigned to the elements.

* According to this Priority, elements are removed from the queue.

Syntax:

```
Priority-queue<T, c, comp> Pq;
```

T- Type of the Priority Queue

Pq - name assigned.

c - underlying container. Uses Vector as default.

Comp - It is a binary Predicate function that tells Priority Queue how to compare two elements.

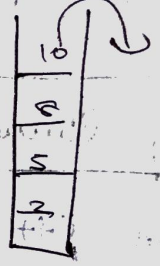
Internal working

Max - heap (default)
Priority is given to the largest element

Min - heap
Priority is given to the smallest element.

```
Void explainPQ()
{
    Priority - queue <int> pq;        // Max heap
    pq.push(5);  => {5}
    pq.push(2);  => {2,5}
    pq.push(8);  => {8,5,2}
    pq.emplace(10);  => {10,8,5,2}
    cout << pq.top();  => {10}
    pq.pop();  => delete 10
    cout << pq.top();  => {8,5,2}
    Priority - Queue <int, Vector<int>, greater<int>> pq;   // Min heap
    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
};
```

(Max Descending order)



Time complexity:

Push → $\log n$

top → $O(1)$

Pop → $\log n$

Set:

* It stores the collection of distinct elements.

Implementation

Hash - based set Tree - Based Set

Types

Ordered (hash) Unordered (BST)

Set

* Stored in sorted order
* Unique values, no duplicate

Syntax:

```
Set<T, comp> s;
```

T → data type of elements in the set

s → name assigned to the set

comp → It is a binary predicate function that tells set how to compare two elements.

Set:

```
Void explainset()
{
    set.insert(1);
    set.emplace(2);
```

`set<int> st;`

st.insert(1); ⇒ {1}

set.emplace(2); ⇒ {2}

set.insert(2); ⇒ {1, 2}

set.insert(4); ⇒ {1, 2, 4}

set.insert(3); ⇒ {1, 2, 3, 4}

// functionality of insert in vector
// can be used also, that only
// Increases efficiency.

/* begin(), end(), rbegin(), rend(), size(), empty() and swap() are same as those of above */.

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

auto it = st.find(6); ⇒ st.end()

st.erase(5); ⇒ {1, 2, 3, 4}

int cnt = st.count(1);

upper bound - lower bound. [watch vdo]

Multiset:

Void explainmulset()

{

multiset<int> ms;

⇒ sorted
⇒ not unique
⇒ stores duplicates

ms.insert(1); ⇒ {1}
ms.insert(1); ⇒ {1,1}
ms.insert(1); ⇒ {1,1,1}
ms.erase(1); ⇒ erased all 1's

int cnt = ms.count(1);
ms.erase(ms.find(1)); ⇒ Only one (1) is erased.
↳ address
ms.erase(ms.find(1), ms.find(1)+2);
// rest all functions same as set.

}

Unordered set:

* Unique
* Unsorted.

Time complexity
O(1)

Syntax:

unordered_set<T> us;

T → datatype
us → name assigned

/* lower bound and upper bound functions does not works, rest all functions are same as above, it does not stores in any particular order it has a better complexity than set in most cases, except some when collision happens */

Map:

Unique values, Sorted Key

* It store data in the form of key value pairs
* Sorted on the basis of keys
* No two mapped values can have the same keys.
* By default it in ascending order.

Syntax:

```
map < key-type , Value_type, comp > m;
```

key type – data type of key
Value type – Data type of value
comp – optional
m → name assigned

```
Void explainMap()
{
    map<int, int> mpp;
    map<int, pair<int, int>> mpp;
    map<pair<int, int>, int> mpp;

    mpp[1] = 2;
    mpp.emplace({3,1});   mpp.emplace({2,3})
    for(auto it : mpp)
    {
        cout << it.first << " " << it.second <<
                                    endl;
    }
    cout << mpp[1];
    cout << mpp[5];
```

Multimap:

* duplicated keys
* sorted

Unordered Map: