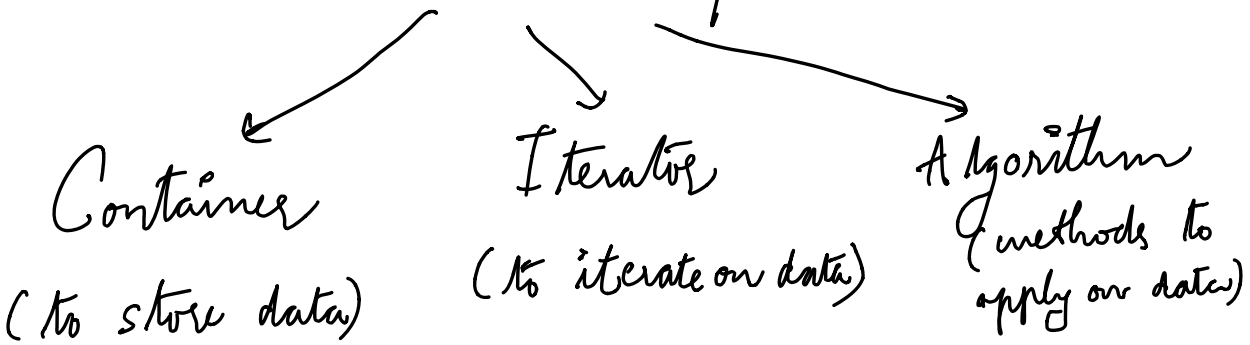
STL

STL : Standard Template Library



(1) std::array #include<array>

Syntax: std::array < T, N > array;

↳ encapsulates fixed size array

→ size is needed at compile Time

→ In pass by value, we need not provide size i.e., fun(array); ! at implementation

fun(array) { array.size(); }

↓ will give size

i.e, like traditional array it'll not decay i.e; needn't pass size

ELEMENT ACCESS

- → at() } similar but at() throws exception in out-of-bound case.
- → [] }
- → front()
- → back()
- → data() → gives access to underlying array.

Initialization

1) array<int, 3> a = {1, 2, 3}; //initializer list
2) array<int, 3> a {1, 2, 3}; //uniform initialization
3) array<int, 3> a;
   a = {1, 2, 3};

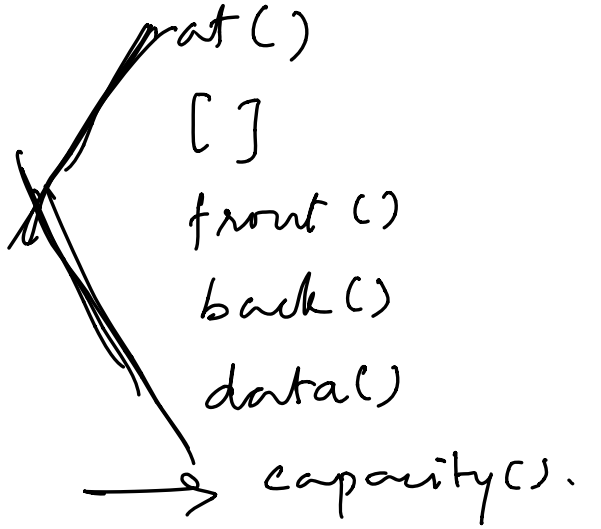
in traditional
array[5];
array = [1, 2, 3, 4, 5];

→ also a.fill(10) → fills 10 in all positions of a. #CHECK_OUT_MODIFIERS

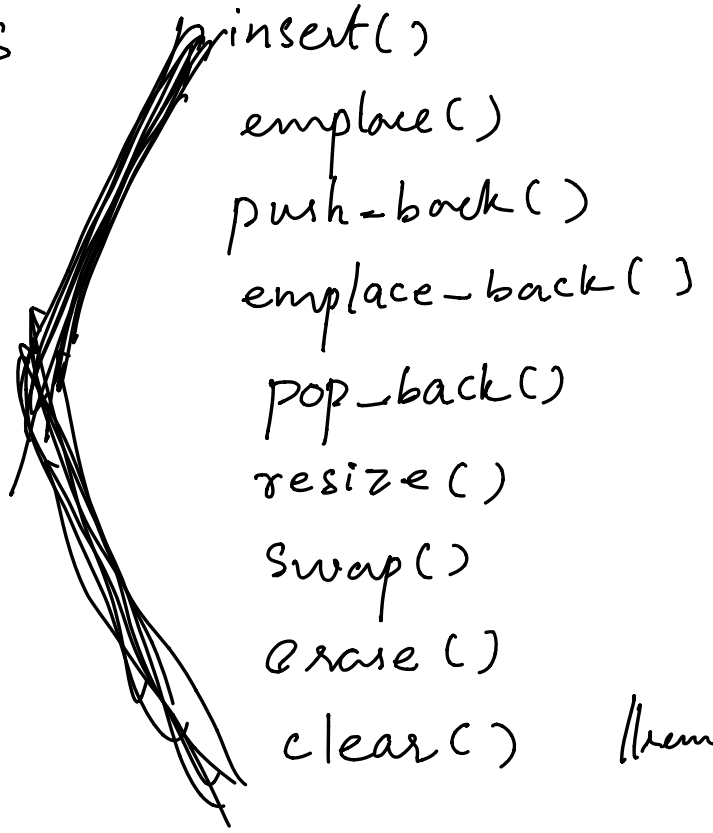
(2) std::vector #include<vector>

→ Also known as Dynamic Array or Array List.

→ ELEMENT_ACCESS



→ MODIFIERS



→ Syntax: std :: vector<T> vec;

... List & uniform initialization

→ initialization list (...)

-ion possible.

size ↓ element in all positions ↓

also vector<int> v (5, 10);

→ Every time we wanna push something to vector if (size == capacity) then the capacity doubles to accomodate incoming, this goes on;

This process may take some time, so to avoid this i.e, when pushing back the vector creates another double size array and copies its element there & then push'es back.

→ Soln to this : v.reserve (size);

i.e, we reserve max possible size for our vector to save time.

set
multiset
map

③ std::set() #include<set>

multimap — data is associated with something else

→ Syntax: set<T> name;

→ Associative container that contains a sorted set of unique objects of type key.

→ usually implemented using red-black tree.

→ Insertion, Removal, Search → O(log n)

→ Default, ascending order.

→ Ex:

```
In main()
set<int> Set = {1,2,5,4,3,1,2,3,4,5};
for (const auto& e : Set){
        cout<<e<<endl;
}
return 0;
```

Output:

```
1
2
3
4
5
```

→ To store user defined data type in set we need to pass the comparator too (overload comp. operator)

i.e., Ex: class Person {

public: float age;

string name;

bool operator < (const Person & p2) const

{ return age < p2.age }

};

main()

{ set<Person> S1 = { {30, "A"}, {25, "B"}, {20, "C"} };

for (const auto & e : S1) {

cout << e.age << " " << e.name << "\n";

}

}

output:

20 C

25 B

30 A

To get in descending order,

In class Person:

```
bool operator > (const Person &p2) const
  { return age > p2.age ; }
```

⇒ `set <Person, greater<> > S1 = { ——— };`

↓

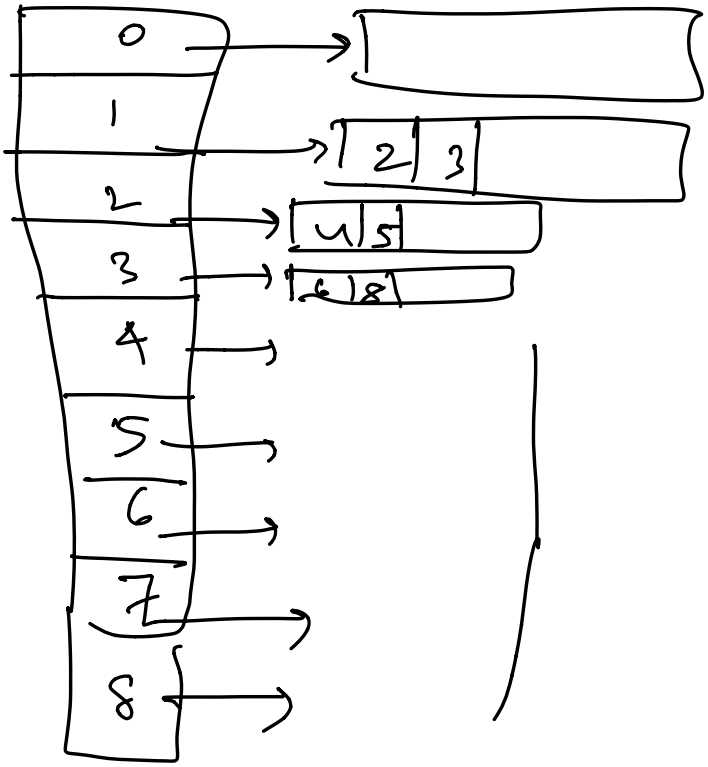
comes from header ⇒ include <functional>

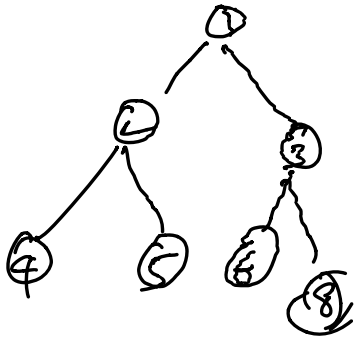
↳ can also mention `set<Person, greater<Person>>`

Modifiers / Lookups

④ Container Inside Container

Ex: vector< vector <int> > Tree;





take no. of edges input : 6

enter edges:

1 2
1 3
2 4
2 5
3 6
3 8

Print

2 3 4 5 6 8

for (const auto &e: Tree) {

```
for (const auto &e1 : e){
        cout << e1 ;
}
cout << endl;
}
```

We can combine any container inside another container to get special work to be done.

(5) std:: multiset < T > #include<set>

Syntax: multiset < T > name;

→ has elements in sorted order and duplicates are allowed.

→ rest similar to set //.

Modifiers / Lookups

(6) std:: map #include<map>

~#include<functional> for greater, less

→ Syntax: map<T1, T2> obj;

→ Associative container that stores elements in key-value combination

→ key should be unique, else overrides prev. value.

→ Implemented using Self Balancing Binary search Tree (AVL/ Red Black)

→ Stores key-value in sorted (A/D) on key basis.

→ Like dictionary of python.

Ex: map<string, int> M;

```
M["a"] = 1;
M["b"] = 2;
M.insert(make_pair("c", 3));
// Loop through Map
for (auto &e : M)
    cout << e.first << " " << e.second;
// Access through []
cout << M["a"] << endl;
```

make_pair (here) ≡ pair<char, int> ('c', 3);

→ In case of user defined data type say class Person, we need to pass comparator too.

Ex:
```
map<string, vector<int>> M;
M["a"].push_back(1);   // push_back anytimes ✓
M["b"].push_back(2);
for (auto &e : M) {
    cout << e.first;
```

```
        for(auto &e1 : e.second){
            cout << e1;
        }
        cout << endl;
    }
```

→ If we've a dictionary type of problem and key is non-numeric then vector is fastest and map is second fastest.

Modifiers / Lock-ups

(7) std::multimap #include<map>

→ diff from map is that it allows multiple entries for same key.

Syntax: multimap<T1, T2> M;

auto range = M.equal_range('a');

→ a pair of pointers (a,1) ← first
(a,2)
(a,7) ← second

to print: for(auto it = range.first; it != range.second; ++it)
{ cout<<it.first<<" "<<it.second; }

Lookup: count, find, contains, equal_range, lower_bound, upper_bound.

1) M.count('a'); 2) M.contains('a') → T/F

3) M.find('a') → returns pair → any of key-val pair of a.
Ex: a-1 (for a-1, a-2, a-3)

4) M.lower_bound('a') → returns pair of lowest (a-1) say (1st of a's)

a 1

a 2

5) M.upper_bound('a') in
given [b | 4] pair

a 3
b 4
b 5

Instead of multi_map we can go for map < T1, vector<int> > M;
:-e; if one of T1 has many int's.

Modifiers:

(8) Emplace in STL

→ All containers support insert and emplace operation.

→ Emplace is used to construct object in-place and avoid unnecessary copy of objects

→ Insert and Emplace is equal for primitive data types but when we deal with heavy objects we should use emplace if we can for efficiency.

Try:

```
class A {
    public:
    int x;
    A(int x=0) : x(x) { cout<<"Construct"<<endl;};
    A(const A& rhs) { x = rhs.x; cout<<"Copy"<<endl;}
};
int main() {
    set<A> Set;
    // A a[10];
```

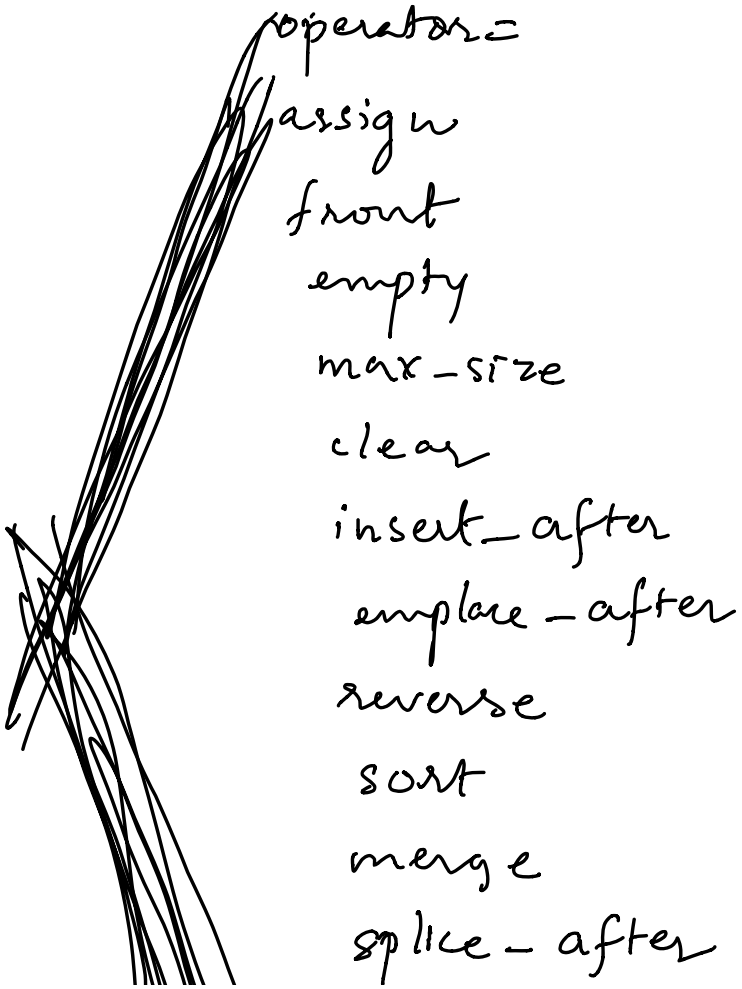
```
set.insert(A[10]);
set.emplace(10);
}
```

(9) std :: forward_list #include<>

→ This is single linked list.

→ We have many functions available on this forward list & is well tested.

Func's



unique
remove
remove_if
resize

Try: forward_list<int> l1 = {5,2,4,6,2};

```
l1.remove_if
( [] (int n) { return n>4; } );
```

(10) std:: list #include<list>

→ double linked list. [SEQUENCE CONTAINER]

→ faster than {vector, forward_list, deque} interms of insertion, removal and moving elements in any position provided we've the iterator of the operation.

Funcs

assign, operator =
front, back
empty, size, max_size
clear

list<int> l1;

Syntax;

insert, emplace |
push_back, pop_back
push_front, pop_front
reverse
sort, merge, splice
unique
remove
remove_if
resize

(11) std::pair

Syntax: pair < T1, T2 > obj;

⇒ pair is a struct template that provides a way to store two heterogeneous objects as a single unit.

⇒ map, multimap, unorder_map, unorder_multimap can use pair to insert data into their structures.

Ex1

```
template <typename T1, typename T2>
void print ( pair<T1, T2>& obj )
{   cout << obj.first << " "
          << obj.second << endl; }

int main()
{   pair <int, int> obj1(10,20);
    pair <int, int> obj2 = make_pair(10,20);

 vector < pair < string, uint > > list;
 list.push_back (make_pair ("A", 1));
 list.push_back ( pair <string, int> ("B", 2));
 list.push_back ( pair ("C", 2));
  for (auto & e : list) {
      cout << e.first << " " << e.second ;
   }
}
```

← doesn't work everywhere (only adv. compilers)

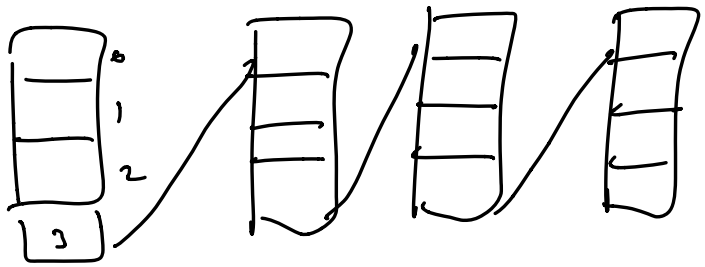
(12) std::deque #include <deque>

Indexed sequence container

→ unlike vector, elements are not stored contiguous in deque.

Ex:

Deque



→ Expansion of deque is cheaper than the vector expansion.

→ A deque holding just one element has to allocate its full internal array. So, for few elements vector is preferable over deque.

→ ~~ff~~has push_front(), pop_back() extra compared to queue stl.

(13) std::queue() #include <queue>

Syntax: queue < T1 > name;

→ push at back, pop from front.

→ front, back, push, pop, empty, size.

(14) std::stack() #include <stack>

→ stack & queue are container adapter i.e., they adapt deque and map functions.

→ push & pop from back

→ empty, size, top, push, pop -

→ To change its internal container from deque to vector just say

stack <int, vector<int> > stk;

(15) std :: priority_queue #include <queue>

→ a container adapter that provides constant time lookup of the largest or smallest element.

→ by default, vector is the container used inside.

→ Insertion, Extraction → O(log n)

→ Implemented using make_heap, push_heap, pop_heap functions.
(# see, sort_heap, is_heap, is_heap_until);

custom comparator.

Using lambda to compare elements.

```
auto cmp = [] (int left, int right)
           { return (left) < (right); };

priority_queue <int, vector<int>, decltype(cmp)>
               q, (cmp);
```

```
for (int elm : {1,8,5,6,3}) {q.push(elm);}
print(q);  → 8 6 5 3 1
```

(16) std:: unordered_set #include <>

→ Contains unique element but not not sorted

→ associative container.

→ search, insertion & removal have average const-time complexity.

→ Internally, the elements are organized into buckets

→ It uses hashing to insert elements into buckets.

→ Why Unordered_set: ?

Maintain a collection of unique items with fast insertion & removal.

(17) std:: unordered_multiset #include <unordered_set>

→ same has unordered_set except that it allows non-unique objects.

→ use can be to find in container fast, as search, insert, remove are average constant time cuz of has _ling. To find, count repetetion etc..;

→ Ex:

```
unordered_multiset<int> ums;
ums= {4,1,2,3,9,4,3,2,9,8,10};
auto search= ums.find(2);
if (search != ums.end())
        cout<< "Found" << (*search);
else
        cout << "Not Found";
```

```
for (auto &ele : ums)
    cout << ele ;
```

(18) std:: unordered_map #include< > >

→ key-value pairs are unique

→ Elements are stored in buckets and are inserted using hashing.

→ Search, remove, insert → avg. const. time

→ key-value pairs aren't SORTED.

(19) std:: unordered_multimap #include <unordered_map>

→ same as unordered_map but key-val pairs mayn't be unique

→ Use:

Maintain a collection of duplicate {key:val} pairs with fast insertion and removal.

(20) STL Algorithm #include<algorithm>

→ algorithms are applied to range of elements

Ex:

```
auto it = find (vec.begin(), vec.end(), 4);
                (RHS)
```

↓

pos is returned (of 4)

a) std:: sort

→ can even be used to sort user defined data types.

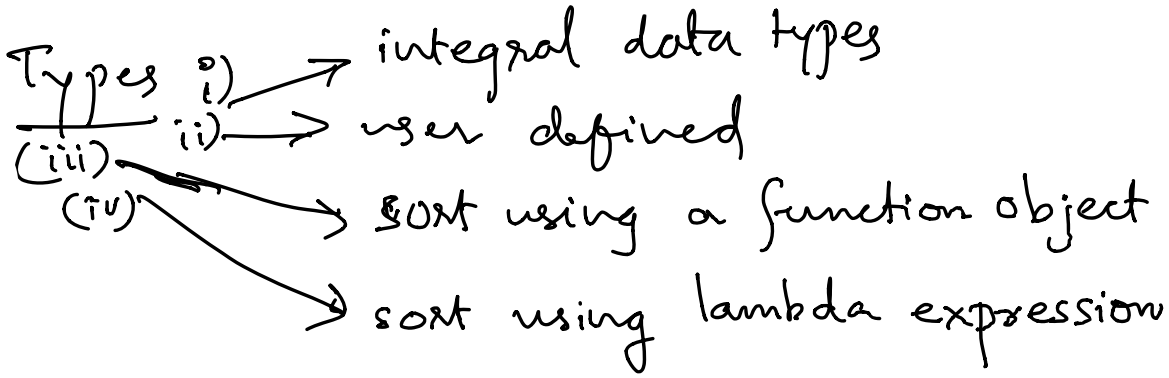
→ Internally, it uses IntroSort which is combo of Quick, Heap & Insertion sort.

→ By default, it uses Quick but if Quick is doing unfair partitioning & taking more time than N log N, then it switches to HeapSort and when it's size becomes really

the array [illegible] small, it switches to Insertion sort.

→ We can use parallel execution policy for better performance.

↳ #include <execution>



i) Ex: sort(execution::par, vec.begin(), vec.end());

ii) Ex:

```
class Point {
    int x, y;
    Point(int x=0, int y=0) : x(x), y(y) { }
    bool operator < (const Point& p1)
    {   return x+y < (p1.x+p1.y); }
};
```

→ vector<Point> Vec {{1,2}, {3,1}, {0,1}};

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

(iii)
```
struct {
  bool operator() (int a, int b)
  {   return a < b;
  }
} customLess;
```

(iv) Lambda (Imp ~~**~~ use this)

```
sort (vec.begin(), vec.end(),
        [] (int a, int b) { return a<b; });
```

b) std:: is_sorted()

is_sorted (vec.begin(), vec.end(),) ;

Same_as sort() ✓

c) std:: partial_sort()

Ex: { 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };

partial_sort (rec.begin(), rec.begin()+4, rec.end());

output: 0 1 2 3 8 7 6 9 5 4

consider whole array but sorts only first four positions.

also say, p_s (begin, begin+4, end-2)

d) std:: nth_element

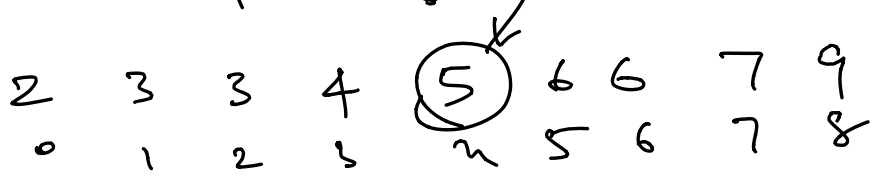
→ uses introselect = hybrid(quickselect + median of medians algo's)

→ Ex: vector<int> v { 5, 6, 4, 3, 2, 6, 7, 9, 3 };

nth_element (v.begin(), v.begin()+ v.size()/2 ,

```
                v.end() );
cout << "The median is" << v[v.size()/2];
```

i.e, here it's like if the list would have been sorted its v.size()/2 element would be ?



so median is 5 //.

→ It doesn't sort the list it's like quick sort partitioning. i.e, all the element before 5 are less than it and vice-versa

→ To get second largest element

```
nth_element(v.begin(), v.begin()+1,
            v.end(), greater<int>());
```

Output here: 7

e) std:: is_sorted_until

Ex:

```
auto it = is_sorted_until (Vec.begin(), Vec.end());
    auto diff = distance (vec.begin(), it );
```

→ comparator function can be used.

f) std:: iota #include<numeric>

→ Fills the range {first, last} with sequentially increasing values, starting with value and repetitively evaluating ++value.

Ex:

```
#include<algorithm>
#include<iostream>
#include<list>
```

```
#include <numeric>
#include <vector>
using namespace std;
int main()
{
    list<int> lst(10);
    iota(lst.begin(), lst.end(), 0);
    print(lst);
vector<list<int>::iterator> Vec(lst.size());
    iota(Vec.begin(), Vec.end(), lst.begin());
    for(auto ele: Vec) cout << *ele;
}
```

output: 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

Try Libraries (coding)

→ <iterator>, <numeric>,
<regex>, <snprintf>, <strings>

g) copy & copy-if

Ex: ⇒ v1(10);

iota(begin, end, 0);

v2 (v1.size());

copy(v1.begin, v1.end , v2.begin)

(or)

v2;

copy (v1.begin, v1.end, back_inserter(v2));

(OR)

v2 = v1;

I are similar but copy's use comes when we've to copy partially.

⇒ v3(5);

```
copy_if ( v1.begin , v1.end , v3.begin ,
        [] (int x) { return (x%2)==1;
        } );
```

or for odd number we can say

```
[] (int x) { return (x&1); };
```

h) copy_n and copy_backward

Ex:

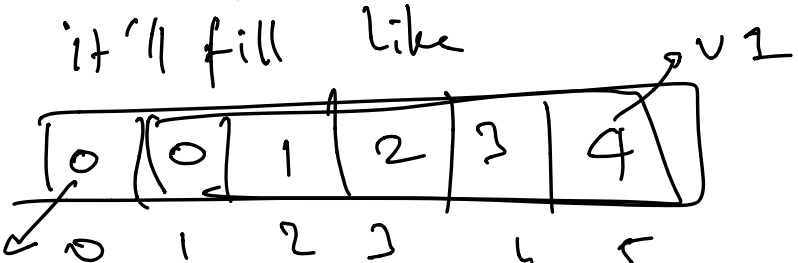
```
v1(5);
iota ( v1.begin, v1.end, 0);
v2; v3(6);
copy_n ( v1.begin , 4, back_inserter(v2));
```

↳ 0, 1, 2, 3 ; ✓

```
copy_backward
( v1.begin , v1.end , v3.end );
```

↳ 0 0 1 2 3 4

i.e, it'll fill like



unfilled is put '0'.

4 ←

3 4

2 3 4

1 2 3 4

0 2 3 4