STL Introduction in C++

→ STL: Standard Template Library.
→ It consist of three main components.

a. Container : ~~Stores~~
b. Iterator
c. Algorithm.

→ Container: Stores the data.
e.g: Vector, Array, Linked List, map, etc.

→ Iterator: Iterator is used to Iterate elements over container.
It is used to access the elements.
There are different iterator based on the containers used.

→ As for linked list we access element sequentiely whereas in the array we access randomly any element.

→ ~~Containers~~: Algorithm: There are different algorithms to perform different operations such as, Searching, sorting, Reverse, etc.

* Containers:

I.Q. Q. Why do we need Array if we have vector?

① Std :: array :

Syntax:

Traditional C++ : int arr [5];

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

→ Here basically it is a array class with associated member functions. and we are kind of creating object of array.

```
std:: array < int, 5 > arr;

Class array
{
    T array [N];
    ⋮
}
```

(we are passing pointer address of a array. address of first element)

⇒ In tradional C++ array we pass array to a function then there is a problem of getting size of array in that function. But if we pass stl array then it is like we are passing object and we can easily call member function.

arr.size (); to get a size.

Point:

→ std:: array is a container that encapsulates fixed size arrays.
→ array size is needed at compile time.
→ Assign by value is actually by value.
→ we can Access Elements:

1. .at () : cout << arr.at (3);

If the array is of size 5 and we are accessing like cout << arr.at (7) then this will throw " Out of Bound Exception".

2. [] : cout << arr [3]; → It will give garbage value.
→ This will not throw Out of Bound Exception.
→ This is overloaded Operator.

3. front () : cout<array.front () → will give first element.

4. back () : cout << array.back() → will give last element.

5. data () : gives access to the Underlying array
(Pointer or address to the array present in the class)

Code:

```
# include <iostream>
# include <array >
 using namespace std;

int main ()
{   //Declare
     std:: array < int, 5 > myarray;
  // Initialization
std:: array < int, 5 > myarray = { 1, 2, 3, 4, 5 }   //Initializer list
std:: array <int, 5> myarray 2= {1, 2, 3, 4, 5 }   //uniform initialization

 //Assign Using initializer list
 std::array <int, 5 > myarray;
 my array = {1, 2, 3, 4, 5 };

 return 0;
}
```

This type of initialization is not possible in Tradional c++

```
int array [5];
array = {1, 2, 3, 4, 5};
```

② std :: vector

Q. why vector name is given to vector?
Q. why we need vector if we have linked list.

→ Array gives faster access to element of O(1). while in linked list we have to transverse the element O(n).

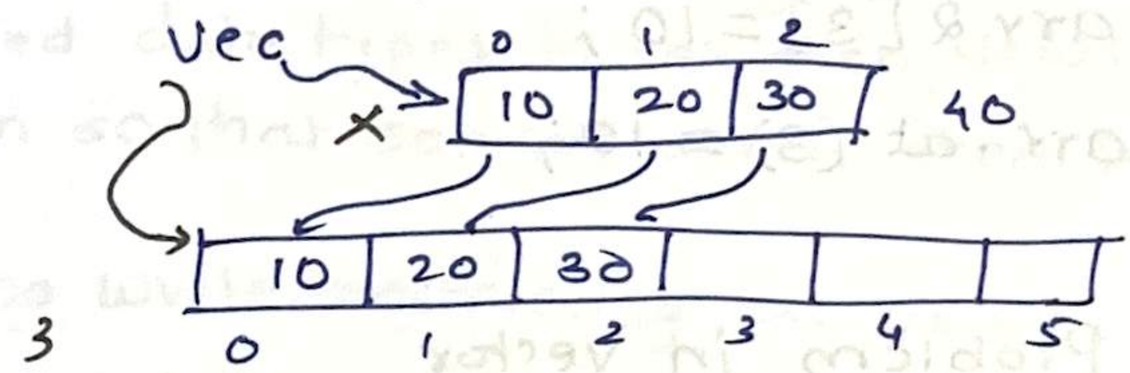
→ But array have fix size but linked list can grow to any size.

→ Hence to give the advantage of both array and linked list. i.e. Faster access O(1) and no fix size vector was introduced.

→ In vector a double size ~~array~~ vector array is created inside a class when the capacity to hold the element is not possible.
→ It copies all element in next vector (array)

e.g:
```
std:: vector <int > vec;
vec.push_back (10);
      ——"—— (20);
      ——"—— (30);
      ——"—— (40);
```

```
class vector
{
```



Q. How vector works internally in c++? (CppNuts)

~~Header~~ Notes:

1. std :: vector is a sequence container and also known as Dynamic Array or Array List.
2. It's size can grow and shrink dynamically, and no need to provide size at compile time.

// Element Access
at (), [], front (), back (), data ()

// Modifiers:
insert ()
emplace ()
push_back ()
emplace_back ()
pop_back ()
resize ()
swap ()
erase ()
clear ()

Code

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
// Declarations
std::vector<int> arr1;
std::vector<int> vec2{5,20}      // creat vector of size of 5 and fill 20 at every place.
std::vector<int> vec3 = {1,2,3,4,5}   // Initializer list
std::vector<int> vec4 {1,2,3,4,5}   // Uniform initialization

// Accessing Elements
arr2[3]=10;
arr.at(3)=10;
}
```

syntax:

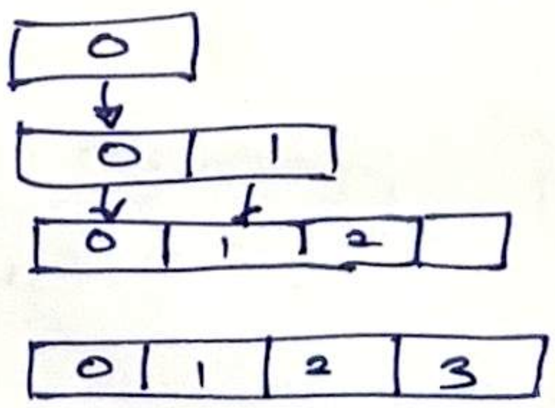
```
std::vector<T> vec1;
```

⇒ Problem in vector

```
int main()
{
 vector<int> vec;
 for(int i=0; i<32; i++)
 {
  vec.push_back(i);
 cout<< vec.size()<<" "<<vec.capacity()<<endl;
 }
 return 0;
}
```

⇒

size: 1	capacity: 1
size: 2	capacity: 2
size: 3	capacity: 4
size: 4	capacity: 4
size: 5	capacity: 8
⋮	⋮
size: 8	capacity: 8
size: 9	capacity: 16
size: 10	capacity: 16



⇒ ~~size~~ ~~Capacity~~ If size is greater than capacity they internally double size array is created and all the previous array elements are copied in new array.

⇒ This copying the elements consumes time every time. so this is a problem in the vector.

⇒ So depending on the elements we can initially reserve (create) some array and it can save the time of copying.

```
vec.reserve(1000);
```

③ Container Set:

⇒ This set is similar to set we studied in school.

1 2 1 5 → 1 2 5
9 5 → 9

→ set do not allows duplicate entries and keep data in sorted order.

→ Default is Ascending Order.

⇒ Syntax: std::set < T > ObjectName ;

⇒ Notes:

1. std::set is an Associative container that contains a sorted set of unique objects of type key Q. what is mean by Associative container
2. It is usually implemented using Red-Black Tree.
3. Insertion, Removal, search have logrithmic complexity.
4. If we want to store user defined data types in set then we will have to provide compare function so that set can stored them in sorted order.
5. we can pass the order of sorting while constructing the object.

e.g: set < int , std::greater > set 1 ← Descending order
set < int , std::less > set 2 ← Ascending order

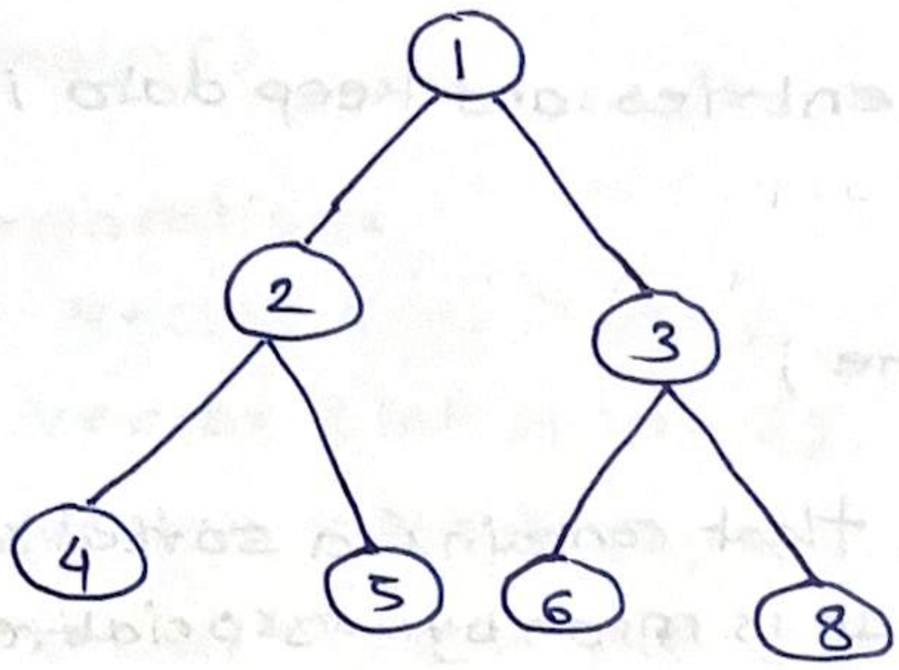
```
# include <iostream>
# include <set>
# include <string>
# include <functional>
using namespace std;
int main()
{
  set <int> set = {1,2,3,5,4,3,1,2,5,4}
  for (const auto & e : set)
    {
      cout << e << endl;
    }
}
```

set for integer
No need to define compare function.

```
class Person {
  public:
      float age ; string name ;
      bool operator < (const Person & rhs) const {return age < rhs.age;}
}
int main()
{ set <Person> Set2 = {{30,"Rupesh"}, {25,"Hitesh"}, {22,"John"}};
  for (const auto & e : Set)
    {
      cout << e.age << " " << e.name << endl;
}
```

* Container Inside Container

→ Generally we use linked list for the tree
→ Lets use vector inside vector for the tree.

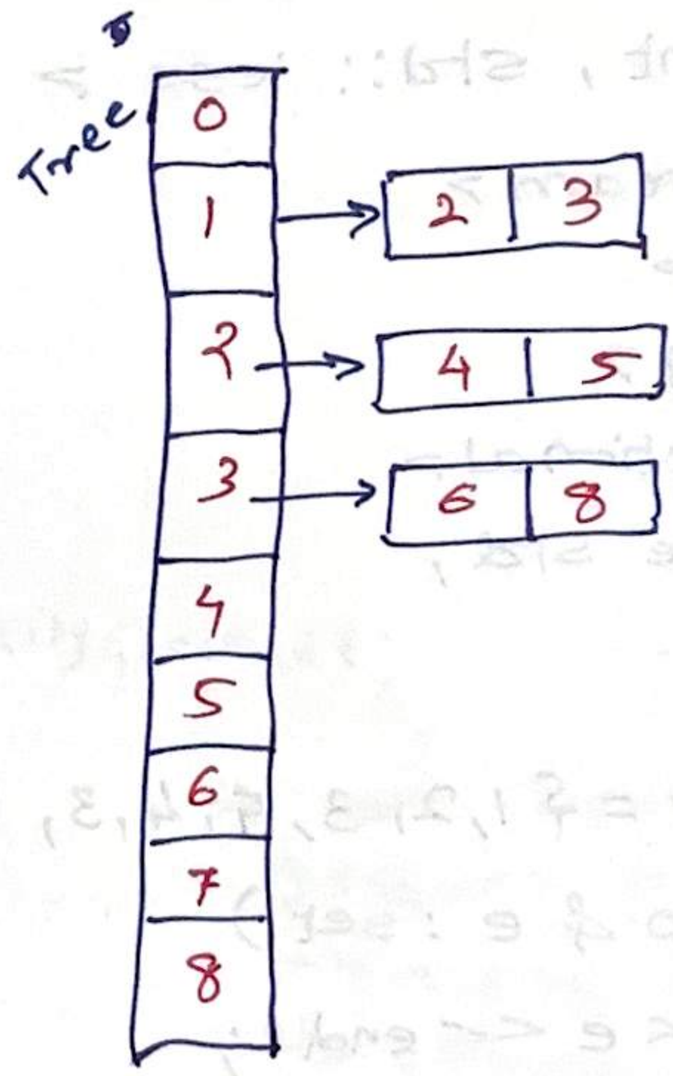


Edges	n_1	n_2
1 - 2		
1 - 3		
2 - 4		
2 - 5		
3 - 6		
3 - 8		

No. of edges
edge = 6

This n_1 & n_2 will be input in our program.

```
# include <iostream>
# include <vector>
using namespace std;

int main()
{
  std:: vector < vector <int >> Tree;
   int edge, n1, n2;
   Cin >> edge;
   Tree. resize (edge);
   for (int i=0; i<edge; i++)
   {  cin >> n1 >> n2;
     Tree [n1]. push_back (n2);
   }
   for (const auto &e1 : Tree)
   {
      for( const auto &e2 : e1)
      {
         cout <<e2;
      }
      cout <<endl;
   }
   return 0;
}
```



(4) Container : Multiset.

Syntax : std::multiset < T > ObjectName;

1. std::multiset is an Associative container that contains a sorted set of duplicate objects of type key.
2. It is usually implemented using Red-Black Tree.
3. Insertion, Removal, search have logarithmic complexity.
4. If we want to store user defined data in multiset then we will have to provide compare function so that multiset can store them in sorted order.
5. We can pass the order of sorting while constructing mult set object.

⇒ Multiset is similar to set except it can have multiple elements with same value.

```
#include <iostream>
#include < set >
#include <string>
#include <functional>
Using namespace std;
int main()
{ std::multiset <int> multiset 1 = {5,2,4,3,2,5} ← Default Ascending
  Std::multiset <int, std::less> multiset 2 = {5,2,4,3,2,5} ← Ascending
  Std::multiset < int, Std::greater> m.s. 3 = {5,2,4,3,2,5} ← Descending order.
  for (const auto& e: multiset 1)
        cout << e << endl;
}
// For user defined data type.
Class Person
{ public: float age; string name;
bool operator < (const Person &rhs) const { return age < rhs.age; }
bool operator > (const Person& rhs) const { return age > rhs.age; }
}
int main()
{ std:: multiset <Person> multiset = { {25,"Rupesh"}, {20, "Hitesh"} };
  for (const auto & e:multiset)
      cout << e.age << "    " << e.name <<endl;
  return 0;
}
```

⑤ Container : Map

Syntax : map < T_1 , T_2 > ObjectName

where, T_1 is key type and T_2 is value Type.

1. std:: map is associative container that store elements in key value combination.
2. key always should be unique, otherwise it overrides the prev. value.
3. Values can be same for different key.
4. once key is created it will be always there we can not delete it. value of key can be changed.
5. It is implemented using self-Balanced Binary ~~tree~~ Search Tree. (AVL / Red Black Tree).
6. It stores key value pair in sorted order on the basis of key.
7. std::map is generally used in Dictionary type problems.

```
// Example :: Dictionary
#include <iostream>   #include <functional>
#include <map>        #include <vector>      using namespace std;

int main()
{
    std:: map < string, int > map1;
    map1 ["chotu"] = 90909090;                creation of pair in map.
    map1 ["Amit"] = 982345 3517;
or  map1.insert(std::make_pair ("Bot", 78237539));
    // Loop through map
    for (auto& e : Map1)
        cout << e.first << "  " << e.second << endl;
    // Access using[]operator
    cout << map1 ["chotu"]<<endl;          O/P : 90909090.
    map1 ["chotu"] = 37458923;
}
// Assigning multiple values to a key
std :: map < string , vector < int > , std::less <> > Map2;
Map2 ["chotu"].push_back [90909090];
Map2 ["Amit"]. push_back (73456923);   } Two values for
Map2 ["Amit"]. push_back ( 86453947);    single key

for (auto &e1 : Map)
{
    cout << e1.first <<endl;
    for (auto &e2 : e1.second)
        cout << e2 << "  ";
    cout <<endl;
}
```

O/P

```
Amit
73456923   86453947
Chotu
90909090
```

⑥ Container : Multimap →

Syntax : multimap < T_1, T_2 > Objectname

T_1 → key type
T_2 → value type.

→ 1. Multimap is an associtive container that contains a sorted list of key-value pairs. while permitting multiple entries with same key.

2. It stores key value pair in sorted order on the basis of key.
syntax : std::multiset < char, int, std::greater <> > mulmap ;

3. Data structure used in multimap is not defined by standard, but red-black tree is assumed by most of the people.

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

5. We don't have at() and [] functions to get element like we have in std::map.

```
# include <iostream>
# include <vector>
# include <map>
using namespace std;

int main()
{   std::multimap < char, int > Multimap1;
    multimap1.insert (make_pair ("a", 1));
    Multimap1.insert (make_pair ('a', 2));
    multimap1.insert (make_pair('a', 3));
    multimap1.insert (make_pair ('b', 4));
or  multimap1.insert (pair <char, int> ('b', 5));    ('b', 5)

    // Iterate over multimap
    for (autof e : multimap1)
    { cout << e.first << " " << e.second <<endl;
    }
    cout <<endl;

    // Get all the pairs of given key
    auto range = multimap1.equal_range ('a');
    for(auto it = range.first ; it != range.second ; it++)
    {     cout << it->first << " " << it->second <<endl;
    }

    return 0;
}
```

* Emplace in stl

⇒ 1. All the container support insert and emplace operation to store the data.

2. Emplace is used to construct object in-place and avoids Unnecessary copy of objects.

3. Insert and Emplace is equal for premetitive data type but when we deal with heavy objects. we should use emplace for the efficiency.

Q. Difference and need / Efficiency ~~of~~ between insert & Emplace

```
# include <iostream>
# include <set>
using namespace std;

Class A
{ public:
    int x;
    A(int x=0) : x{x} { cout << "construct" << endl; };
    A (const A& rhs) { x=rhs.x ; cout << " Copy " << endl; }
};
bool operator < (const A& lhs, const A& rhs) { return lhs.x < rhs.x;}

int main ()
{   set <A> set ;
    // A a(10);
    // set . insert (~~A(10)~~ a);
Or  set . insert (A(10));
    set . emplace (2);
    return 0;
}
```

Creating object & passing object

O/P: construct copy

O/P: construct.

⇒ In insert object is constructed and then value is copied. But in emplace ~~a~~ there is no copy of value. So emplace is efficient in heavy objects. such as image processing, etc.

// forward list in STL (Single Linked List)

1. This is single linked list what we studied in DSA.
2. We should use this class instead of traditional single list (Manually defined) because.
   a) This is well tested.
   b) Bunch of available function to perform different operation.
3. Few Available Operations
   Operator =, assign, front, empty, max_size, clear, insert_after, emplace_after, reverse, sort, merge, splice after, unique, remove, remove_if, resize.

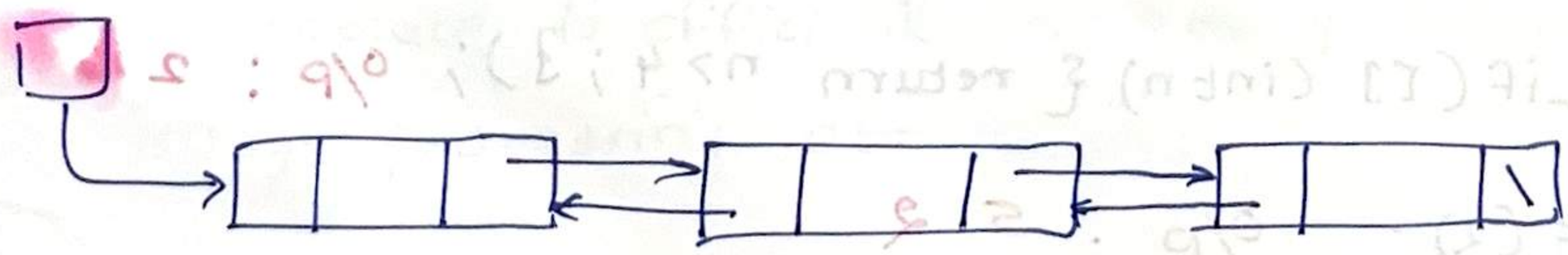
```
#include <iostream>
# include <forward_list>
using namespace std;

int main()
{ forward_list <int> list1 = {5, 8, 4, 6, 2};
  forward_list <int> list2 = {7, 6, 1, 9};

  // list1.insert_after(list1.begin(), 8)   O/p: 5 8 2 4 6 2
  // list1.reverse()                        O/p: 2 6 4 2 5

  // list1.sort();
  // list2.sort();
  // list1.merge(list2);                    O/p: 2 2 4 5 6 6 7 9

  // list1.splice_after(list1.begin(), list2)   O/p 5 7 6 1 9 2 4 6 2

  if {5, 4, 6, 2, 2}
  // list1.unique();                        O/p: 5 4 6 2   unique() funct removes only adjacent duplicate entries.

  But if {5, 2, 4, 6, 2}
  // list.unique();                         O/p: 5 2 4 6 2

  // ~~remo~~ list1.remove(2)               O/p: 5 4 6

  // list1.remove_if([] (int n) { return n>4; });   O/p: 2

  // list1.resize(2);                       O/p: 5 2
}
```

* List in STL (Doubly Linked List)

1. This is double linked list what we studied in DSA.
2. List is a sequence container that allow non-contiguos memory allocation.
3. List is faster compared to other sequence container such as vector, forward_list, deque.
4. We should use this class instead of traditional double linked list (manually defined one) because.
   a. This is well tested.
   b. There are bunch of available ~~opo~~ functions.
5. Few Available functions.
   operator =, assign, front, back, empty, size, max_size, clear, insert, emplace, push_back, pop_back, push_front, pop_front, reverse, sort, merge, splice, unique, remove, remove_if, resize.

```
# include <iostream>
# include <list>
using namespace std;

int main()
{
    list <int> list1 = {5, 2, 4, 6, 2};
    list <int> list2 = {7, 6, 1, 9};

    for (auto& elm: list1)
        cout << elm << endl;
    return 0;
}
```

⇒ All functions we can practise or use as required.



* std :: pair in C++ — Imp

Syntax : std :: pair < T_1, T_2 > obj ;

Notes :

1. std :: pair is a struct template that provides a way to store two heterogeneous objects as a single unit.
2. map, multimap, unorder_map, unorder_multimap can use pair to insert data into their structure.
3. Suppose, if we want to return two data from any function then this is very usefull/helpfull.

```
#include <iostream>
#include <vector>

int main()
{
    std::pair < int , int > obj (10, 20);
    print (obj);

    std::pair< int, int > obj = std::make_pair (10,20);
    print (obj)

Imp  std:: vector <std :: pair < std:: string , uint >> list ;

    list.push_back (make_pair ("Rupesh", 90));
    list.push_back (make_pair ("Hitesh", 28));
or  list.push_back (std::pair<std::string ,int> ("Bhupendra", 22));
or  list.push_back (std::pair ("Sagar", 18));

    for (auto& elm: list)
        cout << elm.first << " " << elm.second <<endl;

    return 0;
}

void print(std :: pair <int, int > &obj)
{
    cout << obj.first << " " << obj.second <<endl;
}
```

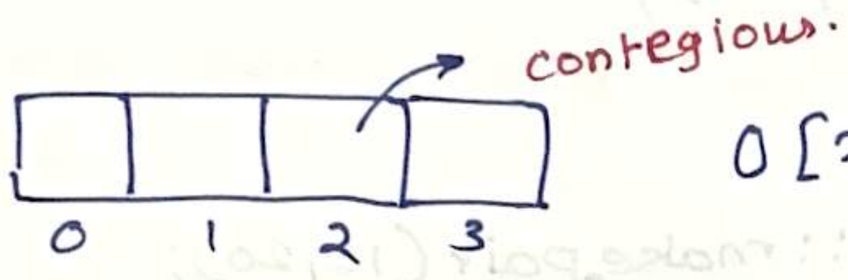
* deque in c++ (container)

→ deque is totally different than queue or stack.
→ If want to compare deque then compare with vector.

1. std::deque is an indexed sequence container A[2].
2. It allows fast insertion or removal from beginning and end.
   → vector allows ~~only~~ insertion or removal from only end.

3. It uses individual allocated fixed size array, with additional bookkeeping meaning index based access to deque must perform two pointer dereference, but in vector we get in one dereference.

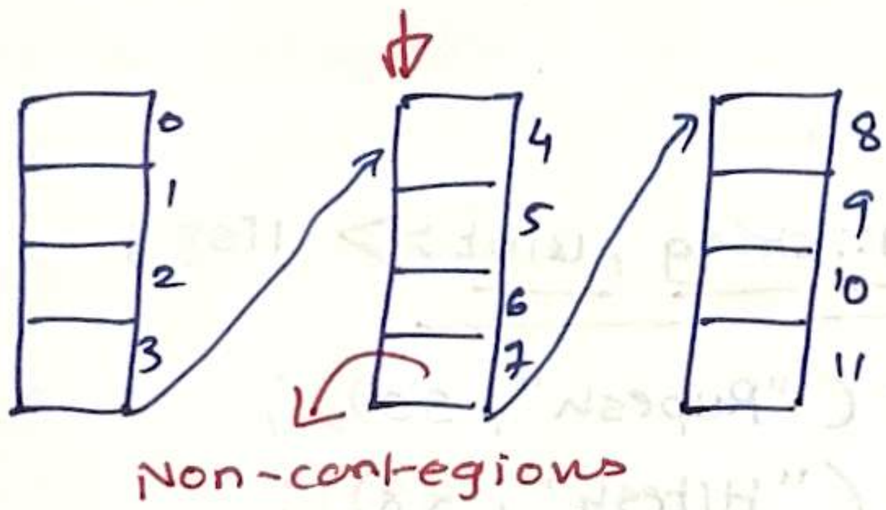
vector



0[2]

we can directly reach to index position

Deque



0[7]

7 ⇒ 4 − 3

first dereference 4 and reach at proper array and dereference 3 to get element.

4. Unlike vector elements of deque are not stored contigeous.
5. The storage of deque is automatically expanded and contracted as needed.
6. Expansion of deque is cheaper than expansion of vector (As vector double the array inside when capacity is exhausted. but in deque we are just adding fixed size small array)
7. A deque holding just one element has to allocate its full internal array.

→ Time Complexity

→ Random access - constant O(1)

→ Insertion or removal of elements at the end or beginning → constant O(1)

→ Insertion or removal of elements in between → linear O(n)

(As we have to shift the element)

```
# include <iostream>
# include <deque>
using namespace std;
void print (const std::deque <int> & dqu)
 {  for (int num:dqu)
      {  cout << num << "  ";
         cout << endl;
      }
 }

int main()
{
    std::deque <int> dqu = { 2, 3, 4 };
    dqu.push_front (1);
    dqu.push_back (5);          O/p: 1, 2  3  4  5
    print (dqu);

    std::deque <int> dqu = { 2, 3, 4 };
    dqu.pop_front ();
    dqu.pop_back ();            O/p: 3
    print (dqu);

    return 0;
}
```

* Container : Queue

Syntax : std::queue < T > ObjectName ;

1. Std:: Queue (queue) class is a container adapter that gives the programmer the functionality of a queue.
2. queue is FIFO (first in first out) data structure.
3. std::queue provide only specific set of functions.
4. std::queue allows to push (insert) at back and pop (remove) from front.
5. std::queue gives front, back, push, pop, empty, size.

```
#include <iostream>
#include < queue>
using namespace std;

Void Print (std::queue <int> que)
{   while (!que.empty)
    {   cout << que.front();
        que.pop();
    }
}

int main()
{
    std::queue <int > que;
    que.push(2);
    que.push (3);
    que.push (4);
    print(que);
    return 0;
}
```

can not iterate using for as it follows FIFO.

* Container : Stack

Syntax : std:: stack <T> ObjectName;

1. Std:: stack class is a container adapter that gives the programmer functionality of a stack.
2. Internally it uses std:: deque STL container.
3. It is LIFO (last in first out) data structure.
4. std:: stack allows to push (insert) and pop (remove) only from back.

* Functions provided — Time Complexity

→ empty () - Returns wheather the stack is empty = O(1)

→ size () - Returns the size of the stack = O(1)

→ top () - Returns a reference to the top most element of the stack. = O(1)

→ push (g) - Adds the element 'g' at the top of stack = O(1)

→ pop () - Deletes the top most element of stack = O(1)

Code

```
# include <iostream>
# include <stack>
using namespace std;
Void Print (std:: stack <int> stk)
{   while (! stk.empty())
    {   cout << stk.top() <<endl;
        stk.pop();
    }
}

int main()
{   std:: stack <int> ~~set~~ stk;
    stk.push(2);
    stk.push(3);
    stk.push(4);
    print (stk);
    return 0;
}
```

We can also overload it

```
std:: stack <int, std:: vector <int >> stk;
```

Now it will use vector internally as a data type.

Container : Priority Queue (Imp) used for leetcode

Syntax : std::priority_queue <int> q;

1. std::priority_queue is a container adaptor that provides constant time lookup of the largest or smallest element.
2. By default std::vector is the container used inside.
3. Cost of insertion and extraction is logarithmic.
4. std::priority_queue is implemented using std::make_heap, std::push_heap, std::pop_heap functions.

```
# include <iostream>        # include <vector>
# include <queue>           # include <functional>

template <typename T> void print_queue (T & q)
{   while (!q.empty())
    {  std::cout << q.top() << " ";
       q.pop();
    }
    std::cout << '\n';
}

int main()
{
    std::priority_queue <int> q;
    for ( int elm : {1,8,5,6,3,4,0,9,7,2}) {q.push (elm);}
    print_queue (q);

    std::priority_queue <int, std::vector<int>, std::greater <int>> q2;
    for (int elm: {1,8,5,6,3,4,0,9,7,2} {q2.push (elm);}
    print_queue (q2);

// using lambda to compare elements.
    auto cmp= [] (int left, int right) { return (left) < (right); }
    std::priority_queue < int, std::vector<int>, decltype (cmp) > q3 (cmp);

    for (int elm: {1,8,5,6,3,4,0,9,7,2} {q3.push (elm);}
    print_queue (q3);

}
```

container

comparator

O/p: 9 8 7 6 5 4 3 2 1 0

O/p: 0 1 2 3 4 5 6 7 8 9

⇒ ~~Pri~~ Priority Queue are used for Min Heap or Max heap.

⇒ we can also overload, std::stack < int, std::vector <int>> stk;

→ Now it will use vector internally

Container : Unordered Set

Syntax: std:: unordered_set <int> object name = { - - - - - }

⇒ In set, we get unique elements (duplicates removed) and sorted order (Ascending or Descending). This two are property of se

⇒ Now, if we just want unique elements and sorting is not require then we use Unordered_set.

Imp

1. Unorder set is a associative container that contains set of unique objects.
2. Search, insertion and removal have average constant time complexity.
3. Internally, the elements are organized into buckets.
4. This allows fast access to individual elements, since once a hash is computed it refers to the exact bucket the element is placed into.

Why Unordered set?

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

```
#include <iostream>
# include < unordered_set >
Using namespace std;

int main()
{
    std:: Unordered_set <int> Uset = { 4, 1, 2, 3, 4, 2, 3 };
    auto search = Uset.find (2);
    if( search != Uset.end ())
    {
        std:: cout << " found " << (*search) << '\n';
    }
    else
        std:: cout << "Not found" \n";

    For (auto & elm : uset)
        std:: cout << elem << " ";

    cout << endl;

    return 0;
}
```

O/p : 4 1 2 3

(unique but not sorted)

* Container : Unordered Multiset

Syntax : std::unordered_multiset <int> umset = { ..... };

Difference betn set, unordered_set, unordered_multiset.

① Unique
② Sorted

e.g: 3, 4, 2, 5, 2, 4

Set	Unordered set	Unordered multiset
① & ②	①	×
2, 3, 4, 5	3, 4, 2, 5	4, 4, 3, 2, 2, 5

* (It may come in any order)

1. std::unordered multiset is an associative container that contains set of possibly non-unique objects.
2. Search, insertion, and removal have average constant time-complexity.
3. Internally, the elements are organized into buckets.
4. It uses hashing to insert elements into buckets.
5. This allows fast access to individual elements, because after computing the hash of value it refers to the exact bucket the element is placed into.

→ This is preffered over map as it uses Red-Black tree inside which has time constant of O(n) and here it is O(1) so this faster.

Imp

```
#include <iostream>
#include <unordered-set>
Using namespace std;
int main()
{
std::unordered_multiset <int> umset= {4,1,2,3,9,4,3,2,9,8,10};
 auto search= umset.find(2);
 if ( search != umset.end()) {
      std::cout << "Found" <<endl;
 else
      std::cout << "Not found" <<endl;

 for (auto& elm : umset)
      cout << elm << " ";

 cout <<endl;
 cout << umset.count(4) <<endl;
 return 0;
}
```

* Container : Unordered Map →

Syntax : std::unordered_map < int, char > umap = {{1,'a'}, {2,'b'}};

⇒ Difference between Maps and Unordered Maps

→ In the Map, internally Red Black Trees are used so time complexity is logarithmic.
While In Unordered map Hashing buckets are used so time complexity for addition and removal is O(const).

→ map gives sorted order but Unoreded doges not give sorted order.

→ Unorderd map is an associative Container that contains key-value pairs with unique keys.

→ Same points as 2, 3, 4 & 5 written in unorder multiset.

Q. why unordered_map ?

⇒ Maintain a collection of unique {key : value} pair with fast insertion and removal.

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main()
{
    std::unordered_map <int, char> umap = {{1,'a'}, {2,'b'}};
    // Access
    cout << umap[1] << endl;
    cout << umap[2] << endl;
    // Update
    umap[1]='c';
    // Iterate
    for (auto& elm:umap)
        std::cout<< elm.first << " " <<elm.second << " ";
    cout<<endl;
    // find
    auto result = umap.find(2);
    if (result != umap.end())
        cout << "found" <<result->first << " " <<result->second <<endl
    else
        cout << "Not found";
    return 0;
}
```

Container: Unordered Multimap

syntax: std::unordered_multimap <int, char> umm = {{5, 'd'}};

① Unique
② sorted

Map	Multimap	Unordered map	Unordered multimap
① & ②	②	①	×

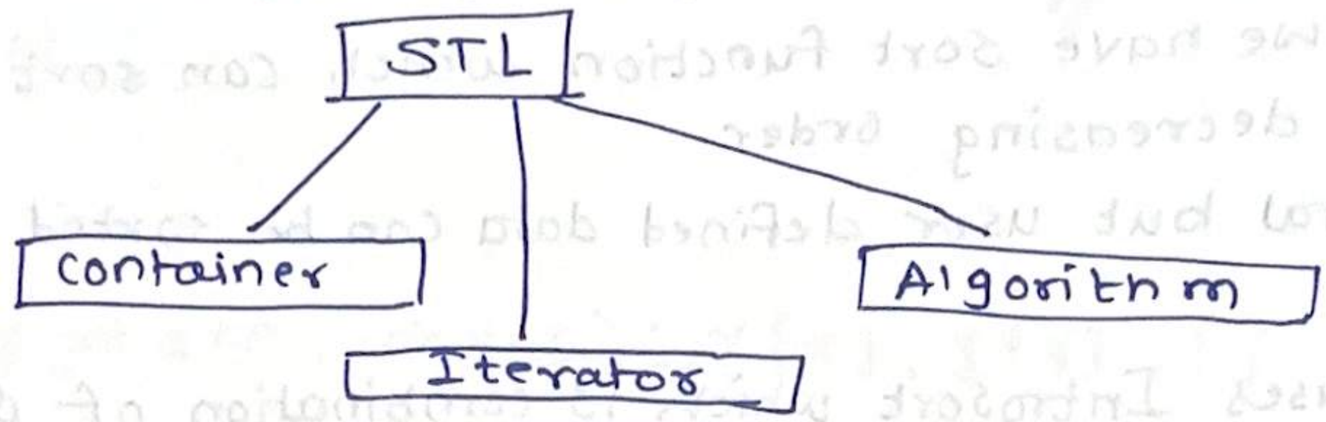
⇒ Uses Hashing, so constant time complexity.

1. Unordered multimap is an unordered associative container that supports equivalent keys (an unordered multimap may contain multiple copies of each key value) and that associates value of another type with the keys.
2. Point 2, 3, 4 & 5 are same as written in Unorder multiset.

Q. Why Unordered Multimap?
→ Maintain a collection of duplicate {key : value} pairs with fast insertion and removal.

```
#include <iostream>
# include <unordered_map>
using namespace std;
int main()
{
  std::unordered_multimap <int, char> umm = {{5, 'd'}};
  umm.insert ({1, 'a'});
  umm.insert (std::pair <int, char> (2, 'b'));
  umm.insert (std::makepair (3, 'c'));
  umm.insert (std::makepair (3, 'c'));

  for (auto& elm : umm)
      cout << elm.first << " " elm.second << endl; }

  return 0;
}
```

STL Algorithm



→ 1. STL Algorithms library defines functions for a variety of purposes. (eg: Searching, sorting, counting, manipulating, etc.)

2. Algorithms are applied to range of elements.

3. <algorithm> header is used to get all algorithm in STL.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector <int> vec {4, 3, 5, 2, 6, 1};
*   vector <int> :: iterator it = std :: find (vec.begin(), vec.end(), 4)
    if (it == vec.end())
        cout << "Not found" << endl;
    else
        cout << "found" << endl;
// Iterator can be defined as.
    auto it = find (vec.begin(), vec.end(), 5)

}
```

// find algorithm

* Algorithm : Sort (Important)

1. In C++, STL, we have sort function which can sort in increasing or decreasing order.
2. Not only integral but user defined data can be sorted using this function.
3. Internally it uses Introsort which is combination of QuickSort, Heapsort and Insertion Sort.
4. By default, it uses Quicksort but if Quicksort is doing unfair partioning and taking more than N*log N time. ~~it~~ then it switches to Heapsort and when the array size becomes really small, then ~~it~~ it switches to Insertion Sort.
5. We can use parallel execution policy for better performance.

Types:

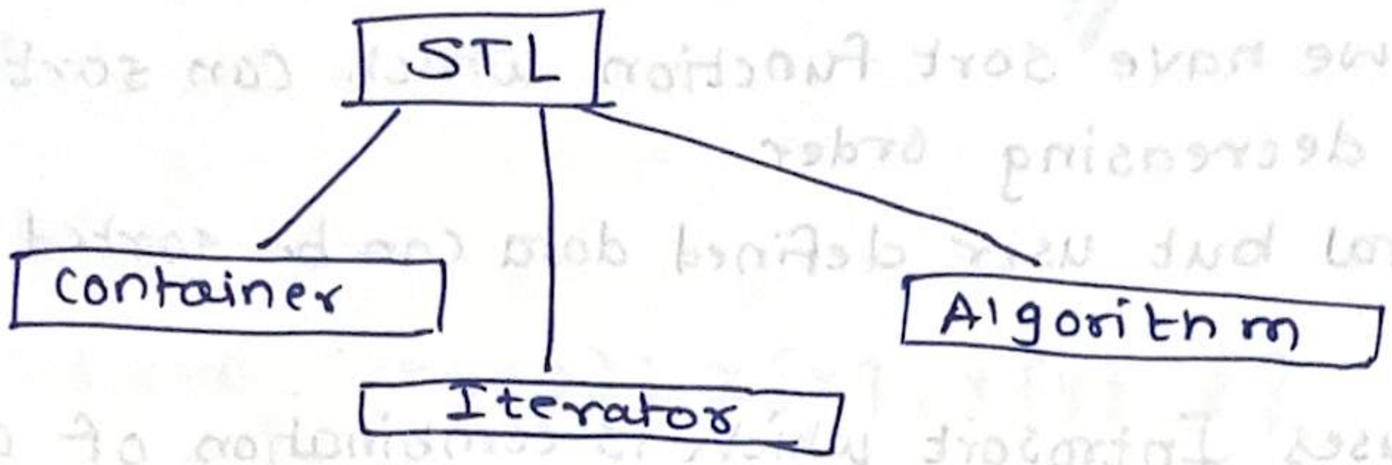
1. Sorting integral data types.
2. Sorting user defined data types.
3. Sort using function object.
4. Sort using lambda expression.

Code:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <execution>
using namespace std;

int main()
{
    // 1. Sort with integral data types.

    std::vector<int> Vec {5,4,6,7,3,2,8,9,1};
    ~~std::sort~~
                                  for multi threading. (optional)
    std::sort(std::execution::par, Vec.begin(), Vec.end());

    for (auto elm:vec)
        cout << elm << " ";

}
```

⇒ Here as integral data type is used hence no need to define the iterator

STL Algorithm



→ 1. STL Algorithms library defines functions for a variety of purposes (eg: Searching, sorting, counting, manipulating, etc.)

2. Algorithms are applied to range of elements.

3. <algorithm> header is used to get all algorithm in STL.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector <int> vec {4, 3, 5, 2, 6, 1};
*   vector <int> :: iterator it = std :: find (vec.begin(), vec.end(), 4)
    if (it == vec.end())
        cout << "Not found" << endl;
    else
        cout << "found" << endl;
// Iterator can be defined as.
    auto it = find (vec.begin(), vec.end(), 5)

}
```

// find algorithm

* Algorithm : Sort (Important)

1. In C++, STL, we have sort function which can sort in increasing or decreasing order.
2. Not only integral but user defined data can be sorted using this function.
3. Internally it uses Introsort which is combination of QuickSort, Heapsort and Insertion Sort.
4. By default, it uses Quicksort but if Quicksort is doing unfair partioning and taking more than N*log N time. ~~it~~ then it switches to Heapsort and when the array size becomes really small, then ~~it~~ it switches to Insertion Sort.
5. We can use parallel execution policy for better performance.

Types:

1. sorting integral data types.
2. Sorting user defined data types.
3. Sort using function object.
4. Sort using lambda expression.

Code:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <execution>
using namespace std;

int main()
{
    // 1. Sort with integral data types.

    std::vector<int> Vec {5,4,6,7,3,2,8,9,1};
    ~~std::sort~~
    std::sort(std::execution::par, Vec.begin(), Vec.end());   for multi threading. (optional)

    for (auto elm:vec)
        cout << elm << " ";

}
```

⇒ Here as integral data type is used hence no need to define the iterator

2. Sort user defined data types. Imp See the code

```
class Point
{
public:
    int x; int y;
    Point (int x=0, int y=0) : x{x}, y{y} {}
    bool Operator < (const Point& P1)
    {
        return (x+y) < (P1.x + P1.y);
    }
}

int main()
{
    std::vector < Point > vec {{1,2}, {3,1}, {0,1}};
    std::sort (vec.begin(), vec.end());
    for(auto e:vec)
        cout << e.x << " " << e.y << endl;
    return 0;
}
```

operator is defined for comparison.

3. Sort using a function object.

```
struct
{
    bool Operator () (int a, int b) const
    { return a<b;
    }
} CustomLess;

int main()
{
    vector <int> vec {5,4,6,7,3,2,8,9,1};
    sort (vec.begin(), vec.end, CustomLess());
    for(auto elm:vec)
        cout <<elm <<" ";
}
```

object of operator is passed.

4. Sort Using Lambda function.

```
int main()
{
    vector <int> vec {5,4,6,7,3,2,8,9,1};
    sort (vec.begin(), vec.end(), [](int a, int b) {return a<b;});
    for (auto elm:vec)
        cout << elm << " ";
}
```

operator is passed

Algorithm: is_Sorted

1. Check if the elements in range [first, last] are sorted.
   - By default it will check ascending order.

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

   - If we want to check wheather they are sorted in descending order

     is_sorted (vec.begin(), vec.end(), greater<int>());

Types:

1. Can check integral data types.
2. Can check user defined data types.
3. Can check using a function object.
4. Can check using lambda expression.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector <int> vec {5,4,6,7,3,2,8,9,1};
    sort (vec.begin(), vec.end())   O/p: 1,2 3 4 5 6 7 8 9
    for (auto& elm: vec)
        cout << elm << " ";
    cout << endl;

    is_sorted (vec.begin(), vec.end());   O/p: 1
    is_sorted (vec.begin(), vec.end(), greater <int> ());
                                          O/p: 0
}
```

Algorithm: Partial_sort

1. Partial sort rearranges elements such that the range [first, middle) contains the sorted elements.
2. The order of equal elements is not guarranted to be preserved.
3. The order of remaining elements is unspecified.

Code:

[ )

```
#include <algorithm>
#include <functional>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector <int> vec {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
    partial_sort (vec.begin(), vec.begin()+4, vec.end());
    for (int val: vec)
        cout << val << " ";
    return 0;
}
```

O/p : 0 1 2 3 8 7 6 9 5 4

Sorted

This order is random.

Algorithm : nth element

1. nth element is partial sorting algorithm that rearranges elements in [first , last) such that. first is included last is not included.

   a. The element at the n'th position is the one which should be at that position if we sort this list.

   b. It does not sort the list, just that all the elements, which precede the nth element are not greater than it.

2. nth element algorithm is implemented using introselect.

   a. introselect is a hybrid of quicksort and medians of medians algorithm.

   b. median of medians is a median selection algorithm for better pivot selection mainly used in quick select.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main()
{ std::vector <int> vec. {5, 6, 4, 3, 2, 6, 7, 9, 3};
  nth_element (vec.begin(), vec.begin()+ v.size()/2 + v.end());
   cout << "The median is:" << v [v.size()/2] << endl;

  nth_element (v.begin, vec.begin()+1, vec.end, greater <int>());
  cout << " The second largest element is " << v[1] << endl;
}
```

Algorithm: is_sorted_Until

→ find how many elements are sorted in given range. [first, last].

Support types:

1. Integral data types.
2. User defined data types
3. a function object.
4. Using lambda function.

```
#include <iostream>
# include < vector >
# include <algorithm>
 using namespace std;

int main()
{
   vector <int> vec {4, 3, 2, 6, 7, 8, 1};
   auto it = is_sorted_until(vec.begin (), vec. end ());
   auto diff = distance (vec.begin (), it);
   cout << diff << endl;          O/P : 1   (As only first 4 is sorted)
                                            (Ascending order)
   auto it = is_sorted_until (vec.begin(), vec.end (), greater<int>());
   auto diff = distanc (vec.begin (), it);
   cout << diff << endl;          O/P: 3   ( As 4,3,2  (3) elements are
                                            sorted in descending order)
}  return 0;
```

Algorithm : iota

⇒ Fills the range [first, last] with sequentially increasing values, starting with value and repetitively evaluating ++value.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <numeric>
#include <vector>
using namespace std;

int main()
{
    std::list <int> lst (10);
    std::iota ( lst.begin(), lst.end(), -4);
    for (auto elm : lst)
        cout << elm << endl;

    std::vector <std::list<int>::iterator > vec (lst.size());
    std::iota (vec.begin(), vec.end(), lst.begin());

    for (auto elm:vec)
        cout << *elm << endl;

    return 0;
}
```

lst.begin() → from; lst.end() → To; -4 → starting from

O/p: -4, -3, -2, -1, 0, 1, 2, 3, 4

Algorithm: ~~Cyp~~ copy and Copy_if

→ Copy copies the elements in the range, defined by [first, last], to another range beginning at d_first.

→ Copy_if only copies if predicate returns true.

→ if source and destination range are overlapping then the behaviour is undefined.

```
#include <algorithm>
#include <iostream>          #include <iterator>
#include <vector>            #include <numeric>
using namespace std;
void print (vector <int> & vec)
{
    copy (vec.begin(), vec.end(), Ostream_iterator <int> (cout, " ");
    cout <<endl;
}
int main()
{
    vector <int> from_vector (10);
    iota (from_vector.begin(), from_vector.end(), 0);
    vector <int> to_vector 1 (from_vector.size());
    copy (from_vector.begin(), from_vector.end(), to_vector1.begin()).
    print (to-vector 1);
// Or ~~alta~~ alternatively,
    vector <int> to_vector 2;
    copy (from_vector.begin(), from_vector.end(), back_inserter (to-vector 2);
    print (to_vector 2);
// either way is equivalent to
    vector <int> to_vector = from_vector
// what is the benefit of copy function if we can use above method?
```

for copying.

→ Using copy we can partially copy elements using copy() function.

```
    copy (from_vector.begin()+1, from-vector.end()-2, to_vector1.begin())
    cout << " Odd number in to_vector are: ";
    vector <int> to_vector 3 (5);
    copy_if (from_vector.begin(), from_vector.end(), to_vector3.begin(),
                                    [] (int x) {return (x %2)==1?;} ;
    print (to_vector 3);
    return 0;
}
```

Algorithm : copy_n & copy_backward

→ copy_n() function is used to copy given number of elements from source to destination.

→ copy_backward() function copies the elements from range, defined by [first, last], to another range ending at d_last. The elements are copied in reverse order (The last element is copied first). but their relative order is preserved.

code

```
#include <iostream>
# include <string>
# include <algorithm>
# include <iterator>
# include <vector>
# include <numeric>
using namespace std;
void Print (vector <int> & vec)
{  copy (vec.begin (), vec.end (), ostream_iterator <int> (cout, " ");
   cout <<endl;
}

int main()
{
   vector <int> from_vector (10);
   iota (from_vector.begin() , from_vector.end , 0) ;   O/P: 0 1 2 3 4 5 6 7 8 9

   // Copy
   vector <int> to_vector1;
   copy_n (from_vector.begin(), 4, back_inserter (to_vector1));
   print (to_vector1);   O/P: 0 1 2 3 4

   // Copy Backward
   vector <int > to_vector2 (15);
   copy_backward (from_vector.begin() , from_vector.end(), to_vector2.end())
   print (to_vector2);

   return 0;
}
```

* std::span <T>

→ It support contigeous memory like C-Type → Array
or STL → Array
→ Vector

→ span contains the references of the memory location in container
→ It can access the data or modify the data but it can not increase or decrease the size of container.

```
#include <iostream>
#include <vector>
#include <array>
#include <span>
using namespace std;

void Print_Content(span<int> container)
{
    for(const auto &e : container)
        cout << e << ' ';
    cout << endl;
}

void Modify_content(span<int> container)
{
    for(auto &e : container)
        e*=2;
}
int main()
{
    int a[] {23,45,67,89};
    print_content(a);

    vector vec {1,2,3,4,5};
    print_content(vec);

    array a2 {-14,55,24,67};
    print_content(a2);

    modify_content(a2);
    print_content(a2);

    return 0;
}
```

we are able to pass array, vector, int the same print function without modifying its parameter.

span is available in c++20 and above

* std :: mdspan

→ mdspan create multi dimensional array (vector from given container.

```
#include <vector>
# include <mdspan>
# include <print>

int main()
{
    std:: vector v {1,2,3,4,5,6,7,8,9,10,11,12};

// view data as contigeous memory representing 2 rows of 6 ints each
  auto ms2 = std:: mdspan (v.data(), 2, 6);
//view the same data as a 3D array 2x3x2
  auto ms 3 = std:: mdspan (v.data (), 2, 3, 2);

//write data using 2D view
For (size_t i=0; i != ms2.extent(0); i++)
    for (size_t j=0; j != ms2.extent (1); j++)
        ms2 [i,j] = i* 1000 + j;

// read back using 3D view
for (size_t i=0 ; i != ms3. extent (0); i++)
{   std:: println (" slice @ i = {}", i);
    for (size_t j=0 ; j != ms3.extent(1) ; j++)
    {   for (size_t k=0 ; k != ms 3.extent (2) ; k++)
            std::print {"{} ", ms3 [i, j, k]);
        std::println (" ");

    }
}
```

Array or Vector access out of the boundary element and not crash in c++.

```
#include <iostream>
# include <vector>
Using namespace std;

int main ()
{
  std::vector <int> vec;
  // cout << vec [100000000] <<endl;
```

too large

This will throw segmentation fault error. and program will crash. instead solution is

```
try {
    cout << vec.at (100000000) <<endl;
  } catch (...) {
    cout << " Out of range handled "<<endl;
  }

  cout << " Hello there I am still alive" <<endl;

  return 0;

}
```