Contents

[Contents 1](#_Toc515274050)

[1 STL(Standard Template Library) 2](#_Toc515274051)

[1.1 Array 3](#_Toc515274052)

[1.1.1 at ( ) 3](#_Toc515274053)

[1.1.2 [ ] Operator 3](#_Toc515274054)

[1.1.3 front() 3](#_Toc515274055)

[1.1.4 back() 4](#_Toc515274056)

[1.1.5 fill() 4](#_Toc515274057)

[1.1.6 swap() 4](#_Toc515274058)

[1.1.7 operators ( == , != , > , < , >= , <= ) 5](#_Toc515274059)

[1.1.8 Empty() 5](#_Toc515274060)

[1.1.9 Size() 5](#_Toc515274061)

[1.1.10 max\_size() 5](#_Toc515274062)

[1.1.11 begin() 5](#_Toc515274063)

[1.1.12 End 5](#_Toc515274064)

[1.2 Vector 5](#_Toc515274065)

[1.2.1 push\_back() 6](#_Toc515274066)

[1.2.2 insert 7](#_Toc515274067)

[1.2.3 std::vector::pop\_back() 9](#_Toc515274068)

[1.2.4 std::vector::erase() 9](#_Toc515274069)

[1.2.5 std::vector:: resize() 11](#_Toc515274070)

[1.2.6 std::vector:: swap() 12](#_Toc515274071)

[1.3 Deque 13](#_Toc515274072)

[1.3.1 std::deque::push\_back / :: push\_front / :: insert 13](#_Toc515274073)

[1.3.2 std::deque::pop\_back / :: pop\_front 13](#_Toc515274074)

[1.3.3 std::deque:: empty / :: size / :: max\_size / :: swap 13](#_Toc515274075)

[1.4 Map 14](#_Toc515274076)

[1.4.1 std::map::at / :: [ ] 15](#_Toc515274077)

[1.4.2 std::map::empty /::size/::max\_size 15](#_Toc515274078)

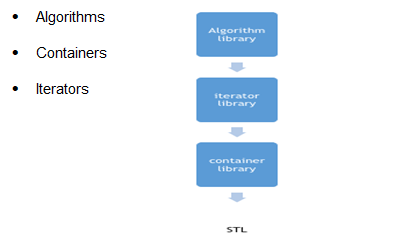
[1.4.3 std::map::insert/::insert\_or\_assign 15](#_Toc515274079)

[1.4.4 std::map::erase/::clear 16](#_Toc515274080)

[1.4.5 begin, end and find 16](#_Toc515274081)

# STL(Standard Template Library)

STL is an acronym for standard template library. It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms .STL is mainly composed of:



STL is a generic library , i.e. a same container or algorithm can be operated on any data types , you don’t have to define the same algorithm for different type of elements.

For example, sort algorithm will sort the elements in the given range irrespective of their data type , we don’t have to implement different sort algorithm for different data types.

**Containers in STL**

Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc.

These container are generic, they can hold elements of any data types, for example: vector can be used for creating dynamic arrays of char, integer, float and other types.

**Algorithms in STL**

STL provide number of algorithms that can be used of any container, irrespective of their type. Algorithms library contains built in functions that performs complex algorithms on the data structures.

For example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary\_search() and so on.

Algorithm library provides abstraction, i.e. you don't necessarily need to know how the algorithm works.

**Iterators in STL**

Iterators in STL are used to point to the containers. Iterators actually acts as a bridge between containers and algorithms.

For example: sort() algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

**STL Container**

## Array

Arrays, as we all know, are collection of homogenous objects. array container in STL provides us the implementation of static array, though it is rarely used in competitive programming as its static in nature but we'll still discuss array container cause it provides some member functions and non-member functions which gives it an edge over the array defined classically like, int array\_name[array\_size].

SYNTAX of array container:

array <object\_type, array\_size> array\_name;

The above code creates an empty array of object\_type with maximum size of array\_size. However, if you want to create an array with elements in it, you can do so by simply using the = operator.

Example of array container:

#include <vector>

int main()

{

array<int, 4> odd\_numbers = { 2, 4, 6, 8 };

}

**Member Functions of array template**

### at ( )

This method returns value in the array at the given range. If the given range is greater than the array size, out\_of\_range exception is thrown. Here is a code snippet explaining the use of this operator :

Example of at ( )

#include <iostream>

#include <array>

using namespace std;

int main()

{

array<int, 10> array1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

cout << array1.at(2); // prints 3

cout << array1.at(4); // prints 5

return 0;

}

### [ ] Operator

The use of operator [ ] is same as it was for normal arrays. It returns the value at the given position in the array. Example : In the above code, statement cout << array1[5]; would print 6 on console as 6 has index 5 in array1.

### front()

This method returns the first element in the array.

### back()

This method returns the last element in the array. The point to note here is that if the array is not completely filled, back() will return the rightmost element in the array.

#include <iostream>

#include <array>

using namespace std;

int main()

{

array<int, 10> array1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

cout << array1.front(); // prints 1

cout << array1.back(); // prints 9

return 0;

}

### fill()

This method assigns the given value to every element of the array, example :

#include <array>

int main(){

array<int, 8> myarray;

myarray.fill(1);

cout << "myarray is : ";

for (int i = 0; i < 8; i++) {

cout << myarray[i] << " ";

}

return 0;

}

/\* ouput will be

myarray is : 1 1 1 1 1 1 1 1\*/

### swap()

This method swaps the content of two arrays of same type and same size. It swaps index wise, thus element of index i of first array will be swapped with the element of index i of the second array, and if swapping any of the two elements throws an exception, swap() throws exception. Below is an example to demonstrate its usage :

#include<iostream>

#include<array>

using namespace std;

int main(){

array <int, 8> a = { 1, 2, 3, 4, 5, 6, 7, 8 };

array <int, 8> b = { 8, 7, 6, 5, 4, 3, 2, 1 };

a.swap(b); // swaps array a and b

cout << "a is : ";

for (int i = 0; i < 8; i++) {

cout << a[i] << " ";

}

cout << endl;

cout << "b is : ";

for (int i = 0; i < 8; i++) {

cout << a[i] << " ";

}

return 0;

/\* ouput will be

a is : 8 7 6 5 4 3 2 1

b is : 1 2 3 4 5 6 7 8 \*/

}

### operators ( == , != , > , < , >= , <= )

All these operators can be used to lexicographically compare values of two arrays.

### Empty()

This method can be used to check whether the array is empty or not.

Syntax : array\_name.empty(), returns true if array is empty else return false.

### Size()

This method returns the number of element present in the array.

### max\_size()

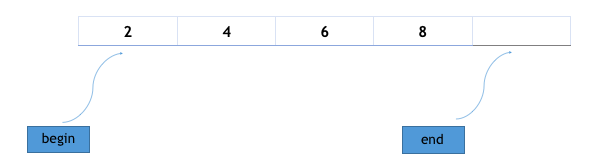
This method returns the maximum size of the array.

### begin()

This method returns the iterator pointing to the first element of the array. Iterators are just like pointers and we’ll discuss them later in the lessons, for now you can just think of an iterator like a pointer to the array.

### End

This method returns an iterator pointing to an element next to the last element in the array, for example the above array has 4 elements and the end() call will return the iterator pointing to the 4th index of the array.



## Vector

**Vectors or dynamic arrays** are sequence containers representing arrays that can change in size dynamically.

Size of the vector can be defined in run-time, i.e. Code below will create blank vector.

Example of array container:

vector <int>V1;

Initial size or capacity of the vector is 0 if user has not initialized it.

Vector can be initialized at the time of definition.

vector <string>V1{ "Jyoti", "ranjan", "Mohanty" };

The vector size is 3. If user wants to add next element in vector then it will be expanded by 6 (double of size of vector).

vector <char> cv(5);

Create a 5 elements char vector.

vector <char> cv1(4, 'a');

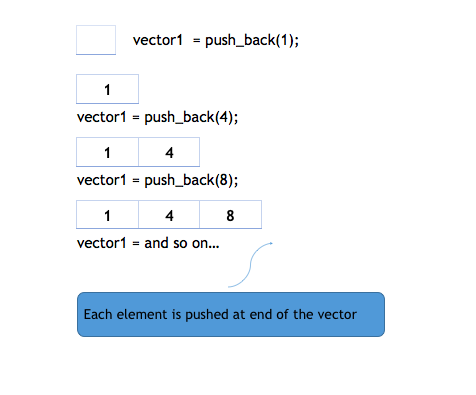
Initializes 4 elements char vector with ‘a’.

**Member Functions of Vector**

### push\_back()

push\_back() is used for inserting an element at the end of the vector. If the type of object passed as parameter in the push\_back() is not same as that of the vector or is not interconvertible an exception is thrown.

The following illustration will show how push\_back() works :



#include<iostream>

#include<vector>

using namespace std;

int main(){

vector<int> v;

v.push\_back(1); //insert 1 at the back of v

v.push\_back(2); //insert 2 at the back of v

v.push\_back(4); //insert 3 at the back of v

for (vector<int>::iterator i = v.begin(); i != v.end(); i++) {

cout << \*i << " "; // for printing the vector

}

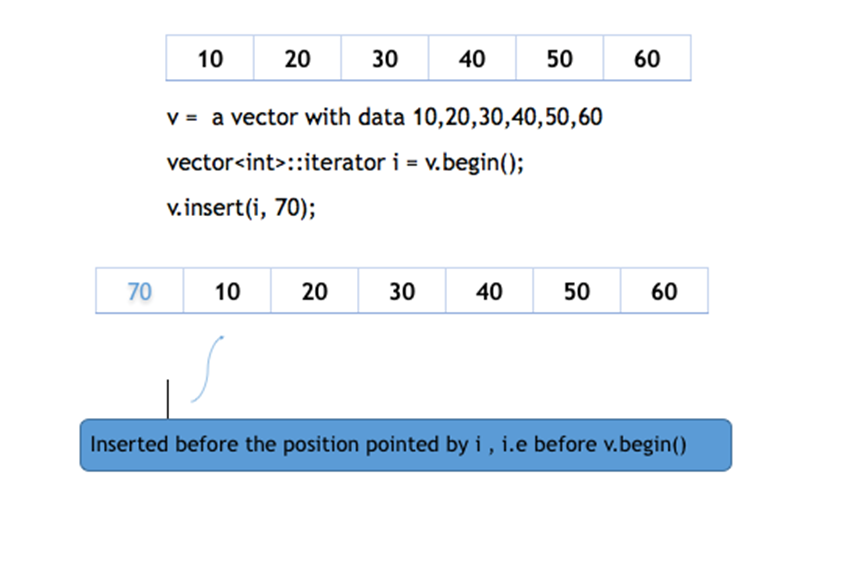
return 0;

}

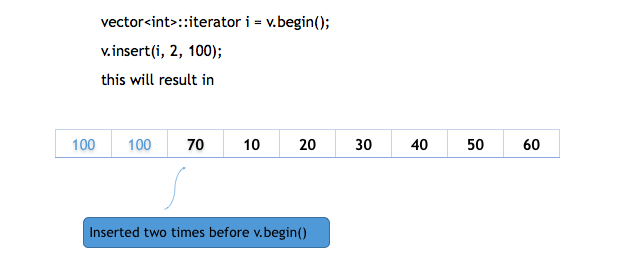
### insert

insert(itr, element) method inserts the element in vector before the position pointed by iterator itr.

The following illustration will show how insert works :



insert() can be overloaded by third argument, count as well. This count parameter defines how many times the element is to be inserted before the pointed position.



This method can also be used to insert elements from any other vector in given range, specified by two iterators, defining starting and ending point of the range.

#include <iostream>

#include <vector>

void print\_vec(const std::vector<int>& vec)

{

for (auto x : vec) {

std::cout << ' ' << x;

}

std::cout << '\n';

}

int main()

{

//iterator insert(iterator pos, const T& value);

//iterator insert(const\_iterator pos, const T& value);

std::vector<int> vec(3, 100);

print\_vec(vec);

auto it = vec.begin();

//iterator insert(const\_iterator pos, T&& value);

it = vec.insert(it, 200);

print\_vec(vec);

//void insert(iterator pos, size\_type count, const T& value);

//iterator insert(const\_iterator pos, size\_type count, const T& value);

vec.insert(it, 2, 300);

print\_vec(vec);

// "it" no longer valid, get a new one:

it = vec.begin();

//template< class InputIt >

//void insert(iterator pos, InputIt first, InputIt last);

//template< class InputIt >

//iterator insert(const\_iterator pos, InputIt first, InputIt last);

std::vector<int> vec2(2, 400);

vec.insert(it + 2, vec2.begin(), vec2.end());

print\_vec(vec);

int arr[] = { 501, 502, 503 };

//iterator insert(const\_iterator pos, std::initializer\_list<T> ilist);

vec.insert(vec.begin(), arr, arr + 3);

print\_vec(vec);

}

OUTPUT:

100 100 100

200 100 100 100

300 300 200 100 100 100

300 300 400 400 200 100 100 100

501 502 503 300 300 400 400 200 100 100 100

### std::vector::pop\_back()

**void pop\_back();**

Removes the last element of the container.

Calling pop\_back on an empty container is undefined.

No iterators or references except for [back()](http://en.cppreference.com/w/cpp/container/vector/back) and [end()](http://en.cppreference.com/w/cpp/container/vector/end) are invalidated.

**Parameters**

(none)

**Return value**

(none)

**Complexity**

Constant.

**Exceptions**

(none)

**Example**

#include <vector>

int main()

{

std::vector<int> numbers;

numbers.push\_back(5);

numbers.push\_back(3);

numbers.push\_back(4);

// Contents are { 5, 3, 4 }

numbers.pop\_back();

// Contents are { 5, 3 }

}

### std::vector::erase()

Removes specified elements from the container.

1) Removes the element at pos.

2) Removes the elements in the range [first; last).

Invalidates iterators and references at or after the point of the erase, including the [end()](http://en.cppreference.com/w/cpp/container/vector/end) iterator.

The iterator pos must be valid and dereference-able. Thus the [end()](http://en.cppreference.com/w/cpp/container/vector/end) iterator (which is valid, but is not dereference-able) cannot be used as a value for pos.

The iterator first does not need to be dereference-able if first==last: erasing an empty range is a no-op.

**Parameters**

pos - iterator to the element to remove

first, last - range of elements to remove

Type requirements

-T must meet the requirements of Move-Assignable.

**Return value**

Iterator following the last removed element. If the iterator pos refers to the last element, the end() iterator is returned.

**Exceptions**

Does not throw unless an exception is thrown by the assignment operator of T.

**Complexity**

Linear: the number of calls to the destructor of T is the same as the number of elements erased, the assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements

**Example**

#include<iostream>

#include<vector>

using namespace std;

void print\_c(const std::vector<int>& c){

for (auto x : c){

std::cout << ' ' << x;

}

std::cout << '\n';

}

int main(){

vector<int> c{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

print\_c(c);

c.erase(c.begin());

print\_c(c);

c.erase(c.begin() + 2, c.begin() + 5);

print\_c(c);

for (auto it = c.begin(); it != c.end();){

if (\*it % 2 == 0){

it = c.erase(it);

}

else

it++;

}

print\_c(c);

}

OUTPUT :

0 1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

### std::vector:: resize()

Resizes the container to contain count elements.

If the current size is greater than count, the container is reduced to its first count elements.

void resize( size\_type count );

void resize( size\_type count, const value\_type& value );

**Parameters**

count - new size of the container

value - the value to initialize the new elements with Type requirements

**Return value**

(none)

**Exceptions**

If an exception is thrown, this function has no effect (strong exception guarantee).

In overload (1), if T's move constructor is not noexcept and T is not CopyInsertable into \*this, vector will use the throwing move constructor. If it throws, the guarantee is waived and the effects are unspecified..

**Complexity**

Linear in the difference between the current size and count. Additional complexity possible due to reallocation if capacity is less than count.

**Example**

#include<iostream>

#include<vector>

using namespace std;

void print\_c(const std::vector<int>& c){

for (auto x : c){

std::cout << ' ' << x;

}

std::cout << '\n';

}

int main(){

vector<int> c{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

print\_c(c);

//iterator erase( iterator pos );

//iterator erase( const\_iterator pos );

c.resize(12);

print\_c(c);

c.resize(5);

print\_c(c);

c.resize(15, 15);

print\_c(c);

}

OUTPUT :

0 1 2 3 4 5 6 7 8 9

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

0 1 2 3 4

0 1 2 3 4 15 15 15 15 15 15 15 15 15 15

### std::vector:: swap()

Exchanges the contents of the container with those of other. Does not invoke any move, copy, or swap operations on individual elements.

All iterators and references remain valid. The past-the-end iterator is invalidated.

void swap( vector& other );

**Parameters**

other - container to exchange the contents with

**Return value**

(none)

**Complexity**

Constant.

**Example**

#include<iostream>

#include<vector>

using namespace std;

void print\_c(const std::vector<int>& c){

for (auto x : c){

std::cout << ' ' << x;

}

std::cout << '\n';

}

int main(){

vector<int> c1{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int> c2{ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };

print\_c(c1);

print\_c(c2);

c1.swap(c2);

std::cout << "\n"<<"After Swapping"<< "\n\n";

print\_c(c1);

print\_c(c2);

}

OUTPUT :

0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18 19

After Swapping

10 11 12 13 14 15 16 17 18 19

0 1 2 3 4 5 6 7 8 9

## Deque

Deque is shorthand for doubly ended queue. Deque allows fast insertion and deletion at both ends of the queue. Although we can also use vector container for the insertion and deletion at both of its ends, but insertion and deletion at the front of the array is costlier than at the back, in case of deque but deque are more complex internally. And it is a leaner data structure.

**SYNTAX of array container:**

**deque**< object\_type > *deque\_name*;

### std::deque::push\_back / :: push\_front / :: insert

push\_back(element e) inserts an element **e** at the back of the deque,

 push\_front(element e)inserts the element **e** at the front of the deque.

insert() method has three variations :

* insert(iterator i, element e) : Inserts element **e** at the position pointed by iterator **i** in the deque.
* insert(iterator i, int count, element e) : Inserts element **e**, **count** number of times from the position pointed by iterator **i**.
* insert(iterator i, iterator first, iterator last) : Inserts the element in the range [first,last] at the position pointed by iterator **i** in deque.

#include <iostream>

#include <deque>

#include <vector>

using namespace std;

int main(){

int a[] = { 1, 5, 8, 9, 3 };

deque<int> dq(a, a + 5); /\* creates s deque with elements 1,5,8,9,3 \*/

dq.push\_back(10); /\* now dq is : 1,5,8,9,3,10 \*/

dq.push\_front(20); /\* now dq is : 20,1,5,8,9,3,10 \*/

deque<int>::iterator i;

i = dq.begin() + 2; /\* i points to 3rd element in dq \*/

dq.insert(i, 15); /\* now dq 20,1,15,5,8,9,3,10 \*/

int a[] = { 7, 7, 7, 7 };

d1.insert(dq.begin(), a, a + 4); /\* now dq is 7,7,7,7,20,1,15,5,8,9,3,10 \*/

}

### std::deque::pop\_back / :: pop\_front

pop\_back() removes an element from the back of the deque whereas pop\_front() removes an element from the front of the deque, both decreasing the size of the deque by one.

#include <iostream>

#include <deque>

#include <vector>

using namespace std;

int main(){

int a[] = { 1, 5, 8, 9, 3, 5, 6, 4 };

deque<int> dq(a, a + 8); /\* creates s deque with elements 1,5,8,9,3,5,6,4 \*/

dq.pop\_back();/\* removes an element from the back :: dq is : 1,5,8,9,3,5,6 \*/ dq.pop\_front();/\* now dq is : 5,8,9,3,5,6 \*/

}

### std::deque:: empty / :: size / :: max\_size / :: swap

empty() returns Boolean true if the deque is empty, else Boolean false is returned.

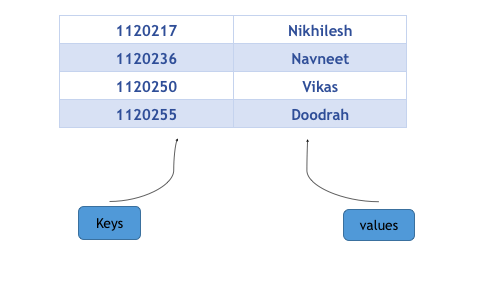
size() returns the number of elements present in the deque.

max\_size() returns the number of element the given deque can hold.

swap()This method can be used to swap elements of two deques.

## Map

* **Maps** are used to replicate associative arrays. Maps contain sorted **key-value** pair.
* In which each key is unique and cannot be changed, and it can be inserted or deleted but cannot be altered.
* Value associated with keys can be altered. We can search, remove and insert in a map within O(n) time complexity.
* For example : A map of students where **roll number** is the key and **name** is the value can be represented graphically as :



Notice that keys are arranged in ascending order, its because maps always arrange its keys in sorted order. In case the keys are of string type, they are sorted lexicographically.

**Creating a Map**

map<key\_type, value\_type> map\_name;

This will create a map with key of type **Key\_type** and value of type **value\_type**. One thing which is to remembered is that key of a map and corresponding values are always inserted as a pair, you cannot insert only key or just a value in a map.

Here is a program that will illustrate creating a map in different ways:

#include <iostream>

#include <map>

using namespace std;

int main()

{

map<int, int> m{ { 1, 2 }, { 2, 3 }, { 3, 4 } };

/\* creates a map m with keys 1,2,3 and

their corresponding values 2,3,4 \*/

map<string, int> map1;

/\* creates a map with keys of type character and

values of type integer \*/

map1["abc"] = 100; // inserts key = "abc" with value = 100

map1["b"] = 200; // inserts key = "b" with value = 200

map1["c"] = 300; // inserts key = "c" with value = 300

map1["def"] = 400; // inserts key = "def" with value = 400

map<char, int> map2(map1.begin(), map1.end());

/\* creates a map map2 which have entries copied

from map1.begin() to map1.end() \*/

map<char, int> map3(m);

/\* creates map map3 which is a copy of map m \*/

}

### std::map::at / :: [ ]

Both **at** and **[ ]** are used for accessing the elements in the map. The only difference between them is that **at** throws an exception if the accessed key is not present in the map, on the other hand operator

**[ ]** inserts the key in the map if the key is not present already in the map.

#include <iostream>

#include <map>

using namespace std;

int main()

{

map<int,string> m{ { 1, ”nikhilesh” }, { 2, ”shrikant” }, { 3, ”ashish” }};

cout << m.at(1); // prints value associated with key 1 ,i.e nikhilesh

cout << m.at(2); // prints value associated with key 2 ,i.e shrikant

/\* note that the parameters in the above at() are the keys not the index \*/

cout << m[3]; // prints value associated with key 3 , i.e ashish

m.at(1) = "vikas"; // changes the value associated with key 1 to “vikas”

m[2] = "navneet"; // changes the value associated with key 2 to “navneet”

m[4] = "doodrah";

/\* since there is no key with value 4 in the map,

it insert a key-value pair in map with key=4 and value = “doodrah” \*/

m.at(5) = "umeshwa";

/\* since there is no key with value 5 in the map ,

it throws an exception \*/

}

### std::map::empty /::size/::max\_size

**empty()** returns boolean true if the map is empty, else it returns Boolean false. **size()** returns number of entries in the map, an entry consist of a key and a value. **max\_size()** returns the upper bound of the entries that a map can contain (maximum possible entries) based on the memory allocated to the map.

### std::map::insert/::insert\_or\_assign

**insert()** is used to insert entries in the map. Since keys are unique in a map, it first checks that whether the given key is already present in the map or not, if it is present the entry is not inserted in the map and the iterator to the existing key is returned otherwise new entry is inserted in the map.

There are two variations of insert() :

* **insert(pair)** : In this variation, a pair of key and value is inserted in the map. The inserted pair is always inserted at the appropriate position as keys are arranged in sorted order.
* **insert(start\_itr , end\_itr)** : This variation inserts the entries in range defined by **start\_itr** and **end\_itr** of another map.

The **insert\_or\_assing()** works exactly as insert() except that if the given key is already present in the map then its value is modified.

#include <iostream>

#include <map>

using namespace std;

int main()

{

map<int, int> m{ { 1, 2 }, { 2, 3 }, { 3, 4 } };

m.insert(pair<int, int>(4, 5));

/\* inserts a new entry of key = 4 and value = 5 in map m \*/

/\* make\_pair() can also be used for creating a pair \*/

m.insert(make\_pair(5, 6));

/\* inserts a new entry of key = 5 and value = 6 \*/

map::iterator i, j;

i = m.find(2); // points to entry having key =2

j = m.find(5); // points to entry having key =5

map<int, int> new\_m;

new\_m.insert(i, j);

/\* insert all the entries which are pointed

by iterator i to iterator j\*/

m.insert(make\_pair(3, 6));

// do not insert the pair as map m already contain key = 3 \*/

m.insert\_or\_assign(make\_pair(3, 6)); // assign value = 6 to key =3

}

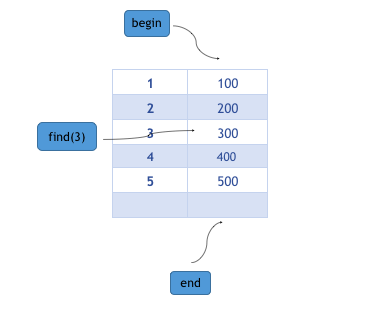
### std::map::erase/::clear

erase() removes the entry from the map pointed by the iterator (which is passed as parameter), however if we want to remove all the elements from the map, we can use clear(), it clears the map and sets its size to 0.

There are two variations of erase :

* erase(iterator\_itr) : This removes entry from the map pointed by iterator **iterator\_itr**, reducing the size of map by 1.
* erase(start\_iterator, end\_iterator) : It removes the elements in range specified by the **start\_iterator** and **end\_iterator**.

### begin, end and find



**begin**, end and find returns an iterator. **begin()** returns the iterator to the starting entry of the map, **end()** returns the iterator next to the last entry in the map and **find()** returns the iterator to the entry having key equal to given key (passed as parameter).