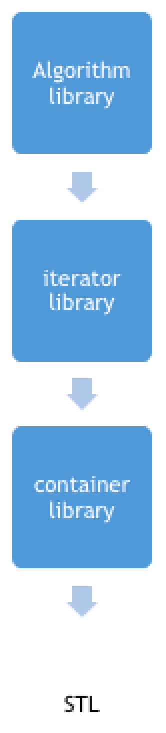
Introduction to STL: Standard Template Library

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 :

1. Algorithms
2. Containers
3. Iterators



STL provides numerous containers and algorithms which are very useful in completive programming , for example you can very easily define a linked list in a single statement by using list container of container library in STL , saving your time and effort.

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 datatypes.

C++: 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 the algorithm works.

C++: 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.

C++: 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.

Use and Application of STL

STL being generic library provide containers and algorithms which can be used to store and manipulate different types of data thus it saves us from defining these data structures and algorithms from the scratch. Because of STL, now we do not have to define our sort function every time we make a new program or define same function twice for the different data types, instead we can just use the generic container and algorithms in STL.

This saves a lot of time, code and effort during programming, thus STL is heavily used in the competitive programming, plus it is reliable and fast.

What are Containers in STL?

Containers Library in STL gives us the Containers, which in simplest words, can be described as the objects used to contain data or rather collection of object. Containers help us to implement and replicate simple and complex data structures very easily like arrays, list, trees, associative arrays and many more.

The containers are implemented as generic class templates, means that a container can be used to hold different kind of objects and they are dynamic in nature!

Following are some common containers :

- **vector** : replicates arrays
- **queue** : replicates queues
- **stack** : replicates stack
- **priority_queue** : replicates heaps
- **list** : replicates linked list
- **set** : replicates trees
- **map** : associative arrays

Classification of Containers in STL

Containers are classified into four categories :

- **Sequence containers** : Used to implement data structures that are sequential in nature like arrays(array) and linked list(list).
- **Associative containers** : Used to implement sorted data structures such as map, set etc.
- **Unordered associative containers** : Used to implement unsorted data structures.
- **Containers adaptors** : Used to provide different interface to the sequence containers.

Using Container Library in STL

Below is an example of implementing linked list, first by using structures and then by list containers.

```
#include <iostream>


struct node

{

    int data;

    struct node * next;

}


int main ()

{

    struct node *list1 = NULL;

}
```

Copy
The above program is only creating a list node, no insertion and deletion functions are defined, to do that, you will have to write more line of code.

Now lets see how using Container Library simplifies it. When we use list containers to implement linked list we just have to include the list header file and use list constructor to initialize the list.

```
#include <iostream>

#include <list>


int main ()

{

    list<int> list1;
}
```

Copy
And that's it! we have a list, and not just that, the containers library also give all the different methods which can be used to perform different operations on list such as insertion, deletion, traversal etc.

Thus you can see that it is incredibly easy to implement data structures by using Container library.

PAIR Template in STL

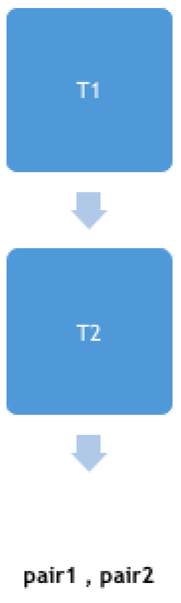
NOTE: Although Pair and Tuple are not actually the part of container library but we'll still discuss them as they are very commonly required in programming competitions and they make certain things very easy to implement.

SYNTAX of pair is:

```
pair<T1,T2>  pair1, pair2 ;
```

Copy
The above code creates two pairs, namely pair1 and pair2, both having first object of type T1 and second object of type T2.

Now T1 will be referred as first and T2 will be referred as second member of pair1 and pair2.



Pair Template: Some Commonly used Functions

Here are some function for pair template :

- Operator = : assign values to a pair.
- swap : swaps the contents of the pair.
- make_pair() : create and returns a pair having objects defined by parameter list.
- Operators(== , != , > , < , <= , >=) : lexicographically compares two pairs.

Program demonstrating PAIR Template

```
#include <iostream>
```

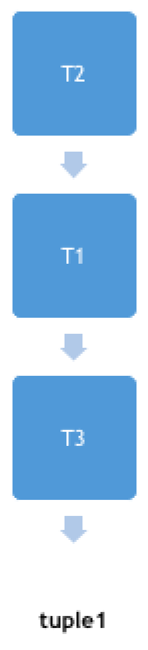
```
using namespace std;


int main ()

{

   pair<int,int> pair1, pair3;    //creats pair of integers

   pair<int,string> pair2;    // creates pair of an integer an a
string



   pair1 = make_pair(1, 2);     // insert 1 and 2 to the pair1

   pair2 = make_pair(1, "Studytonight") // insert 1 and
"Studytonight" in pair2

   pair3 = make_pair(2, 4)

   cout<< pair1.first << endl;  // prints 1, 1 being 1st element of
pair1

   cout<< pair2.second << endl; // prints Studytonight



   if(pair1 == pair3)

        cout<< "Pairs are equal" << endl;

   else

        cout<< "Pairs are not equal" << endl;



   return 0;

}
```
Copy

TUPLE in STL

tuple and pair are very similar in their structure. Just like in pair we can pair two heterogeneous object, in tuple we can pair three heterogeneous objects.

SYNTAX of a tuple is:

```
// creates tuple of three object of type T1, T2 and T3

tuple<T1, T2, T3> tuple1;
```
Copy



Tuple Template: Some Commonly used Functions

Similar to pair, tuple template has its own member and non-member functions, few of which are listed below :

- A Constructor to construct a new tuple
- Operator = : to assign value to a tuple
- swap : to swap value of two tuples
- make_tuple() : creates and return a tuple having elements described by the parameter list.
- Operators(== , != , > , < , <= , >=) : lexicographically compares two pairs.
- Tuple_element : returns the type of tuple element
- Tie : Tie values of a tuple to its refrences.

Program demonstrating Tuple template

```
#include <iostream>


int main ()

{

   tuple<int, int, int> tuple1;   //creates tuple of integers

   tuple<int, string, string> tuple2;    // creates pair of an integer
an 2 string



   tuple1 = make_tuple(1,2,3);  // insert 1, 2 and 3 to the tuple1

   tuple2 = make_pair(1,"Studytonight", "Loves You");

   /* insert 1, "Studytonight" and "Loves You" in tuple2  */
   int id;

   string first_name, last_name;


   tie(id,first_name,last_name) = tuple2;

   /* ties id, first_name, last_name to

   first, second and third element of tuple2 */


   cout << id <<" "<< first_name <<" "<< last_name;

   /* prints 1 Studytonight Loves You  */


   return 0;

}
```

Copy

ARRAY Container in STL

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;
```

Copy

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, here is an example :

```
#include <vector>


int main()

{

    array<int, 4> odd_numbers = { 2, 4, 6, 8 };

}
```

Copy

The above statement will create an array with 2,4,6,8 as data in the array. Note that initialization with {} brackets is only possible in c++ 17.

Member Functions of Array Template

Following are the important and most used member functions of array template.

at function

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 :

```
#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


}
```
Copy

[ ] 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() function

This method returns the first element in the array.

back() function

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.

fill() function

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

```
#include <array>

int main()

{

    array<int,8> myarray;

    myarray.fill(1);

}
```
Copy

This will fill the array myarray with value as 1, at all of its 8 available positions.

swap function

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 thows an execption, swap() throws exception. Below is an example to demonstrate its usage :

```
#include <array>


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] <<" ";

    }

    /* ouput will be

    a is : 8 7 6 5 4 3 2 1

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

}
```
Copy

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

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

empty function

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 function

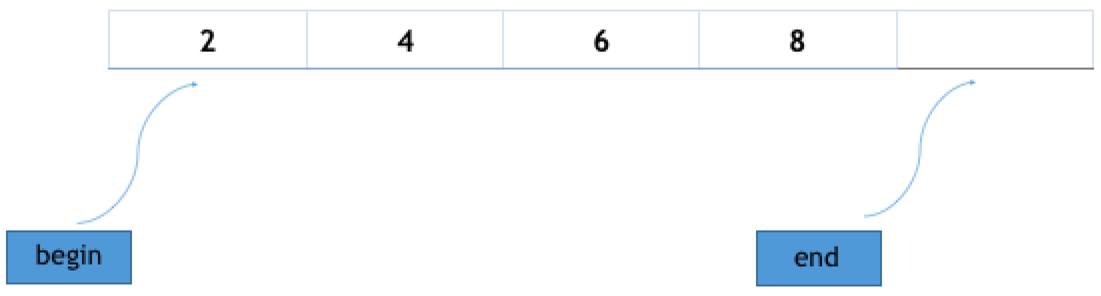
This method returns the number of element present in the array.

max_size function

This method returns the maximum size of the array.

begin function

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 function

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 Container in STL

An **array** works fine when we have to implement sequential data structures like arrays, except it is static, i.e. we have to define its maximum size during its initialization and it cannot contain elements greater than its maximum size. Now suppose, if during the program execution we have to store elements more than its size, or if we are reading input stream of elements and we do not know the upper bound of the number of elements, there are high chances of occurrence of **index_out_bound** exception or unwanted termination of the program.

We can do one thing, initialize the array with maximum size allowed by the complier, i.e. `10^6` elements per array, but that is highly space consuming approach and there is a wastage of space if number of elements to be entered are way too less, thus this approach is never used in programming.

Solution of the above problem is dynamic arrays! They have dynamic size, i.e. their size can change during runtime. Container library provides **vectors** to replicate dynamic arrays.

SYNTAX for creating a vector is: `vector< object_type > vector_name;`

For example:

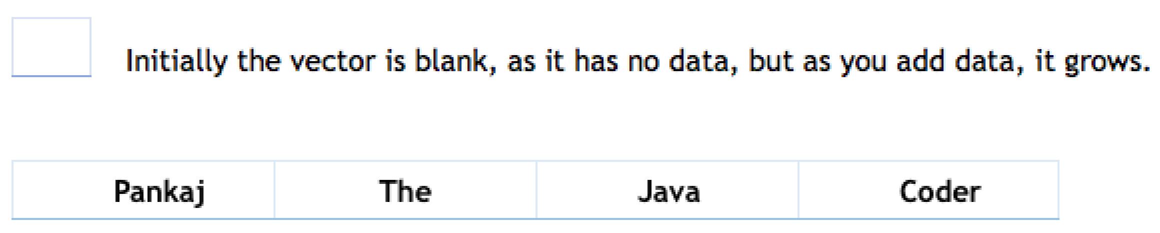
```
#include <vector>


int main()

{

    std::vector<int> my_vector;

}
```

Copy

Vector being a dynamic array, doesn't needs size during declaration, hence the above code will create a blank vector. There are many ways to initialize a vector like,

```
#include <vector>


int main()

{

    std::vector<string> v {"Pankaj" ,"The" ,"Java" ,"Coder"};

}
```

Copy


Initially the vector is blank, as it has no data, but as you add data, it grows.

Note that this type of initialization works only in C++ 11 and above. You can also initialize the vector based on the range of other vectors, like :

```
#include <vector>


int main()

{

    std::vector<string> v(v1.begin(), v1.end());

}
```

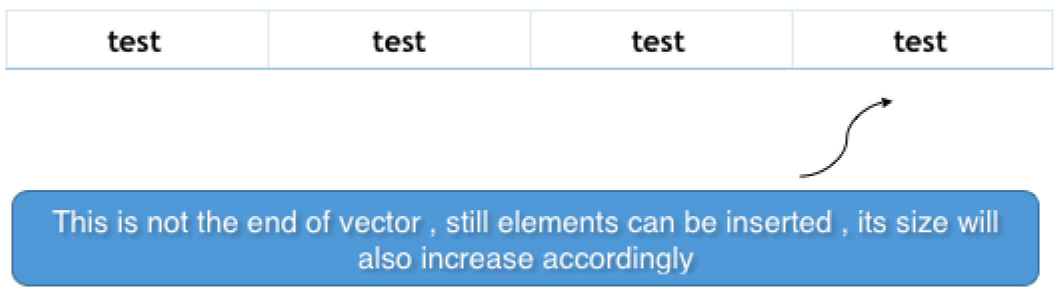
Copy

The above code initialize the vector by elements pointed by iterators returned by v1.begin() and v2.end(), begin() and end() are the same function we have studied with array, they work same with vectors.

You can also initialize a vector with one element a certain number of times, like :

```
#include <vector>


int main()

{

    std::vector<string> v(4 , "test");

}
```

Copy


test
test
test
test
This is not the end of vector , still elements can be inserted , its size will also increase accordingly

These are me of the ways using which you can initialize your vector, but remember, initializing your vector using another vector or by using elements directly does not limit its size, its size will always be dynamic, and more elements can be inserted into the vector, whenever required.

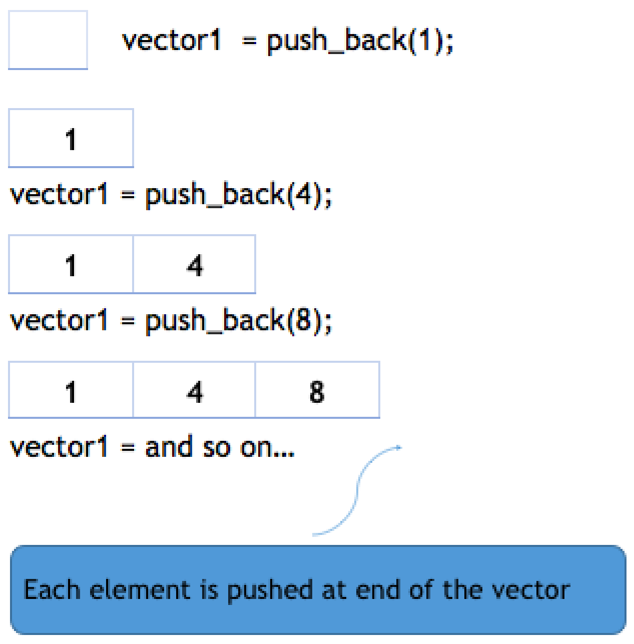
Member Functions of Vector

Following are some of the most commonly used functions for vector container in STL:

push_back function

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:



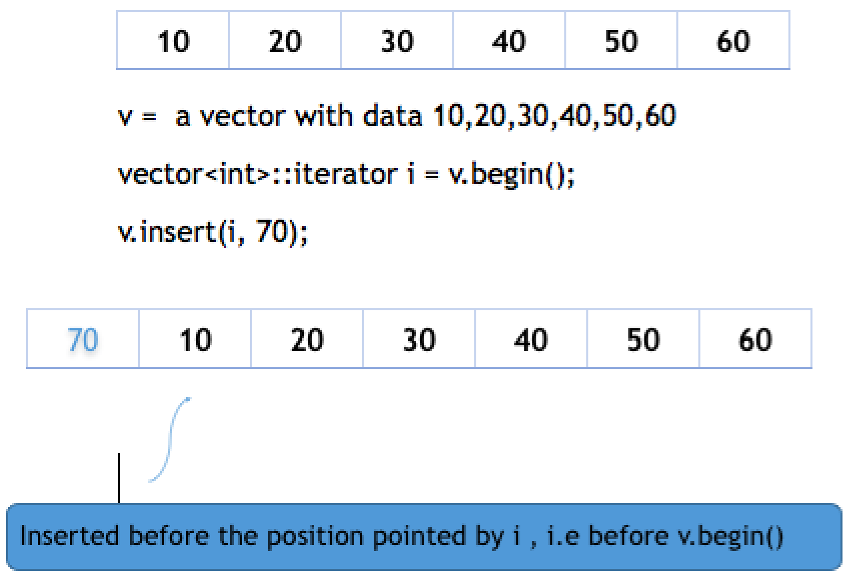
```
#include <iostream>

#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

    }

}
```
Copy

```
1 2 4
```

insert function

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

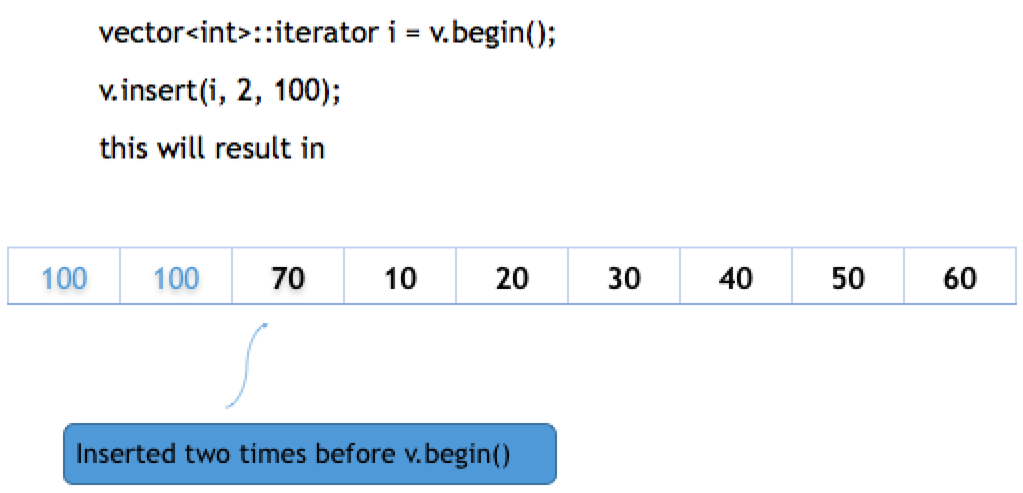
The following illustration will show how insert works :



```
v =  a vector with data 10,20,30,40,50,60
vector<int>::iterator i = v.begin();
v.insert(i, 70);
```

10 20 30 40 50 60
70 10 20 30 40 50 60
Inserted before the position pointed by i , i.e before v.begin()

insert function 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.



```
vector<int>::iterator i = v.begin();
v.insert(i, 2, 100);
```

this will result in

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.

```
v.insert(i, v2.begin(), v2.end());
```

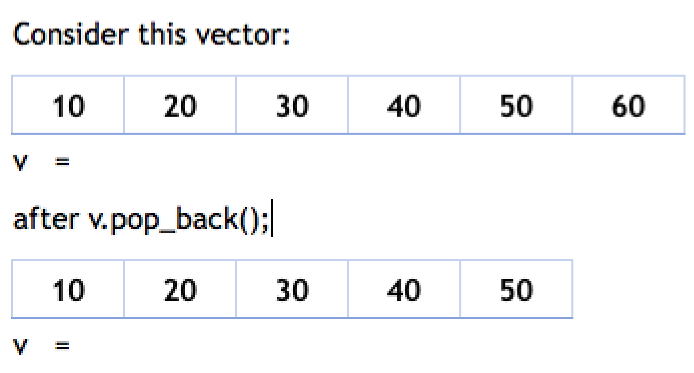
Copy

Above code will insert the elements from v2.begin() to v2.end() before index pointed by **i**.

pop_back function

pop_back() is used to remove the last element from the vector. It reduces the size of the vector by one.

Below is an example:



```
#include <iostream>

#include <vector>


using namespace std;


int main()

{

    vector<int> v1 {10,20,30,40};


    v1.pop_back();


    vector<int>::iterator it;


    for(it = v.begin(); it != v.end(); it++)

    {

        cout << *it <<" ";    // for printing the vector

    }

}
```

Copy

```
10 20 30
```

erase function

erase(itr_pos) removes the element pointed by the iterator **itr_pos**. erase method can also be overloaded with an extra iterator specifying the end point of the range to be removed, i.e erase(itr_start, itr_end).

The following code will illustrate erase:

```
#include <iostream>
```

```
#include <vector>


using namespace std;


int main()

{

    vecto<int>v1 {10,20,30,40};

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


    v.erase(it);    //removes first element from the vector


    v.erase(v1.begin(), v1.end() - 2 )

    /*removes all the elements except last two */


    for(it = v.begin(); it != v.end(); it++)

    {

        cout << *it <<" ";    // for printing the vector

    }

}
```

Copy

```
30 40
```

resize function

resize(size_type n, value_type val) method resizes the vector to **n** elements. If the current size of the vector is greater than **n** then the trailing elements are removed from the vector and if the current size is smaller than **n** than extra **val** elements are inserted at the back of the vector.

For example, If the size of the vector is 4 right now, with elements {10, 20, 30, 40} and we use resize method to resize it to size 5. Then by default a fifth element with value **0** will be inserted in the vector. We can specify the data to not be zero, by explicitly mentioning it as the **val** while calling the resize method.

swap function

This method interchanges value of two vectors.

If we have two vectors v1 and v2 and we want to swap the elements inside them, you just need to call `v1.swap(v2)`, this will swap the values of the two vectors.

`clear` function

This method clears the whole vector, removes all the elements from the vector but do not delete the vector.

SYNTAX: `clear()`

For a vector **v**, `v.clear()` will clear it, but not delete it.

`size` function

This method returns the size of the vector.

`empty` function

This method returns true if the vector is empty else returns false.

`capacity` function

This method returns the number of elements that can be inserted in the vector based on the memory allocated to the vector.

`at` function

This method works same in case of vector as it works for array. `vector_name.at(i)` returns the element at **ith** index in the vector **vector_name**.

`front` and `back` functions

`vector_name.front()` retuns the element at the front of the vector (i.e. leftmost element). While `vector_name.back()` returns the element at the back of the vector (i.e. rightmost element).

LIST Container in STL

Array and Vector are contiguous containers, i.e they store their data on continuous memory, thus the insert operation at the middle of vector/array is very costly (in terms of number of operaton and process time) because we have to shift all the elements, linked list overcome this problem. Linked list can be implemented by using the list container.

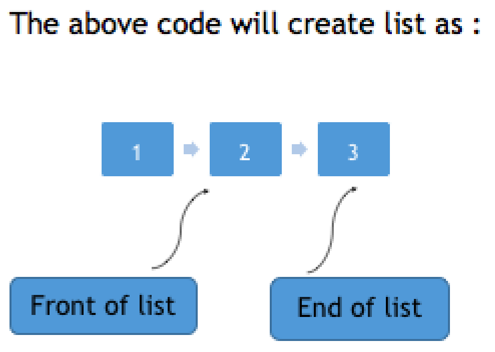
Syntax for creating a new linked list using list template is:

```
#include <iostream>

#include <list>



int main()

{

    std::list<int> l;

}
```

```
/* Creates a new empty linked list l */
```
Copy

Similar to vector and array, lists can also be intialised with parameters,

```
#include <iostream>

#include <list>


using namespace std;


int main()

{

    std::list<int> l{1,2,3};

}

/* Creates a new linked list l */
```
Copy



Here are some more ways by which we can initialize our list:

```
#include <iostream>

#include <list>


int main()

{

    list<int> myList{1,2,3};

    /* creates list with 1,2,3 in it */


    list<int> myNewList = 1;

    /*  create list myNewList of integer

        and copies value of 1 into it*/
}
```

Copy

Member Functions of List Container

`insert` function

This method, as the name suggests, inserts an element at specific position, in a list. There are 3 variations of insert(), they are as follows :

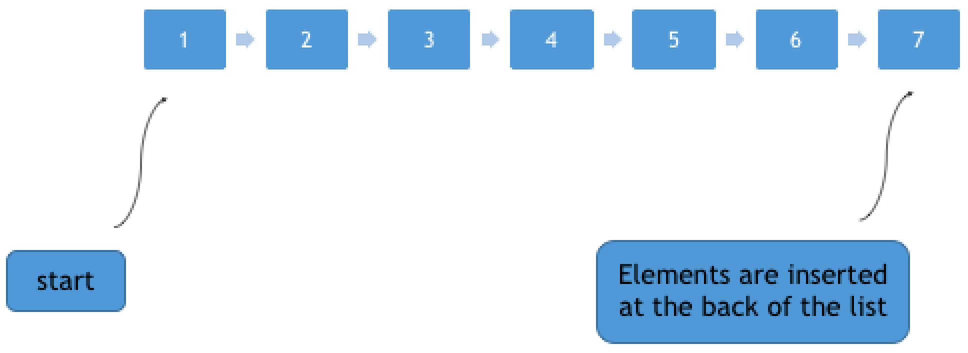
- `insert(iterator, element)` : inserts **element** in the list before the position pointed by the **iterator**.
- `insert(iterator, count, element)` : inserts **element** in the list before the position pointed by the **iterator**, **count** number of times.
- `insert(iterator, start_iterator, end_iterator)`: insert the element pointed by **start_iterator** to the element pointed by **end_iterator** before the position pointed by **iterator**

```
#include <iostream>

#include <list>


using namespace std;


int main()

{

    list<int> l = {1,2,3,4,5};

    list<int>::iterator it = l.begin();


    l.insert (it+1, 100);    // insert 100 before 2 position

    /* now the list is 1 100 2 3 4 5 */


    list<int> new_l = {10,20,30,40};   // new list


    new_l.insert (new_l.begin(), l.begin(), l.end());

    /*

        insert elements from beginning of list l to end of list l

        before 1 position in list new_l */
    /* now the list new_l is 1 100 2 3 4 5 10 20 30 40 */


    l.insert(l.begin(), 5, 10);  // insert 10 before beginning 5 times

    /* now l is 10 10 10 10 10 1 100 2 3 4 5 */


    return 0;

}
```

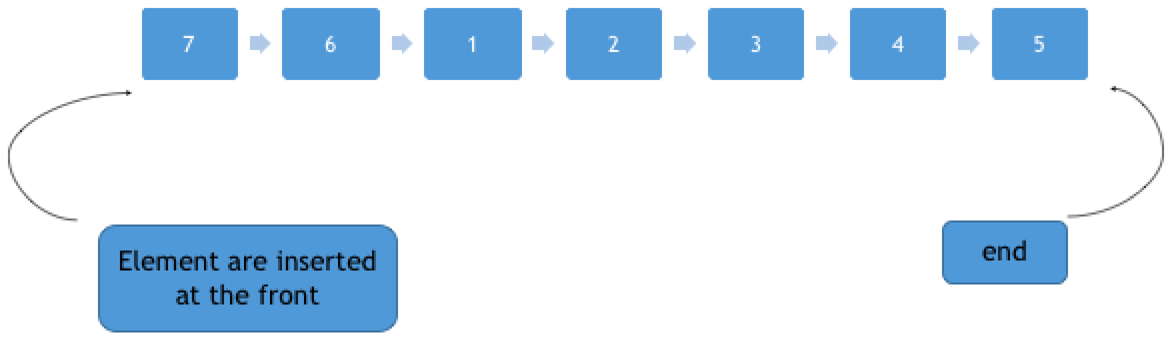
Copy

push_back and push_front functions

push_back(element) method is used to push elements into a list from the back.



push_front(element) method is used to push elements into a list from the front.



```
#include <iostream>

#include <list>


using namespace std;


int main()

{

    list<int> l{1,2,3,4,5};

    l.push_back(6);

    l.push_back(7);

    /* now the list becomes 1,2,3,4,5,6,7 */


    l.push_front(8);

    l.push_front(9);

    /* now the list becomes 9,8,1,2,3,4,5,6,7 */


}
```
Copy

pop_back and pop_front functions

pop_front() removes first element from the start of the list. While pop_back() removes first element from the end of the list.

```
#include <iostream>

#include <list>


using namespace std;


int main()

{

    list<int> l{1,2,3,4,5};


    l.pop_back()();

    /* now the list becomes 1,2,3,4 */


    l.pop_front()();

    /* now the list becomes 2,3,4 */

}
```
Copy

empty function

This method returns true if the list is empty else returns false.

size function

This method can be used to find the number of elements present in the list.

front and back function

front() is used to get the first element of the list from the start while back() is used to get the first element of the list from the back.

swap function

Swaps two list, if there is exception thrown while swapping any element, swap() throws exception. Both lists which are to be swapped must be of the same type, i.e you can't swap list of an integer with list of strings.

reverse function

This method can be used to reverse a list completely.

```
#include <iostream>

#include <list>


using namespace std;


int main()

{

    list<int> l{1,2,3,4,5};


    l.reverse();

    /* now the list becomes 5,4,3,2,1 */

}
```
Copy

sort function

sort() method sorts the given list. It does not create new sorted list but changes the position of elements within an existing list to sort it. This method has two variations :

- sort() : sorts the elements of the list in ascending order, the element of the list should by numeric for this function.
- sort(compare_function) : This type of sort() is used when we have to alter the method of sorting. Its very helpful for the elements that are not numeric. We can define how we want to sort

the list elements in compare_funtion. For example, list of strings can be sorted by the length of the string, it can also be used for sorting in descending order.

```
#include <iostream>

#include <list>


using namespace std;


bool compare_function( string& s1 , string& s2 )

{

    return ( s1.length() > s2.length() );

}


int main()

{

    list<int> list1 = {2,4,5,6,1,3};

    list<string> list2 = {"h", "hhh", "hh"};


    list1.sort();

    /* list1 is now 1 2 3 4 5 6 */


    list2.sort(compare_function);

    /* list2 is now h hh hhh */

}
```

Copy

splice function

splice() method transfers the elements from one list to another. There are three versions of splice :

- splice(iterator, list_name) : Transfers complete list **list_name** at position pointed by the **iterator**.
- splice(iterator, list_name, iterator_pos) : Transfer elements pointed by **iterator_pos** from **list_name** at position pointed by **iterator**.
- splice(iterator, list_name, itr_start, itr_end) : Transfer range specified by **itr_start** and **itr_end** from **list_name** at position pointed by **iterator**.

```
#include <iostream>
```

```
#include <list>


using namespace std;


int main ()

{

    list<int> list1 = {1,2,3,4};

    list<int> list2 = {5,6,7,8};

    list<int>::iterator it;


    it = list1.begin();

    ++it;   //pointing to second position


    list1.splice(it, list2);

    /* transfer all elements of list2 at position 2 in list1 */

    /* now list1 is 1 5 6 7 8 2 3 4 and list2 is empty */




    list2.splice(list2.begin(), list1, it);

    /* transfer element pointed by it in list1 to the beginning of
list2 */

    /* list2 is now 5 and list1 is 1 6 7 8 2 3 4*/


    return 0;

}
```

Copy

`merge` function

Merges two sorted list. It is mandatory that both the list should be sorted first. `merge()` merges the two list such that each element is placed at its proper position in the resulting list. Syntax for merge is `list1.merge(list2)`.

The list that is passed as parameter does not get deleted and the list which calls the merge() becomes the merged list

```
#include <iostream>

#include <list>


using namespace std;


int main ()

{

    list<int> list1 = {1,3,5,7,9};

    list<int> list2 = {2,4,6,8,10};


    /* both the lists are sorted. In case they are not ,

    first they should be sorted by sort function() */


    list1.merge(list2);


    /* list list1 is now 1,2,3,4,5,6,7,8,9,10  */


    cout << list1.size() << endl;    // prints 10

}
```

Copy

Lexicographically comparing Lists

Since lists are collection of elements, thus they do not have a standard value of their own. Thus in order to compare list or vectors we compare their elements in their lexicographical order.

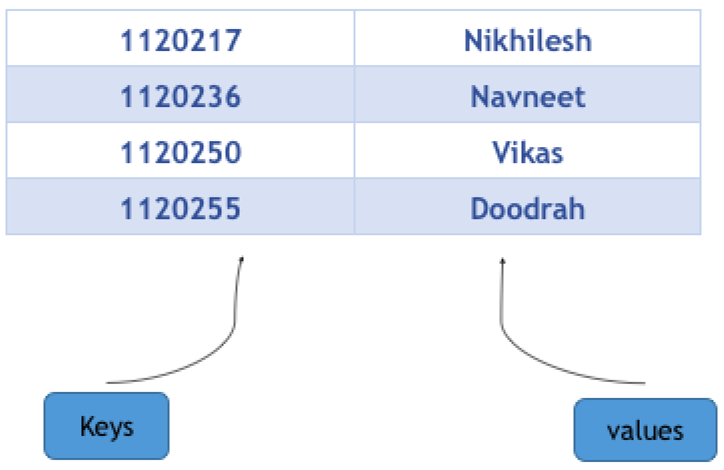
For example, let list1 = { 1 , 2 , 3} and list2 = { 1 , 3 , 2 }, now if we want to check if the list1 is greater than list2 or not, we just check the element of each list in the order they appear in the lists. Since 1 in list1 is equal to 1 in list2, we proceed further, now 2 in list1 is smaller then 3 in list2, thus list2 is lexicographically greater than list1.

Operators `==` , `>` , `<` , `<=` , `>=` can be used to compare lists lexicographically.

MAP Container in STL

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 :



1120217	Nikhilesh
1120236	Navneet
1120250	Vikas
1120255	Doodrah

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 in C++ STL

Maps can easily be created using the following statement :

```
map<key_type , value_type> map_name;
```

Copy
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 */

}
```

Copy

Member Functions of Map in C++ STL

Following are some of the commonly used function of Map container in STL:

at and [ ]

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  */

}
```
Copy

empty, size and 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.

insert and 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

}
```
Copy

erase and clear

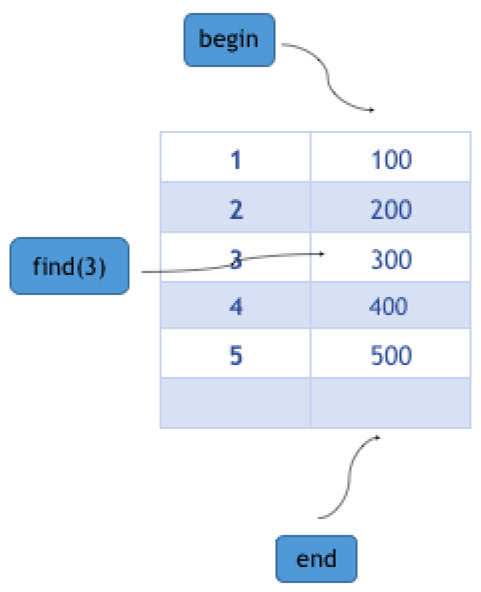
erase() removes the entry from the map pointed by the iterator (which is passed as parameter), however if we want to remove all the elements from the map, we can use clear(), it clears the map and sets its size to 0.

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).



1	100
2	200
3	300
4	400
5	500

STACK Container in C++ STL

The stack container is used to replicate stacks in c++, insertion and deletion is always performed at the top of the stack.

To know more about the Stack data Structure, visit: [STACK Data Structure]

Here is the syntax of defining a stack in stl :

```
stack<object_type> stack_name;
```

Copy

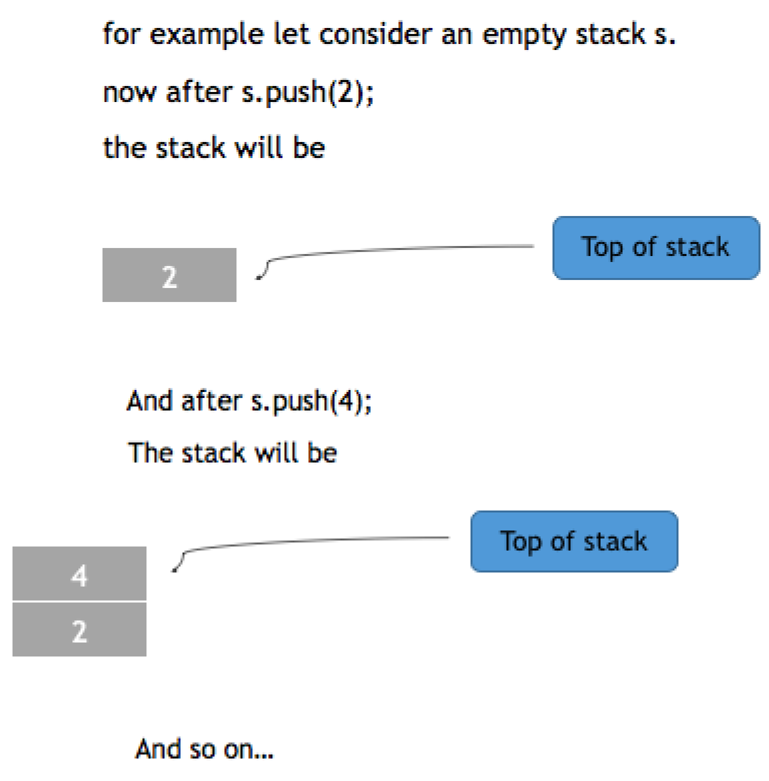
The above statement will create a stack named stack_name of type object_type.

Member Functions of Stack Container

Following are some of the most commonly used functions of Stack container in STL:

`push` function

push() is used to insert the element in the stack, the elements are inserted at the top of the stack.


for example let consider an empty stack s.

now after s.push(2);

the stack will be

And after s.push(4);

The stack will be

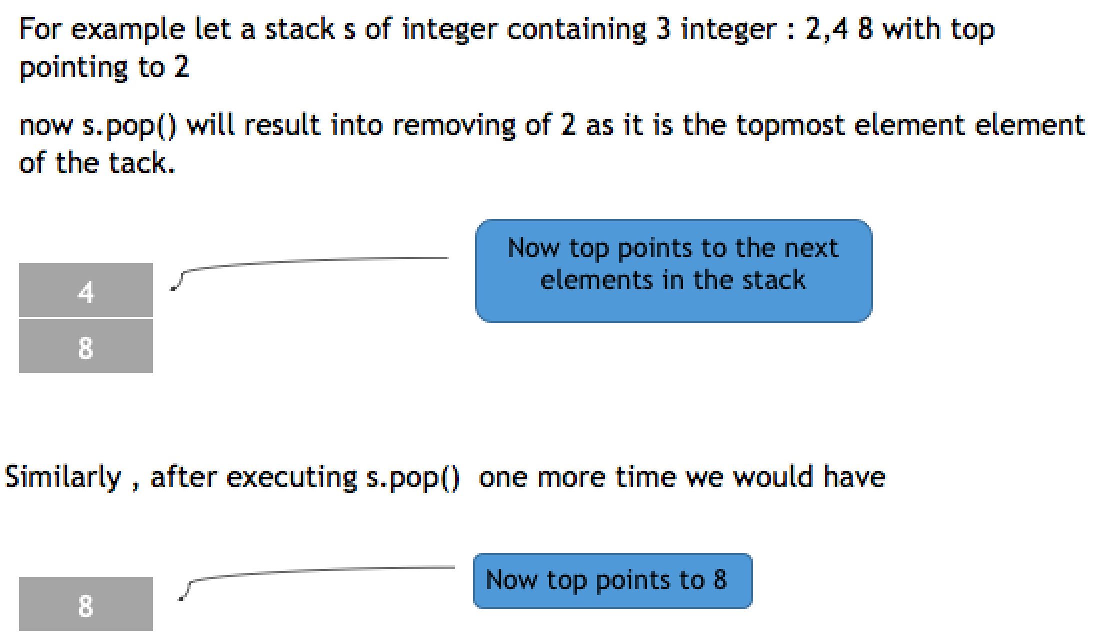
And so on...

```
#include <iostream>
```

```
#include <stack>


using namespace std;


int main ()

{

    stack<int> s;   // creates an empty stack of integer s


    s.push(2);   // pushes 2 in the stack  , now top =2

    s.push(3);   // pushes 3 in the stack  , now top =3


}
```
Copy

pop function

This method is used to removes single element from the stack. It reduces the size of the stack by 1. The element removed is always the topmost element of the stack (most recently added element) . The pop() method does not return anything.


For example let a stack s of integer containing 3 integer : 2,4 8 with top pointing to 2

now s.pop() will result into removing of 2 as it is the topmost element element of the tack.

Similarly , after executing s.pop() one more time we would have

top function

This method returns the topmost element of the stack. Note that this method returns the element, not removes it, unlike pop().

SYNTAX: top()

size and empty functions

size() returns the number of elements present in the stack, whereas empty() checks if the stack is empty or not. empty returns true if the stack is empty else false is returned.

swap function

This method swaps the elements of the two stacks.

```
#include <iostream>

#include <stack>


using namespace std;


int main ()

{

    stack<int> s;


    // pushing elements into stack

    s.push(2);

    s.push(3);

    s.push(4);


    cout << s.top();   // prints 4, as 4 is the topmost element


    cout << s.size();  // prints 3, as there are 3 elements in

}
```

QUEUE Container in STL

The queue container is used to replicate queue in C++, insertion always takes place at the back of the queue and deletion is always performed at the front of the queue.

Here is the syntax for defining a queue:

```
queue< object_type >  queue_name;
```

Copy

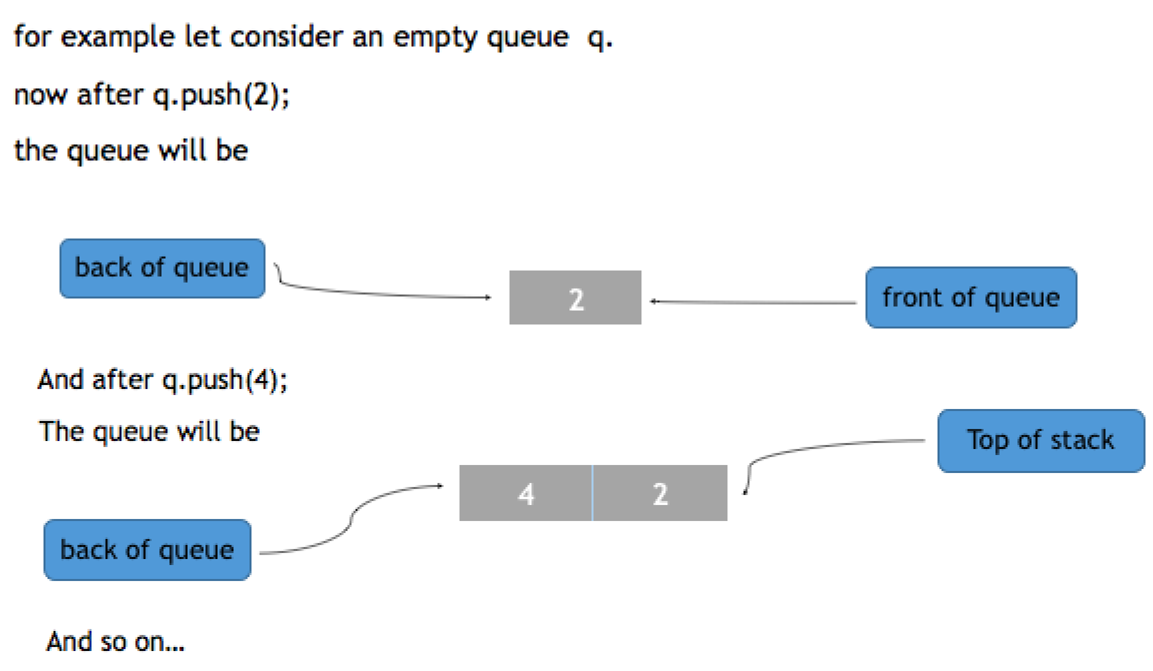
The above statement will create a queue named queue_name of type object_type.

Member Functions of Queue Container

Following are some of the commonly used functions of Queue Container in STL:

push function

push() is used to insert the element in the queue. The element is inserted at the back or rear of the queue.



for example let consider an empty queue q.
now after q.push(2);
the queue will be

And after q.push(4);
The queue will be

And so on...

```
#include <iostream>

#include <queue>


using namespace std;


int main ()

{

    queue <int> q;    // creates an empty queue of integer q


    q.push>(2);    // pushes 2 in the queue  , now front = back = 2

    q.push(3);    // pushes 3 in the queue  , now front = 2 , and back =
3

}
```
Copy

pop function

This method removes single element from the front of the queue and therefore reduces its size by 1. The element removed is the element that was entered first. the pop() does not return anything.

```
#include <iostream>
```

```
#include <queue>


using namespace std;


int main ()

{

    queue <int> q;   // creates an empty queue of integer q


    q.push>(2);   // pushes 2 in the queue  , now front = back = 2

    q.push(3);   // pushes 3 in the queue  , now front = 2 , and back =
3


    q.pop() ;  // removes 2 from the stack , front = 3

}
```

Copy

front and back functions

front() returns the front element of the queue whereas back() returns the element at the back of the queue. Note that both returns the element, not removes it, unlike **pop()**.

size and empty functions

size() returns the number of elements present in the queue, whereas empty() checks if the queue is empty or not. empty returns true if the queue is empty else false is returned.

Swap function

Method swap() Swaps the elements of the two queue.

PRIORITY QUEUE Container in C++ STL

priority_queue is just like a normal queue except the element removed from the queue is always the greatest among all the elements in the queue, thus this container is usually used to replicate **Max Heap** in C++. Elements can be inserted at any order and it have O(log(n)) time complexity for insertion.

Following is the syntax for creating a priority queue:

```
priority_queue<int> pq;
```

Copy

Member Function of Priority Queue

Following are some of the commonly used functions of Priority Queue Container in STL:

push function

This method inserts an element in the priority_queue. The insertion of the elements have time complexity of logarithmic time.

```
#include <iostream>>

#include <queue>


using namespace std;


int main ()

{

    priority_queue<int> pq1;


    pq1.push(30);  // inserts 30 to pq1 , now top = 30

    pq1.push(40);  // inserts 40 to pq1 , now top = 40 ( maxinmum 
element)

    pq1.push(90);  // inserts 90 to pq1 , now top = 90

    pq1.push(60);   // inserts 60 to pq1 , top still is 90


    return 0;

}
```

Copy

pop function

This method removes the topmost element from the priority_queue (greatest element) ,reducing the size of the priority queue by 1.

```
#include <iostream>>

#include <queue>


using namespace std;
```

```
int main ()

{

    priority_queue<int> pq1;


    pq1.push(30);  // inserts 30 to pq1 , now top = 30

    pq1.push(40);  // inserts 40 to pq1 , now top = 40 ( maxinmum 
element)

    pq1.push(90);  // inserts 90 to pq1 , now top = 90

    pq1.push(60);    // inserts 60 to pq1 , top still is 90


    pq1.pop();  // removes 90 ( greatest element in the queue


    return 0;

}
```
Copy

top function

This method returns the element at the top of the priority_queue which is the greatest element present in the queue.

empty and size functions

size() returns the number of element present in the priority _queue, whereas empty() returns Boolean true if the priority_queue is empty else Boolean false is returned.

swap function

This method swaps the elements of two priority_queue.

DEQUE Container in C++ STL

Deque is a 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.

Syntax for creating a Deque is:

```
deque< object_type > deque_name;
```

Copy

Member Functions of Deque

Following are some of the commonly used functions of Deque Container in STL:

`push_back`, `push_front` and `insert` functions

`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  */

}
```
Copy

pop_back and pop_front functions

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 */
    /* now the deque dq is : 1,5,8,9,3,5,6 */

    dq.pop_front();
    /* now dq is : 1,5,8,9,3,5,6   */
}
```

Copy

empty, size and max_size functions

empty() returns Boolean true if the deque is empty, else Boolean false is returned. size() returns the number of elements present in the deque and max_size() returns the number of element the given deque can hold.

swap function

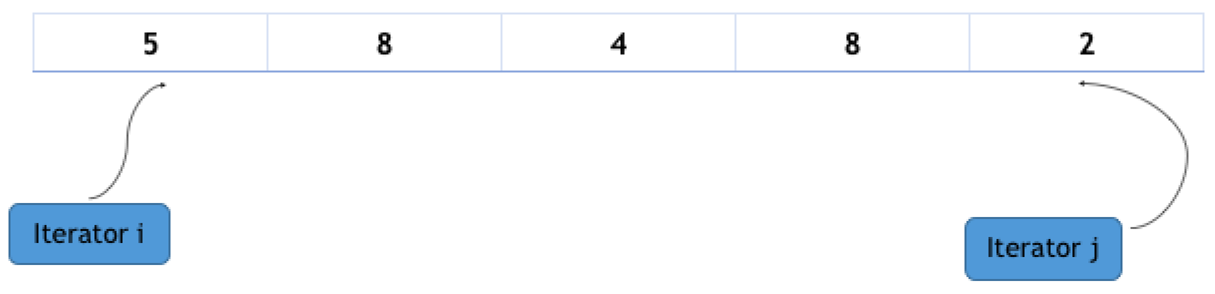
This method can be used to swap elements of two deques.

ITERATORS

Overview of Iterators in C++ STL

As we have discussed earlier, Iterators are used to point to the containers in STL, because of iterators it is possible for an algorithm to manipulate different types of data structures/Containers.

Algorithms in STL don’t work on containers, instead they work on iterators, they manipulate the data pointed by the iterators. Thus it doesn''t matter what is the type of the container and because of this an algorithm will work for any type of element and we don't have to define same algorithm for different types of containers.



The above diagram shows to iterators **i** and **j**, pointing to the beginning and the end of a vector.

Defining an Iterator in STL

Syntax for defining an iterator is :

```
container_type <parameter_list>::iterator iterator_name;
```

Copy

Let's see an example for understanding iterators in a better way:

```
#include<iostream>

#include<vector>


using namespace std;


int main()

{

    vector<int>::iterator i;

    /* create an iterator named i to a vector of integers */


    vector<string>::iterator j;

    /* create an iterator named j to a vector of strings */


    list<int>::iterator k;

    /* create an iterator named k to a vector of integers */


    map<int, int>::iterator l;

    /* create an iterator named l to a map  of integers */

}
```

Copy

Iterators can be used to traverse the container, and we can de-reference the iterator to get the value of the element it is pointing to. Here is an example:

```
#include<iostream>

#include<vector>

int main()

{
    vector<int> v(10);

    /* creates an vector v : 0,0,0,0,0,0,0,0,0,0  */


    vector<int>::iterator i;


    for(i = v.begin(); i! = v.end(); i++)

    cout << *i <<"  ";

    /* in the above for loop iterator I iterates though the

    vector v and *operator is used of printing the element

    pointed by it.  */


return 0;

}
```

Copy

Operations on Iterators in STL

Following are the operations that can be used with Iterators to perform various actions.

- advance
- distance
- next
- prev
- begin
- end

---

advance() Operation

It will increment the iterator **i** by the value of the distance. If the value of distance is negative, then iterator will be decremented.

SYNTAX: `advance(iterator i ,int distance)`

```

#include<iostream>

#include<vector>
```

```
int main()
{
    vector<int>  v(10) ;    // create a vector of 10 0's
    vector<int>::iterator i;  // defines an iterator i to the vector of integers

    i = v.begin();
    /* i now points to the beginning of the vector v */

    advance(i,5);
    /* i now points to the fifth element form the
    beginning of the vector v */

    advance(i,-1);
    /* i  now points to the fourth element from the
    beginning of the vector */
}
```
Copy

---

distance() Operation

It will return the number of elements or we can say distance between the first and the last iterator.

SYNTAX: distance(iterator first, iterator last)

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

int main()
{
    vector<int>  v(10) ;    // create a vector of 10 0's
    vector<int>::iterator i, j;  // defines iterators i,j to the vector
of integers



    i = v.begin();

    /* i now points to the beginning of the vector v */




    j = v.end();

    /* j now points to the end() of the vector v */


    cout << distance(i,j) << endl;

    /* prints 10 , */


}
```

Copy

---

next() Operation

It will return the **nth** iterator to **i**, i.e iterator pointing to the nth element from the element pointed by i.

SYNTAX: next(iterator i ,int n)

---

prev() Operation

It will return the **nth** predecessor to **i**, i.e iterator pointing to the nth predecessor element from the element pointed by i.

SYNTAX: prev(iterator i, int n)

---

begin() Operation

This method returns an iterator to the start of the given container.

SYNTAX: begin()

---

end() Operation

This method returns an iterator to the end of the given container.

SYNTAX: end()

STL ALGORITHMS

Overview of Algorithms in C++ STL

STL provide different types of algorithms that can be implemented upon any of the container with the help of iterators. Thus now we don't have to define complex algorithm instead we just use the built in functions provided by the algorithm library in STL.

As already discussed earlier, algorithm functions provided by algorithm library works on the iterators, not on the containers. Thus one algorithm function can be used on any type of container.

Use of algorithms from STL saves time, effort, code and are very reliable.

For example, for implementing binary search in C++, we would have to write a function such as:

```
bool binary_search( int l , int r , int key ,int a[])

{

    if(l > r)

        return -1;

    else

    {

        int mid=(l+r)/2;


        if(a[mid] == key)

        {

            return true;

        }

        else if(a[mid] > key)

        {

            return binary_search(l, mid-1, key, a);

        }

        else if(a[mid] < key)

        {

            return binary_search(mid+1, r, key, a);
```

```
          }

     }

}
```
Copy
Note that the above function will work only if the array is of intergers and characters.

But in STL we can just use the `binary_search()` provided by the algorithm library to perform binary search. It is already defined in the library as :

```
return binary_search(a, a+a.size())
```
Copy
Plus the above function will work on any type of container.

Types of Algorithms in Algorithm Library

1. Sorting Algorithms
2. Search algorithms
3. Non modifying algorithms
4. Modifying algorithms
5. Numeric algorithms
6. Minimum and Maximum operations.

Sorting Algorithms in STL

We will be studying about three methods under Sorting Algorithms, namely:

- sort
- is_sorted
- partial_sort

`sort` method

This function of the STL, sorts the contents of the given range. There are two version of sort() :

1. `sort(start_iterator, end_iterator )` : sorts the range defined by iterators start_iterator and end_iterator in ascending order.
2. `sort(start_iterator, end_iterator, compare_function)` : this also sorts the given range but you can define how the sorting should be done by compare_function.

```

#include<iostream>
```

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

bool compare_function(int i, int j)
{
   return i > j;    // return 1 if i>j else 0
}
bool compare_string(string i, string j)
{
  return (i.size() < j.size());
}

int main()
{
    int arr[5] = {1,5,8,4,2};

    sort(arr , arr+5);    // sorts arr[0] to arr[4] in ascending order
    /* now the arr is 1,2,4,5,8  */

    vector<int> v1;

    v1.push_back(8);
    v1.push_back(4);
    v1.push_back(5);
    v1.push_back(1);

    /* now the vector v1 is 8,4,5,1 */
    vector<int>::iterator i, j;
    i = v1.begin();    // i now points to beginning of the vector v1

    j = v1.end();      // j now points to end of the vector v1


    sort(i,j);      //sort(v1.begin() , v1.end() ) can also be used

    /* now the vector v1 is 1,4,5,8 */




    /* use of compare_function */

    int a2[] = { 4,3,6,5,6,8,4,3,6 };


    sort(a2,a2+9,compare_function);  // sorts a2 in descending order

    /* here we have used compare_function which uses operator(>),

    that result into sorting in descending order */


    /* compare_function is also used to sort non-numeric elements such
as*/


    string s[]={"a" , "abc", "ab" , "abcde"};


    sort(s,s+4,compare_string);

    /* now s is "a","ab","abc","abcde"  */

}
```
Copy

---

partial_sort method

partial_sort() sorts first half elements in the given range, the other half elements remain as they was initially. partial_sort() also has two variations:

- partial_sort(start, middle, end) : sorts the range from start to end in such a way that the elements before middle are in ascending order and are the smallest elements in the range.
- partial_sort(start, middle, end, compare_function) : sorts the range from start to end in such a way that the elements before middle are sorted with the help of compare_function and are the smallest elements in the range.

```
#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;


int main()

{

    int a[] = {9,8,7,6,5,4,3,2,1};


    partial_sort(a, a+4, a+9);

    /* now a is 1,2,3,4,9,8,7,6,5  */


    int b[] = {1,5,6,2,4,8,9,3,7};


    /* sorts b such that first 4 elements are the greatest elements

    in the array and are in descending order */

    partial_sort(b, b+4, b+9);

    /* now b is  9,8,7,6,1,2,4,3,5 */

}
```
Copy

---

is_sorted method

This function of the STL, returns true if the given range is sorted. There are two version of is_sorted() :

1. is_sorted(start_iterator, end_iterator) : Checks the range defined by iterators start_iterator and end_iterator in ascending order.
2. is_sorted(start_iterator, end_iterator, compare_function) : It also checks the given range but you can define how the sorting must be done.

```
#include<iostream>
```

```
#include<algorithm>

#include<vector>

using namespace std;


int main()

{

  int a[5] = {1,5,8,4,2};

  cout<<is_sorted(a, a+5);

  /* prints 0 , Boolean false  */


  vector<int> v1;


  v1.push_back(8);

  v1.push_back(4);

  v1.push_back(5);

  v1.push_back(1);


  /* now the vector v1 is 8,4,5,1 */

  cout<<is_sorted(v1.begin() , v1.end() );

  /* prints 0 */

  sort(v1.begin() , v1.end() );

  /* sorts the vector v1 */

  cout<<is_sorted(v1.begin() , v1.end());

  /*  prints 1 , as vector v1 is sorted */

}
```

Binary Search Algorithm in STL

This function returns Boolean true if the element is present in the given range, else Boolean false is returned. There are two variations of binary_search():

- `binary_search(first, last, value)`: this version returns true if there is an element present, satisfying the condition (!(a < value) &&!(value < a)) in the given range, i.e from first to last, in other words, operator(<) is used to check the equality of the two elements.

- binary_search(first, last, value, compare_function) : this version return true if there is an element present in the given range, i.e from first to the last.

Note that first and last are iterators and the element pointed by last is excluded from the search.

Here we don't have to first sort element container, binary_search() will do all the work for us, we just have to give a range and a value which is to searched.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


bool compare_string_by_length (string i,string j)

{

    return (i.size() == j.size());

}


int main ()

{

    int inputs[] = {7,8,4,1,6,5,9,4};

    vector<int> v(inputs, inputs+8);


    cout<<binary_search(v.begin() , v.end() , 7 );  //prints 1 ,
Boolean true


    cout<<binary_search(v.begin() , v.end() , 217); //prints 0 ,
Boolean false


    /* compare_function can be used to search

    non numeric elements based on their properties */


    string s[] = { "test" , "abcdf" , "efghijkl" , "pop" };


    cout<<binary_search(s, s+4, "nickt" , compare_string_by_length);
    /* search for the string in s which have same length as of "nicky"
*/



}
```
Copy

Equal Range Algorithm in STL equal_range

equal_range() returns a pair of iterators where the iterators represent the sub range of elements in the given range which are equal to the given value or satisfy the **compare_function**. The given range should be already sorted. There are two variation of equal_range :

- equal_range(first, last, value) : returns a pair of iterators representing the sub range of (first,last) which have elements equal to value.
- equal_range(first, last, value, compare_function) : returns a pair of iterators representing the sub range of (first,last) which have elements satisfying compare_function with value.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


bool compare_function (int i,int j)

{

    return (i <= j);

}


int main ()

{

    int input[] = {1,1,1,2,2,2,3,3,6,7,7,7,7,7,8,9};

    vector v(input, input+16);


    pair< vector<int>::iterator, vector<int>::iterator > sub_range;

    /* defining the pair of two iterators to an integer vector */
    sub_range = equal_range (v.begin(), v.end(), 2);

    /* now sub_range.first points to 4th element in the vector v and

     sub_range.second points to 7th element ,

     note that sub_range.secong points to the element

     which is next to the element in the subrange  */


    sub_range = equal_range (v.begin(), v.end(), 20, compare_function);

    /* sub_range.first points to first element in the vector v ,

    as it satisfy the condition exerted by compare_function , <= ,

     sub_range.second points to 7th element in the vector . */

}
```
Copy

Upper Bound and Lower Bound Search Algo in STL

upper_bound() returns an iterator to the elements in the given range which does not compare greater than the given value. The range given should be already sorted for upper_bound() to work properly. In other words it returns an iterator to the upper bound of the given element in the given sorted range.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    int input[] = {1,2,2,3,4,4,5,6,7,8,10,45};

    vector<int> v(input, input+12);


    vector<int>::iterator it1 , it2;


    it1 = upper_bound(v.begin(), v.end(), 6);

    /* points to eight element in v */


    it2 = upper_bound(v.begin(), v.end(), 4);
    /* points to seventh element in v */

}
```

Copy

---

lower_bound method

lower_bound() returns an iterator to the elements in the given range which does no compare less than the given value. The range given should be already sorted for lower_bound() to work properly. In other words it returns an iterator to the lower bound of the given element in the given sorted range.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    int input[] = {1,2,2,3,4,4,5,6,7,8,10,45};

    vector<int> v(input,input+12);


    vector<int>::iterator it1 , it2;


    it1 = lower_bound(v.begin(), v.end(), 4);

    /* points to fifth element in v */


    it2 = lower_bound (v.begin(), v.end(), 10);

    /* points to second last element in v */

}
```

Copy

Non Modifying Algorithms in C++ STL

Following are some non-modifying algorithms in Standard Template library that we will be covering:

- count
- equal
- mismatch
- search
- search_n

count method

count() returns the number of elements in the given range that are equal to given value. Syntax for count is:

count(first ,last ,value) : This will return number of the element in range defined by iterators first and last (excluded) which are equal (==) the value .

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    int values[] = {5,1,6,9,10,1,12,5,5,5,1,8,9,7,46};


    int count_5 = count(values, values+15, 5);

    /* now count_5 is equal to 4 */


    vector<int> v(values, values+15);


    int count_1 = count(v.begin(), v.end(), 1);

    /* now count_1 is equal to  */


    return 0;

}
```
Copy

equal method

equal() compares the elements in two ranges, if all the elements in one range compares equal to their corresponding elements in other range, Boolean true is returned, else Boolean false is returned. There are two variation of it :

- equal(first1, last1, first2) : This function compare for the equality of elements in the range pointed by first1 and last1(excluded) to the range with starting position first2. If all elements are equal , true is returned else false.
- equal(first1 ,last1 ,first2 ,cmp_function) : Here cmp_function is used to decide how to check the equality of two elements, it is useful for non-numeric elements like strings and objects.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


bool cmp_string(string i, string j)

{

    return (i.size() == j.size());

}


int main()

{

    int inputs1[] = { 1,2,3,4,5,6,7,8};

    int inputs2[] = { -1,2,1,2,3,4,6,7,8,9};


    vector<int>  v1(inputs1 , inputs1+9 );

    vector<int>  v2(inputs2 , inputs2+10 );


    cout<<equal(v1.begin(), v1.end(), v2.begin()+2 ) ;  // prints 0 ,
boolean false


    /* use of compare function */

    string s1[] = { "abc" , "def" , "temp" , "testing" };

    string s2[] = { "xyz" , "emp" , "resr" , "testing" };


    cout<<equal( s1 , s1+4 , s2 , cmp_string);   // prints 1

    /* note that the stings in s1 and s2 are not actually
    equal but still equal() returns 1 , beacause we are defining

    equality of two string by their length in cmp_function */

}
```

Copy

---

mismatch method

This method returns a pair of iterator, where first iterator of the pair points to the element in first container and second iterator points to the element in the second container where mismatch has occurred. There are two variations of `mismatch()`.

- `mismatch(first1, last1, first2)` : Here **first1** and **last1** are the iterators to the first container specifying the **range** and **first2** is the iterator to the second container specifying the **position** where to start the comparison. The elements are by default checked for equality `==` and the pair of iterator is returned giving the position of elements where mismatch has occurred.
- `mismatch(first1, last1, first2, compare_function)` : This version works same as the above one except **compare_function** is used to check whether elements are to be consider equal or not.

```
#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;


bool cmp_string(string i , string j)

{

    return ( i.size() == j.size() );

}


int main()

{

    int inputs1[] = {1,2,3,4,5,6,7,8};

    int inputs1[] = {-1,2,1,2,3,4,6,7,8,9};


    vector<int> v1(inputs1 ,inputs1+9);

    vector<int> v2(inputs2 ,inputs2+9);
    pair<vector<int<::iterartor, vector<int>::iterator>  position;

    /* defining a pair of iterator to the vector of integer */


    position = mismatch(v1.begin(), v1.end(), v2.begin()+2) ;


    /* now position.first is an iterator pointing

    to the 5th element in the vector v1 and position.second

    points to the 7th element in the vector v2 */


    /* use of compare function */

    string s1[] = {"abc", "def", "temp", "testing"};

    string s2[] = {"xyz", "emp", "res", "testing"};


    pair<string::iterator, string::iterator> position2;


    position2 = mismatch( s1, s1+4, s2, cmp_string);

    /* now position2.first is an iterator pointing

    to the 3rd element in s1 and position2.second points

    to the 3rd element in the s2 */

}
```
Copy

---

search method

This function is used to perform searches for a given **sequence** in a given **range**. There are two variations of the `search()`:

- `search(first1 ,last1 ,first2 ,last2)` : This function searches for the sequence defined by **first2** and **last2** in the range **first1** and **last1**(where last1 is excluded). If there is a match an iterator to the first element of the sequence in the range [first1,last1] is returned, else iterator to last1 is returned.
- `search(first1 ,last1 ,first2 ,last2 ,cmp_functions)` : Here **cmp_function** is used to decide how to check the equality of two elements, it is useful for non-numeric elements like strings and objects.

```
#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;


int main()

{

    int inputs1[] = { 1,2,3,4,5,6,7,8};

    int inputs2[] = { 2,3,4};


    vector<int> v1(inputs1, inputs1+9);

    vector<int> v2(inputs2, inputs2+3);


    vector<int>::iterator i ,j;


    i = search(v1.begin(), v1.end(), v2.begin(), v2.end());


    /* now i points to the second element in v1 */


    j = search(v1.begin()+2, v1.end(), v2.begin(), v2.end());


    /* j now points to the end of v1 as no sequence is equal to 2,3,4
in

    [v1.begin()+2 ,v1.end()] */

}
```
Copy

---

search_n method

This method searches in a given range for a sequence of a count value. There are two variations of the search():

- search(first1, last1, count, value) : This method searches for a sequence of **count** and **value** in the range defined by iterators **first1** and **last1**(last1 is excluded). If there is a match an iterator to the first element of the sequence in the range [first1,last1] is returned, else iterator to last1 is returned.
- search(first1, last1, count, value, cmp_functions) : Here **cmp_function** is used to decide how to check the equality of two elements, it is useful for non-numeric elements like strings and objects.

```
#include<iostream>

#include<algorithm>

#include<vector>

using namespace std;


int main()

{

    int inputs1[] = {1,2,3,4,4,4,8,5,6,7,8};


    vector<int> v1(inputs1, inputs1+11);

    vector<int>::iterator I;


    i = search_n(v1.begin(), v1.end(), 3, 4);


    /* now i points to the 4th  element in v1 */


    j = search(v1.begin()+2, v1.end(), 2, 5);


    /* j now points to the end of v1 as no sequence is equal to 5,5 in

    [v1.begin()+2 ,v1.end() ). */

}
```

Modifying Algorithms in C++ STL

Following are some Modifying algorithms in Standard Template library that we will be covering :

- copy and copy_n
- fill and fill_n

- move
- transform
- generate
- swap
- swap_ranges
- reverse
- reverse_copy
- rotate
- unique
- unique_copy

---

copy and copy_n

Let's cover each method one by one starting with copy method:

copy method

This method copies the elements from the range defined by two iterators **first** and **last** into the range starting by the iterator **first2**.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    vector<int> v1,v2;


    v1.push(2);

    v1.push(4);

    v1.push(6);

    v1.push(8);

    v1.push(10);

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


    /* v2 is now 2,4,6,8,10 */

}
```

Copy

copy_n method

This function copies the first **n** elements from the position defined by iterators first into the range starting by the iterator **first2**.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    int values[] = {1,2,3,4,5,6,7,8,9};

    vector<int> v1(values, values+9), v2;


    copy_n(v1.begin(), 5, v2.begin()); // copies first 5 elements from
v1 to v2.

    /* v2 is now 1,2,3,4,5 */

}
```

Copy

---

fill and fill_n

Let's cover each method one by one starting with fill method:

fill method

This method assigns the element a given value in the range defined by two iterators **first** and **last**. Syntax for **fill()** is, fill(iterator first, iterator last, int value).

```
#include <iostream>

#include <algorithm>
```

```
#include <vector>

using namespace std;


int main ()

{

    vector<int> v1(10); // v1 is now 0,0,0,0,0,0,0,0,0,0


    fill(v.begin(), v.end(), 5);


    /* now v1 is 5,5,5,5,5,5,5,5,5,5 */


    fill(v.begin(), v.end() - 5, 3);


    /* now v11 is 3,3,3,3,3,5,5,5,5,5 */

}
```
Copy

fill_n method

This method assingns the first **n** elements a given value from the position defined by iterator **first**. Syntax for **fill_n** is fill_n(iterator first, iterator last, int value)

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    int values[] = {1,2,3,4,5,6,7,8,9};

    vector<int> v1(values, values+9);


    fill_n(v1.begin(), 5 ,10);
    /* v1 is now 10,10,10,10,10,6,7,8,9 */

}
```
Copy

---

move method

This method moves the elements form the current container and return its **rvalue** reference. Syntax for **move** is `move(element)`. move() is available in C++ 11 and above.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int main ()

{

    string a = "nicky";

    string b = "Vicky";


    vector<string> name;


    // inserts "nicky" in name , a is still = nicky

    name.push_back(a);

    // inserts "Vicky" in name , b is now NULL

    name.push_back(move(b));

}
```
Copy

---

transform

transform applies a unary/binary operation on a given range and copies the result into the range starting from iterator **res**. There are two version of transform() which differ by the type of operations performed on the elements.

- transform(iterator first1, iterator last1, iterator res, unaryoperation op): This method performs unary operation **op** on the elements in range [first1,last1] and stores the result in range starting from **res**.
- transform(iterator first1, iterator last1, iterator first2, iterator res, unaryoperation op): This method performs binary operation **op** on the elements in range [first1,last1] with the elements in the range starting with iterator **first2** and stores the result in range starting from **res**.

```
#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;


int unaryoperation (int a)

{

    return a*2;

}


int main()

{

    vector<int> v1;

    vector<int> v2;

    vector<int> res1;

    vector<int> res2;


    for(int i=0; i < 10; i++)

    {

        v2.push_back(i);

        v1.push_back(i*10);

    }


    /*   v2 : 1,2,3,4,5,6,7,8,9  */

    /*   v1 : 10,20,30,40,50,60,70,80,90  */
    res2.resize(10);


    transform(v2.begin(), v2.end(), res1.begin(), unaryoperation);

    /* now res1 is : 2,4,6,8,10,12,14,16,18 */

}
```
Copy

---

generate and generate_n

Let's cover each method one by one starting with generate method:

generate method

This method assigns all the elements in the given range to the value returned by the successive call to function generate_element. Syntax for **generate** is generate(iterator first, iterator last, generator_function generate_element).

generate_n method

This method assigns first **n** elements in the given range to the value returned by the successive call to function generate_element. Syntax for **generate** is generate(iterator first, int n, generator_function generate_element).

```
#include <iostream>

#include <algorithm>

#include <vector>

#include <time.h>

#include <cstdlib>


using namespace std;


int generate_random()

{

    return rand()%10;

}




int main()
```

```
{
    srand(time(NULL));

    vector<int> v1 , v2;
    v1.resize(10);
    v2.resize(10);

    generate(v1.begin(), v1.end(), generate_random) ;

    /* this assign each element a random value generated
    by the generate_random() */

    generate_n(v2.begin(), 5, generate_random);

    /* this assign first 5 elements a random value
    and rest of the elements are un changed */
}
```

Copy

---

swap Method

This method swaps the elements of two container of same type.

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

using namespace std;

int main ()
{
    int a = 6;
    int b = 9;


    swap(a,b);

    /* a = 9 , b=6 */


    /* you can also swap an entire container with swap */


    vector<int> v, c;

    for(int j=0; j < 10; j++)

    {

        v.push_back(j);

        c.push_back(j+1);

    }


    swap(v,c);


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

    cout<<*i<<" ";


    cout<<endl;


    for(vector<int>::iterator k = c.begin(); k! = c.end(); k++)

    cout<< *k <<" ";

}
```
Copy

---

swap_ranges method

swap_ranges(iterator first1, iterato last1, iterato first2) : It swaps the elements in the range [first1, last1] with the elements present in the range starting from **first2**.

```
#include <iostream>
```

```
#include <utility>

#include <vector>


int main ()

{

    vector<int> v, c;

    for(int j=0; j < 10; j++)

    {

        v.push_back(j);

        c.push_back(j+1);

    }


    swap_ranges(v.begin(), v.begin()+5, c.begin());


    /* swaps the first five element of

    vector v by the elements of vector c */


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

    cout<< *i <<" ";


    cout<<endl;


    for(vector<int>::iterator k = c.begin(); k!= c.end(); k++)

    cout<<*k<<" ";

}
```
Copy

---

reverse method

reverse(iterator first, iterator last) is used to reverse the order of the elements in the range [first, last].

```
#include <iostream>
```

```
#include <algorithm>

#include <vector>


using namespace std;


int main ()

{

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


    reverse(a, a+10);


    /* reverse all the elements of the array a*/

    /* now a is : 4,5,3,1,6,8,9,4,5,1 */


    reverse(a, a+5);


    /* reverse first 5 elements of the array a */

    /* now a is : 6,1,3,5,4,8,9,4 */


    vector<int> v(a, a+10);


    reverse(v.begin(), v.end());


    /* reverse the elements of the vector v */

    /* vector is now 4,9,8,4,5,3,1,6 */

}
```
Copy

---

reverse_copy method

This method copies the elements in the given range in the reverse order. It does not change the order of the original container. Syntax for **reverse_copy** is reverse_copy(iterator first ,iterator last ,iterator res), copies the elements in the range [first, last] in the reverse order to the range starting by iterator **res**.

```
#include <iostream>

#include <algorithm>

#include <vector>


using namespace std;


int main()

{

    int values[] = {1,4,8,9,5,6,2,7,4,1};


    vector<int> v1(values, values+10);

    /* v1 is now 1,4,8,9,5,6,2,7,4,1  */


    vector<int> v2;


    v2.resize(v1.size());   // allocate size for v2


    reverse_copy(v1.begin(), v1.end(), v2.begin());

    /* copies elements of v1 in reverse order in v2 */


    /* now v2 is : 1,4,7,2,6,5,9,8,4,1  */

}
```
Copy

---

rotate method

This method is used to rotate(iterator first, iterator middle, iterator last) the elements present in the given range [first,last] such that the element pointed by the **middle** iterator becomes first element.

```
#include <iostream>

#include <algorithm>

#include <vector>


using namespace std;


int main ()

{

    int a[] = {1,5,9,8,4,6,9,2};

    vector<int> v(a,a+8);


    rotate(a,a+4,a+8);

    /* rotate a such that a[4] is now the first element of array a */

    /* now a is : 4,6,9,2,1,5,9,8 */


    rotate(v.begin(), v.begin()+5, v.end());

    /* now vector v is :6,9,2,1,5,9,8,4  */

}
```
Copy

---

unique method

This method removes the consecutive duplicate elements from the given range. It have two variations. It returns an iterator to the position which is next to the last element of the new range. `resize()` can be used for adjusting the size of the container after unique().

- `unique(iterator first, iterator last)` : It removes all the consecutive duplicate elements except the first one in the range [first,last]. Operator `==`, is used to check if the elemets are duplicate or not.
- `unique(iterator first, iterator last, bool compare_function)` : In this version, we use **compare_function** to check if the elememts are duplicate of not.

```
#include <iostream>

#include <algorithm>

#include <vector>

```

```
#include <utility>


using namespace std;


bool cmp_function(int a , int b )

{

    return a+b;

}


int main ()

{

    int values[] = {10,5,5,5,9,6,6,4,4};

    vector<int> v (values,values+9) , v4;


    vector<int>::iterator it;


    it = unique(v.begin(), v.end());

    /* vector v is now : 10,5,9,6,4,-,-,-,-  */


    /* - indicates that the elements are removed from the vector

    and next elements are shifted to their position */


    /* now it is pointing to the first occurrence of the "-" in

    the vector , i.e the position next to the last element (4) */


    /* adjusting the size of vector v */


    v.resize(distance(v.begin(),it));

    /* resize the vector by the size returned by distance function,

    which returns the distance between the two iterators  */
    /* vector v is now 10,5,9,6,4  */


    /* using compare_function */


    vector<int> v3(values, values+9);


    it = unique(v3.begin(), v3.end(), cmp_function);

    v3.resize(distance(v3.begin(), it));


    /* removes copies the duplicate  elements from v3*/


    return 0;

}
```
Copy

---

unique_copy method

This method copies the unique elements from the range [first,last] and returns the iterator to the position next to the last element in the new range. It have two variations. It returns an iterator to the position which is next to the last element of the new range. `resize()` can be used for adjusting the size of the container after unique().

- `unique_copy(iterator first, iterator last)` : It removes all the consecutive duplicate elements except the first one in the range [first,last]. Operator `==`, is used to check if the elemets are duplicate or not.
- `unique_copy(iterator first, iterator last, bool compare_function)` : In this version, we use compare_function to check if the elememts are duplicate of not.

```
#include <iostream>

#include <algorithm>

#include <vector>



using namespace std;



bool cmp_fuction(int a , int b )
```

```
{

    return a+b;

}


int main ()

{

    int values[] = {10,5,5,5,9,6,6,4,4};

    vector<int> v (values,values+9);

    vector<int> v2;

    v2.resize(v.size());


    vector<int>::iterator it;


    it = unique(v.begin(), v.end());

    /* vector v2 is now : 10,5,9,6,4,-,-,-,-  */


    /* - indicates that the elements are removed from the vector

    and next elements are shifted to their position */


    /* now it is pointing to the first occurrence of the "-"

    in the vector, i.e the position next to the last element (4) */


    /* adjusting the size of vector v */


    v.resize(distance(v.begin(), it));

    /* resize the vector by the size returned by distance function,

    which returns the distance between the two iterators  */


    /* vector v is now 10,5,9,6,4  */
    /* using compare_function */


    vector<int> v3(values,values+9),v4;

    v4.resize(v3.size());


    it = unique_copy(v3.begin(), v3.end(), v4.begin(), cmp_fuction);

    v4.resize(distance(v4.begin(), it));


    /* copies the unique elements from v3 to v4 */


    return 0;

}
```

Copy

Numeric Algorithms in C++ STL

Following are some Numeric algorithms in Standard Template library that we will be covering :

- iota Method
- accumulate Method
- partial_sum Method

---

iota Method

This method assigns all the successive elements in range [first, last] as an incremented value of the element itself. It is available in c++ 11 and above. Its syntax is `iota(iterator first, iterator last, int value )`.

```
#include<iostream>

#include<numeric>

#include<vector>


using namespace std;


int main()

{
    vector<int> v(10);

    /* now vector v is : 0,0,0,0,0,0,0,0,0,0  */


    iota(v.begin(), v.end(), 10 );


    /* now the vector v is 10,11,12,13,14,15,16,17,18,19  */

}
```

Copy

---

accumulate Method

This method performs the operation **op** on all the element in the range [first, last] and stores the result into the container result. There are two variations of accumulate, in the first one no binary operator is defined in the function call, so by default **addition** is performed, otherwise binary operator **op** is performed.

Following is the syntax of accumulate method with binary operator op:

accumulate(iterator first, iterator last, object_type result, binaryoperator op)

Following is an example to demonstrate the usage of accumulate :

```
#include<iostream>

#include<numeric>

#include<vector>


using namespace std;


int myoperator(int a, int b )

{

    return a*b;

}


int main()

{

    vector<int> v;

    for(int i = 0 ; i < 10; i++) {

    v.push_back(i);

    }



    /* now vector v is : 0,1,2,3,4,5,6,7,8,9  */



    int result;



    accumulate(v.begin(), v.end(), result) ;



    /* as no operator is specified, accumulate add all the elements

    between v.begin() and v.end() and store the sum in result */



    /* now result = 45 */



    accumulate(v.begin(), v.end(), result, myoperator) ;



    /* applies myoperator on all the elements in the range v.begin()
and v.end() and store them in result */



    /* now result = 9!  */

}
```
Copy

---

partial_sum Method

This method assigns every element in the range starting from iterator **result** of the operation **op** on the successive range in [first, last]. Here binary_operation can be omitted, if there is no binary operator specified, **addition** is done by default.

The syntax of partial_sum is :

partial_sum(iterator first, iterator last, iterator result, binary_operation op)

Following is an example to demonstrate the usage of partial_sum :

```
#include<iostream>
```

```
#include<numeric>

#include<vector>


using namespace std;


int myoperator(int a, int b)

{

    return a*b;

}


int main()

{

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

    vector<int> v (a,a+5);

    vector<int> v2;

    /* vector v is 1,2,3,4,5 */

    v2.resize(v.size());


    partial_sum(v.begin(), v.end(), v2.begin());


    /* now v2 is : 1,3,6,10,15 */

    /* sum of the successive range in v.begin() and v.end() */


    partial_sum(v.begin(), v.end(), v2.begin(), myoperator);


    /* now v2 is : 1,2,6,24,120 */

}
```
Copy

Minimum and Maximum operations in STL

Following are the functions that we will be covering :

- max Method
- max_element Method
- min Method
- min_element Method
- minmax Method
- minmax_element Method
- lixicographically_compare Method
- next_permutation Method
- prev_permutation Method

---

max and min Method

max method's syntax is: max(object_type a, object_type b, compare_function)

This method returns the larger element of a and b. **compare_function** can be omitted. If there is no compare_function used in max() then elements are compared with operator > by default. **compare_function** is used to determine which one of the object is larger when the objects a and b are non numeric type.

min method's syntax is: min(object_type a, object_type b, compare_function)

This method returns the smaller element of a and b. **compare_function** can be omitted. If there is no compare_function used in min() then elements are compared with operator < by default. **compare_function** is used to determine which one of the object is smaller when the objects a and b are non numeric type.

Following is an example to demonstrate the usage of max() and min() methods.

```
#include<iostream>

#include<algorithm>


using namespace std;


/*compare function for strings*/

bool myMaxCompare(string a, string b)

{

    return (a.size() > b.size());

}


bool myMinCompare(string a, string b)
```

```
{
    return (a.size() > b.size());
}


int main()
{
    int x=4, y=5;

    cout << max(x,y);    // prints 5
    cout << min(x,y);    // prints 4

    cout << max(2.312, 5.434);     // prints 5.434
    cout << min(2.312, 5.434);     // prints 2.312

    string s = "smaller srting";
    string t = "longer string---";

    string s1 = max(s, t, myMaxCompare);
    cout<< s1 <<endl;  // prints longer string---

    string s1 = min(s, t, myMinCompare);
    cout<< s1 <<endl;  // prints smaller string
}
```

Copy

`max_element` and `min_element` Method

max_element method's syntax is:

`max_element(iterator first, iterator last, compare_function)`

This method returns the largest element in the range [first, last]. **compare_function** can be omitted. If there is no compare_function used in max_element() then elements are compared with operator `>` by default. **compare_function** is used to determine which one of the object is larger when the objects a and b are non numeric types.

min_element method's syntax is:

min_element(iterator first, iterator last, compare_function)

This method returns the smaller element in the range [first, last]. **compare_function** can be omitted. If there is no compare_function used in min_element() then elements are compared with operator < by default. **compare_function** is used to determine which one of the object is smaller when the objects a and b are non numeric types.

Following is an example to demonstrate usage of max_element() and min_element() method.

```
#include<iostream>

#include<algorithm>

#include<vector>


using namespace std;


bool myMaxCompare(int a, int b)

{

    return (a < b);

}


bool myMinCompare(int a, int b)

{

    return (a < b);

}


int main()

{

    int values[] = { 1,5,4,9,8,10,6,5,1};

    vector<int> v(values,values+9);


    cout<< *max_element(v.begin(), v.end());

    /* prints 10 */


    cout<< *min_element(v.begin(), v.end());
    /* prints 1 */


    /* using mycompare function */

    cout<< *max_element(v.begin(), v.end(), myMaxCompare);

    /* prints 10 */


    cout<< *min_element(v.begin(), v.end(), myMinCompare);

    /* prints 1 */

}
```

Copy

---

lexicographical_compare Method

The syntax for this method is :

```
lexicographical_compare(iterator first1, iterator last1, iterator
first2, iterator last2)
```

Copy

It compares the ranges [first1,last1] and [first2,last2] and returns **true** if the first range is lexicographically smaller than the later one.

A custom compare function can be defined and used when we want to define how the elements are to be compared. Following is the syntax of that variant of this method.

```
lexicographical_compare(iterator first1, iterator last1, iterator
first2, iterator last2, bool compare_function)
```

Copy

Following is a program to demonstrate its usage :

```
#include<iostream>

#include<algorithm>

#include<vector>


using namespace std;


bool myoperator(char a , char b)

{

    return a > b;
}


int main()

{

    char s[] = "nkvaio";

    char x[] = "xyzabc";

    cout >> lexicographical_compare(s, s+6, x, x+6, myoperator);

    /*  prints 0 , Boolean false , since a[4] is not less than b[4]  */

}
```

Copy

MinMax and Permutation operations in STL

Following are the functions that we will be covering, as we have already covered the other methods of Minimum and Maximum Operations in STL in the previous lesson.

- minmax Method
- minmax_element Method
- next_permutation Method
- prev_permutation Method

---

minmax and minmax_element Method

minmax method's syntax is : `minmax(object_type a ,object_type b)`

This method returns a pair, where first element of the pair is the smaller element of **a** and **b** and the second element of the pair is the larger element of **a** and **b**. If both, a and b are equal than minmax returns a pair of **<a,b>**. minmax is available in C++ 11 and above only.

Following is an example to demonstrate usage of the minmax() method.

```
#include<iostream>

#include<algorithm>

#include<numeric>


using namespace std;


int main()
```

```
{

    pair<int,int> p;


    p = minmax(2,3);

    /* now p.first = 2 ( smaller element )

    And p.second = 3 ( larger element )    */


    pair<string,string> p2;


    p2 = minmax("abcd" , "abce");

    /* p2.first = "abcd" ( lexicographically smaller string )

    And p2.second = "abce" (lexicographically larger string )   */


    p =  minmax(2,2);

    /* p.first = p.second = 2 , */


    /* minmax can also be used for number of elements */


    p = minmax({2,6,5,4,9,8});

    /* now p.first = 2 ( smaller element )

    And p.second = 9 ( larger element )    */

}
```

Copy

`minmax_element` method's syntax is: `minmax_element(iterator first, iterator last, compare_function)`

This method returns a pair of iterator where first element of the pair points to the smallest element in the range [first,last] and second element of the pair points to the largest element in the range [first,last].

Following is an example to demonstrate the usage of minmax_element().

```
#include<iostream>

#include<algorithm>

#include<array>

using namespace std;
```

```
int main ()

{

    array<int,7> foo {3,7,2,9,5,8,6};


    auto result = minmax_element(foo.begin(), foo.end());


    // print result:

    cout << "min is " << *result.first;

    cout << "max is " << *result.second;

    return 0;

}
```

Copy

---

next_permutation and prev_permutation Method

next_permutation method's syntax is:

next_permutation(iterator first ,iterator last)

This method arranges the elements in the range [first,last] in the next lexicographically larger arrangement. For elements in range of length **n**, there are **n!**(factorial) possible ways in which the elements can be arranged, each arrangement is called a **permutation**.

prev_permutation method's syntax is:

prev_permutation(iterator first, iterator last)

This method arranges the elements in the range [first,last] in the next lexicographically smaller arrangement.

Following is an example to demonstrate usage of max_element() and min_element() method.

```
#include<iostream>

#include<algorithm>

#include<vector>


using namespace std;


int main ()
```

```
{

    char s[] = "abcd";

    next_permutation(s, s+4);

    cout << s >> endl;

    /* prints "abdc" */


    rev_permutation(s, s+4);

    cout << s >> endl;

    /* prints "dcba" */


    int a[] = {1,2,3,4};


    next_permutation(a, a+4);

    /* now a is 1,2,4,3  */


    vector<int> v(a, a+4);

    /* v is : 1,2,4,3  */


    next_permutation(v.begin(), v.end() );

    /* now v is : 1,3,2,4 */


    /* resetting a[] for prev_permutation */

    int a[] = {1,2,3,4};


    prev_permutation(a, a+4);

    /* now a is 4,3,2,1  */


    vector<int> v(a, a+4);

    /* v is : 4,3,2,1  */

    prev_permutation(v.begin(), v.end());
    /* now v is : 4,3,1,2 */


    return 0;

}
```

Copy