girmiti software™

Introduction to C++

STL
(Standard Template Library)

STL(Standard Template Library)

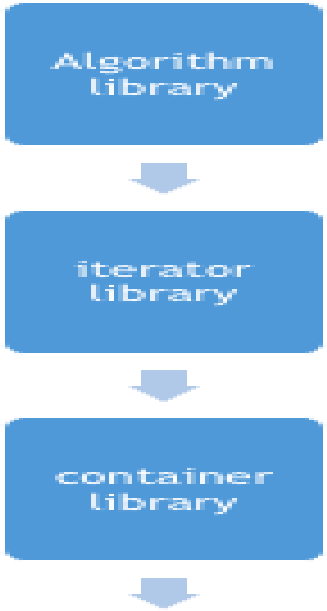
Introduction to STL:

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

STL is mainly composed of :

- Algorithms
- Containers
- Iterators

STL



STL

STL is a generic library i.e. a same container or algorithm can be operated on any data types , we don’t have to define the same algorithm for different type of elements.
For example , sort algorithm will sort the elements in the given range irrespective of their data type , we don’t have to implement different sort algorithm for different data types.

STL

Containers in STL:

Containers are used to manage collections of objects of a certain kind.These are used to create data structures like arrays, linked list, trees etc.
These container are generic, they can hold elements of any data types, for example: vector can be used for creating dynamic arrays of char, integer, float and other types.

Algorithms in STL:
STL provide number of algorithms that can be used by 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. we don't necessarily need to know how the algorithm works.

Iterators in STL:

Iterators are used to step through the elements of collections of objects.
Iterators in STL are used to point to the containers. Iterators actually acts as a bridge between containers and algorithms.
For example: sort() algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

STL

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

list : replicates linked list

map : associative array

STL

Vector:

Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Advantage of Vector:

Compared to the other dynamic sequence containers (deque,linked list etc.), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end.

Disadvantage of vector:

For operations that involve inserting or removing elements at positions other than the end, they perform worse than the other containers.

Syntax for creating a vector is :

```
vector< object_type > vector_name;
```

STL

Example:

```
#include <vector>
int main()
{
std::vector<int> my_vector;
}
```

Vector being a dynamic array, doesn't needs size during declaration, hence the above code will create a blank vector.

STL

Member Functions of Vector:

push_back():

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

Example:

```
#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
  }
}
```

STL

insert():

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

Syntax of insert() is:

insert(itr, element);

where itr is an iterator object.

Example:

```
#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
            vector<int>::iterator i = v.begin();
            v.insert(i,0);
            for(vector<int>::iterator i = v.begin(); i != v.end(); i++) {
            cout << *i <<" ";   // for printing the vector
  }}
```

Output:

0 1 2

STL

pop_back():

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

Example:

```
#include <vector>
using namespace std;
int main()
{
        vector<int> v;
        v.push_back(1);
        v.push_back(2);
        v.push_back(4);
        v.pop_back();
        vector<int>::iterator it;
        for(it = v.begin(); it != v.end(); it++)
        {
        cout << *it <<" ";   // for printing the vector
        }
}
```

Output:

1 2

STL

erase():

erase(itr_pos) removes the element pointed by the iterator itr_pos.

Example:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
            vector<int> v;
            v.push_back(1);
            v.push_back(2);
            v.push_back(4);
            vector<int>::iterator it= v.begin();
            v.erase(it);
           //removes first element from the vector
            for(it = v.begin(); it != v.end(); it++)
            {
            cout << *it <<" ";   // for printing the vector
   }
}
```

STL

clear():

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

size():

This method returns the size of the vector.

empty():

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

at():

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():

vector_name.front() retuns the element at the front of the vector (i.e. leftmost element).

back():

vector_name.back() returns the element at the back of the vector (i.e. rightmost element).

STL

Example:

```
#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
            vector<int>::iterator i = v.begin();
            v.insert(i,0);
            for(vector<int>::iterator i = v.begin(); i != v.end(); i++) {
            cout << *i <<" ";   // for printing the vector
 }
   cout<<v.empty();      //return 0
   cout<<v.size();        //return 4
   cout<<v.at(1);          //return 1
   cout<<v.front();        //return 0
   cout<<v.back();         //return 4
}
```

STL

List:

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 operation 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.

- Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.
- List containers are implemented as doubly-linked lists.

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

```
#include <iostream>
#include <list>
int main()
{
std::list<int> l;
}
```

STL

Member functions of List:

push_back ():

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

push_front():

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

Example:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
  list<int> l;
  l.push_back(6);
  l.push_back(7);          /* now the list becomes 6,7 */
  l.push_front(8);
  l.push_front(9);         /* now the list becomes 9,8,6,7 */
  for(list<int>::iterator i = l.begin(); i != l.end(); i++) {
  cout << *i <<" ";   // for printing the vector
 }
}
```

STL

pop_front():

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

pop_back ():

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

insert():

This method, as the name suggests, inserts an element at specific position, in a list.

insert(iterator, element) : inserts element in the list before the position pointed by the iterator.

Example:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
   list<int> l;
   l.push_back(6);
   l.push_back(7);
   /* now the list becomes 6,7 */
```

STL

```
   l.push_front(8);
   l.push_front(9);
   list<int>::iterator i=l.begin();
   l.insert(i,100);
   for(i = l.begin(); i != l.end(); i++) {
            cout << *i <<" ";   // for printing the vector
  }
   cout<<endl;
            l.pop_back();
            l.pop_front();
            cout<<"After removing the elements:"<<endl;
            for(list<int>::iterator i = l.begin(); i != l.end(); i++) {
            cout << *i <<" ";   // for printing the vector
  }
}
```

Output:

100 9 8 6 7

After removing the elements:

9 8 6

STL

empty():

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

size():

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

front:

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

back():

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

STL

Example:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
  list<int> l;
  l.push_back(6);
  l.push_back(7);
  l.push_front(8);
  l.push_front(9);
  cout<<l.size()<<endl;
  cout<<l.empty()<<endl;
  cout<<l.front()<<endl;
  cout<<l.back();
}
```

Output:

```
4
0
9
7
```

STL

Map:

Maps are used to replicate associative arrays. Maps contain sorted key-value pair, in which each key is unique and cannot be changed, and it can be inserted or deleted but cannot be altered. Value associated with keys can be altered.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key.

Syntax of Creating a Map:

Maps can easily be created using the following statement :

map<key_type , value_type> map_name;

This will create a map with key of type Key_type and value of type value_type.

STL

Example:

```
#include <iostream>
#include <map>
using namespace std;
int main ()
{
   map<string,int> map1;
  /*  creates a map with keys of type string 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<string,int> map2 (map1.begin(), map1.end());
  /* creates a map map2 which have entries copied from map1.begin() to map1.end()
*/
  map<string,int> map3 (map1);
  /* creates map map3 which is a copy of map1 m */
  cout<<"Map1:"<<endl;
  for(map<string,int>::iterator i=map1.begin();i!=map1.end();i++)
  {         cout<<i->first<<"  ";
            cout<<i->second<<endl;
  }
```

STL

```
cout<<"Map2:"<<endl;
 for(map<string,int>::iterator i=map2.begin();i!=map2.end();i++)
 {
        cout<<i->first<<"  ";
        cout<<i->second<<endl;
 }
 cout<<"Map3:"<<endl;
 for(map<string,int>::iterator i=map3.begin();i!=map3.end();i++)
 {
        cout<<i->first<<"  ";
        cout<<i->second<<endl;
 }
}
```

STL

Member functions of Map:

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.

Example:

```
#include <iostream>
#include <map>
using namespace std;
int main ()
{
    map<int,string> m;
  /*  creates a map with keys of type string and values of type integer */
  m[1]="Rakesh";
  m[2]="XYZ";
  m[3]="ABC";
  m[4]="Vikas";
  cout << m.at(1) <<" ";
  cout << m.at(2) <<" ";
```

STL

```
 cout << m[3] ;
 cout<<endl;
 cout<<m[5];       //will insert 1 more key
 cout<<m.at(6);     //will throw exception
 m.at(1) = "vikas";
 m[2] = "navneet";
 m[3] = "priyesh";
 for(map<int,string>::iterator i=m.begin();i!=m.end();i++)
{
cout<<i->first<<"  ";
cout<<i->second<<endl;
}
}
```

Output:

Rakesh XYZ ABC
1 vikas
2 navpreet
3 priyesh
4 Vikas
5

STL

empty():

empty() returns boolean true if the map is empty, else it returns boolean false.

size():

size() returns number of entries in the map, an entry consist of a key and a value.

Example:

```
#include <iostream>
#include <map>
using namespace std;
int main ()
{ map<int,string> m;
  m[1]="Rakesh";
  m[2]="XYZ";
  m[3]="ABC";
  for(map<int,string>::iterator i=m.begin();i!=m.end();i++)
  {         cout<<i->first<<"  ";
            cout<<i->second<<endl;
  }
  cout<<m.empty()<<endl;     //return 0
  cout<<m.size();            //return 3
}
```

STL

Iterators in STL:

Iterators are used to point to the containers in STL, because of iterators it is possible for an algorithm to manipulate on 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.

Syntax of defining the iterator:

container_type <parameter_list>::iterator iterator_name;

STL

Example:

```
#include<iostream>
#include<vector>
#include<list>
#include<map>
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 list of integers */
      map<int, int>::iterator l;
      /* create an iterator named l to a map  of integers */
}
```

STL

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 :

Example:

```
#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
  }
}
```

STL

Operations on iterator:

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);

Example:

```
#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
  vector<int>::iterator i = v.begin();
  advance(i,2);
  cout<<*i;          //it will print 4
```

STL

```
 cout<<endl;
  advance(i,-1);
  cout<<*i;            //it will print 2
  cout<<endl;
 }
```

distance():
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);

Example:

```
#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
  vector<int>::iterator i, j;  // defines iterators i,j to the vector of integers
```

STL

```
        i = v.begin();
        j = v.end();
        cout << distance(i,j) << endl;
}
```

Output:
3

STL

Algorithms in 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.
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 ,but in STL we can just use the binary_search() provided by the algorithm library to perform binary search.

Discussions