BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City

Object Orented Programming in C++ Department

Unit-6 Standard Template Library

6.1 C++ Standard Template Library (STL)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.
- A working knowledge of template classes is a prerequisite for working with STL.
- 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.

STL has three components

1. **Algorithms**
2. **Containers**
3. **Iterators**

6.2. Algorithms

- 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.
- The algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Example: Sort and Binary Search in C++ Standard Template Library (STL)

- Sorting is one of the most basic functions applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing.
- There is a builtin function in C++ STL by the name of sort().
- This function internally uses IntroSort. In more details it is implemented using hybrid of QuickSort, HeapSort and InsertionSort. By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort.

```
// C++ progrma to sort an array
#include <algorithm>
#include <iostream>
 using namespace std;
 void show(int a[], int array_size)
{
    for (int i = 0; i < array_size; ++i)
        cout << a[i] << " ";
}
int main()
{
    int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
    int asize = sizeof(a) / sizeof(a[0]);         // size of the array
    cout << "The array before sorting is : \n";
```

```
    show(a, asize);                         // print the array
    sort(a, a + asize);                     // sort the array using STL
    cout << "\n\nThe array after sorting is :\n";
    show(a, asize);
    cout << "\n\nNow, we do the binary search";
    if (binary_search(a, a + 10, 2))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    cout << "\n\nNow, say we want to search for 10";
    if (binary_search(a, a + 10, 10))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";
                            // print the array after sorting
 return 0;
}
```

Output:

```
The array before sorting is ::
1,5,8,9,6,7,3,4,2,0,
The array after sorting is :
0,1,2,3,4,5,6,7,8,9,

Now, we do the binary search
Element found in the array

Now, say we want to search for 10
Element not found in the array
```

6.3 Containers

Containers or container classes store objects and data. There are in total seven standard "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

1. Sequence Containers: It implements data structures which can be accessed in a sequential manner.

- vector
- list
- deque
- arrays
- forward_list

i. Vector:

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- In vectors, data is inserted at the end.
- Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
- Removing the last element takes only constant time because no resizing happens.
- Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions associated with the vector are:

- **begin()** – Returns an iterator pointing to the first element in the vector
- **end()** – Returns an iterator pointing to the theoretical element that follows the last element in the vector
- **rbegin()** – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
- **rend()** – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

- **cbegin()** – Returns a constant iterator pointing to the first element in the vector.
- **cend()** – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
- **crbegin()** – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
- **crend()** – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

Vector Example:

```
// iterators in vector
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int> g1;
   for (int i = 1; i <= 5; i++)
      g1.push_back(i);
   cout << "Output of begin and end: ";
   for (auto i = g1.begin(); i != g1.end(); ++i)
      cout << *i << " "
   cout << "\nOutput of cbegin and cend: ";
   for (auto i = g1.cbegin(); i != g1.cend(); ++i)
      cout << *i << " ";
   cout << "\nOutput of rbegin and rend: ";
   for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
      cout << *ir << " ";
   cout << "\nOutput of crbegin and crend : ";
   for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
      cout << *ir << " ";
   return 0;
}
```

Output:

Output of begin and end: 1 2 3 4 5

Output of cbegin and cend: 1 2 3 4 5

Output of rbegin and rend: 5 4 3 2 1

Output of crbegin and crend : 5 4 3 2 1

ii. List:

- Lists are sequence containers that allow non-contiguous memory allocation.
- As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.
- Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

Example: Below is the program to show the working of some functions of List:

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;
  //function for printing the elements in a list
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main()
{
    list <int> gqlist1, gqlist2;
    for (int i = 0; i < 10; ++i)
    {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);
```

```
    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();
    cout << "\ngqlist1.reverse() : ";
    gqlist1.reverse();
    showlist(gqlist1);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);
    return 0;
}
```

Output:

```
List 1 (gqlist1) is :    0    2    4    6
8    10    12    14    16    18

List 2 (gqlist2) is :    27    24    21    18
15    12    9    6    3    0

gqlist1.front() : 0
gqlist1.back() : 18

gqlist1.reverse() :    18    16    14    12
10    8    6    4    2
gqlist2.sort():    3    6    9    12
15    18    21    24    27
```

2. Container Adapters: It provides a different interface for sequential containers.

- queue
- priority_queue
- stack

Queue

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement.

Elements are inserted at the back (end) and are deleted from the front.

Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

The functions supported by queue are:

- **empty()** – Returns whether the queue is empty.
- **size()** – Returns the size of the queue.
- **queue::swap()** in C++ STL: Exchange the contents of two queues but the queues must be of same type, although sizes may differ.
- **queue::emplace()** in C++ STL: Insert a new element into the queue container, the new element is added to the end of the queue.
- **push(g) and pop()** – push() function adds the element 'g' at the end of the queue. pop() function deletes the first element of the queue.
- **queue::front() and queue::back()** in C++ STL– front() function returns a reference to the first element of the queue. back() function returns a reference to the last element of the queue.

```
// CPP code to illustrate
// Queue in Standard Template Library (STL)
#include <iostream>
#include <queue>
using namespace std;
void showq(queue<int> gq)
{
	queue<int> g = gq;
	while (!g.empty()) {
		cout << '\t' << g.front();
		g.pop();
	}
	cout << '\n';
}
int main()
{
	queue<int> gquiz;
	gquiz.push(10);
	gquiz.push(20);
```

```
    gquiz.push(30);
    cout << "The queue gquiz is : ";
    showq(gquiz);
    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.front() : " << gquiz.front();
    cout << "\ngquiz.back() : " << gquiz.back();
    cout << "\ngquiz.pop() : ";
    gquiz.pop();
    showq(gquiz);
    return 0;
}
```

Output:

```
The queue gquiz is :     10    20    30
gquiz.size() : 3
gquiz.front() : 10
gquiz.back() : 30
gquiz.pop() :     20    30
```

3. Associative Containers: It implement sorted data structures that can be quickly searched (O(log n) complexity).

- set
- multiset
- map
- multimap

Set:

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it.

The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Some basic functions associated with Set:

- begin() – Returns an iterator to the first element in the set.
- end() – Returns an iterator to the theoretical element that follows last element in the set.

- size() – Returns the number of elements in the set.
- max_size() – Returns the maximum number of elements that the set can hold.
- empty() – Returns whether the set is empty.

```
#include <iostream>
#include <iterator>
#include <set>
using namespace std;
int main()
{
        // empty set container
        set<int, greater<int> > s1;
        // insert elements in random order
        s1.insert(40);
        s1.insert(30);
        s1.insert(60);
        s1.insert(20);
        s1.insert(50);
        // only one 10 will be added to the set
        s1.insert(50);
        s1.insert(10);
        // printing set s1
        set<int, greater<int> >::iterator itr;
        cout << "\nThe set s1 is : \n";
        for (itr = s1.begin(); itr != s1.end(); itr++)
        {
                cout << *itr<<" ";
        }
        cout << endl;
        // assigning the elements from s1 to s2
        set<int> s2(s1.begin(), s1.end());
        // print all elements of the set s2
```

```
        cout << "\nThe set s2 after assign from s1 is : \n";
        for (itr = s2.begin(); itr != s2.end(); itr++)
        {
                cout << *itr<<" ";
        }
        cout << endl;
        // remove all elements up to 30 in s2
        cout<< "\ns2 after removal of elements less than 30 :\n";
        s2.erase(s2.begin(), s2.find(30));
        for (itr = s2.begin(); itr != s2.end(); itr++)
        {
                cout <<*itr<<" ";
        }
        cout << endl;
        return 0;
}
```

4. Unordered Associative Containers: It implements unordered data structures that can be quickly searched

- unordered set (Introduced in C++11)
- unordered_multiset (Introduced in C++11)
- unordered_map (Introduced in C++11)
- unordered_multimap (Introduced in C++11)

Unordered Set:

An unordered_set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized.

```
// C++ program to demonstrate various function of unordered_set
#include <bits/stdc++.h>
using namespace std;
int main()
{
        // declaring set for storing string data-type
```

```
    unordered_set <string> stringSet ;
    // inserting various string, same string will be stored once in set
    stringSet.insert("code") ;
    stringSet.insert("in") ;
    stringSet.insert("c++") ;
    stringSet.insert("is") ;
    stringSet.insert("fast") ;

    string key = "slow" ;
    // find returns end iterator if key is not found, else it returns iterator to that key

    if (stringSet.find(key) == stringSet.end())
        cout << key << " not found" << endl << endl ;
    else
        cout << "Found " << key << endl << endl ;

    key = "c++";
    if (stringSet.find(key) == stringSet.end())
        cout << key << " not found\n" ;
    else
        cout << "Found " << key << endl ;

    // now iterating over whole set and printing its content
    cout << "\nAll elements : ";
    unordered_set<string> :: iterator itr;
    for (itr = stringSet.begin(); itr != stringSet.end(); itr++)
        cout << (*itr) << endl;
}
```

6.4 Iterators

- As the name suggests, iterators are used for working upon a sequence of values.
- They are the major feature that allows generality in STL.
- An iterator is an object (like a pointer) that points to an element inside the container.
- We can use iterators to move through the contents of the container.
- They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them.
- Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers.
- The most obvious form of an iterator is a pointer.
- A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers.
- Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

Operations of iterators:-

- 1. begin() :- This function is used to return the beginning position of the container.
- 2. end() :- This function is used to return the after end position of the container.
- 3. advance() :- This function is used to increment the iterator position till the specified number mentioned in its arguments.
- 4. next() :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.
- 5. prev() :- This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.
- 6. inserter() :- This function is used to insert the elements at any position in the container. It accepts 2 arguments, the container and iterator to position where the elements have to be inserted.

```
// C++ code to demonstrate the working of
// iterator, begin() and end()
```

```
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
   vector<int> ar = { 1, 2, 3, 4, 5 };
   // Declaring iterator to a vector
   vector<int>::iterator ptr;
   // Displaying vector elements using begin() and end()
   cout << "The vector elements are : ";
   for (ptr = ar.begin(); ptr < ar.end(); ptr++)
      cout << *ptr << " ";

   return 0;
}
```

Output:

The vector elements are: 1 2 3 4 5

BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City