Experiment No: 11 Date:

Aim: To study Standard template library in C++

Theory:

1. Containers

STL provides various container classes like vectors, lists, and maps. Containers store and manage collections of objects, offering different interfaces and performance characteristics to suit diverse programming needs.

2. Iterators

Iterators are used to traverse through elements in containers. They provide a uniform way to access data regardless of the underlying container type, enhancing the flexibility and genericity of algorithms.

3. Algorithms

STL includes a rich set of algorithms that operate on containers. These algorithms cover tasks such as sorting (`std::sort`), searching (`std::find`), and manipulation (`std::transform`). They promote code reuse and readability.

4. Functions and Functors

STL leverages function objects (functors) and function pointers to customize algorithms. This allows developers to pass behavior to algorithms, making them adaptable to different data types and processing requirements.

5. Strings

The `std::string` class simplifies string manipulation in C++. It provides a dynamic and convenient alternative to traditional character arrays, offering extensive functionality for string handling.

6. Streams

STL stream classes (`std::ifstream`, `std::ofstream`) facilitate input and output operations, abstracting the complexities of interacting with files. They support formatted and unformatted I/O, making file processing more straightforward.

7. Memory Allocation

Dynamic memory allocation is managed by STL allocators (`std::allocator`). These allocators provide a flexible way to control memory resources, promoting efficient memory management in containers and algorithms.

8. Generic Programming

STL promotes generic programming through templated classes and functions. This enables developers to create versatile and reusable code that works seamlessly with various data types and structures.

9. Smart Pointers

STL includes smart pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`) that automate memory management. They enhance resource safety by providing controlled ownership and automatic deallocation of memory.

10. Exception Handling

STL components often handle errors using exceptions. Proper exception handling is crucial for robust programming with STL, ensuring graceful recovery from unexpected situations and enhancing code reliability.

11. Algorithms Complexity

Understanding the time and space complexity of STL algorithms is essential for making informed choices. STL provides a standardized way to express the efficiency of algorithms, aiding in algorithm selection based on specific requirements.

[A] Write a C++ program to implement standard library vector sequence container

```
//Vector Sequence Container
#include<iostream>
#include<conio.h>
#include<vector>
using namespace std;
void display(vector <int> &);   // display function prototype
int main()
{
   vector <int> v;     // integer vector created

   cout<<"\n\nInitial size() = "<<v.size();    // gives no of elements
   cout<<"\n\nInitial capacity() = "<<v.capacity(); // capacity returns no elements that vector can store b4 that vector needs to dynamically resize itself to accommodate more elements

   v.push_back(10);    // pushing the element at back of vector
   v.push_back(20);
   v.push_back(30);
   v.push_back(40);
   v.push_back(50);

   cout<<"\n\nAfter push_back() size() = "<<v.size();
   cout<<"\n\nAfter push_back() capacity() = "<<v.capacity();

   cout<<"\n\nDisplay vector elements after push_back() :";
   display(v);

   cout<<"\n\nFrist element of vector = "<<v.front();
   cout<<"\n\nLast element of vector = "<<v.back();

   //Inserting elements in vector using iterator
   vector<int>::iterator itr=v.begin();    //here itr is pointing to 0th element of v
   itr = itr + 5;  // itr made to point 4th element;
   v.insert(itr,60);   // insert 40 as 4th element of v


   cout<<"\n\nDisplay vector elements after insertion :";
   display(v);

   //pop_back() function to delete last element
   v.pop_back();

   cout<<"\n\nDisplay vector elements after pop_back() :";
   display(v);

   // erase(delete) vector elements
   v.erase(v.begin()+2,v.begin()+4);   // erase(2,4) = deletes 30 & 40 but not 50

   cout<<"\n\nDisplay vector elements after erase() :";
   display(v);

   //resizing vector
   v.resize(10);
   cout<<"\n\nAfter resize() vector size = "<<v.size();

   //using clear function
   v.clear();

   cout<<"\n\nAfter clear() function :";
   display(v);

   cout<<"\n\nIs vector empty = "<<v.empty();

   getch();
   return 0;
}
void display(vector <int> & v)
{
   for(int i=0;i<v.size();i++)
   {
      cout<<" "<<v.at(i); // at() prints vector element at each reference index
```

```
  }
}
```

Output:

```
Initial size() = 0

Initial capacity() = 0

After push_back() size() = 5

After push_back() capacity() = 8

Display vector elements after push_back() : 10 20 30 40

Frist element of vector = 10

Last element of vector = 50

Display vector elements after insertion : 10 20 30 40 5

Display vector elements after pop_back() : 10 20 30 40

Display vector elements after erase() : 10 20 50

After resize() vector size = 10

After clear() function :

Is vector empty = 1
```

[B] Write a C++ program to implement standard library list sequence container

```
//List sequence Container
#include<iostream>
#include<conio.h>
#include<list>  //linear linked list
using namespace std;
void display(list <int> &); // display function prototype
int main()
{
   list <int> list1;      //empty list1 of zero length
   list <int> list2;      //empty list2
   list <int> list3;

   cout<<"\n\nsize of list1 = "<<list1.size();

   list1.push_front(2);
   list1.push_front(1);
   list1.push_back(3);
   list1.push_back(4);  //1234

   cout<<"\n\nList1 elements after push_front() and
push_back() :";
   display(list1);

   //Remove an element from front end
   list1.pop_front();     // same way pop_back()
   cout<<"\n\nAfter removing front element of list1 :
";
   display(list1); //234

   //insert an element(1) at the begining of list1
   list1.insert(list1.begin(),1);
   cout<<"\n\nAfter inserting an element at begining
of list1 : ";
   display(list1); //1234

   // pushing elements in list2
   list2.push_front(5);
   list2.push_front(6);
   list2.push_back(9);
   list2.push_back(8);
   list2.push_back(7);  // 65987

   cout<<"\n\nList2 elements after push_front() and
push_back() :";
   display(list2);

   //sorting list2 elements
   list2.sort();

   cout<<"\n\nList2 elements after sorting :";
   display(list2);  //56789

   //splice(insert) the elements of list2 at the end of
list1
   list1.splice(list1.end(),list2);    // similarly splice
could be at begin() also

   cout<<"\n\nList1 elements after splicing :";
   display(list1);  //123456789

   //merging list1 contents into list3
   list3.merge(list1);

   cout<<"\n\nList3 elements after merging :";
   display(list3);

   //reverse a list
   list3.reverse();

   cout<<"\n\nList3 elements after reversing :";
   display(list3);

   list3.push_back(9);
   list3.push_back(9);

   cout<<"\n\nUpdated list3 elements :";
   display(list3);

   //Removing duplicates from list3 elements
   list3.unique();
```

```
   cout<<"\n\nAfter removing duplicates from list3
elements are :";
   display(list3);

   //remove all 9's from list3
   list3.remove(9);

   cout<<"\n\nAfter removing all 9's from list3
elements :";
   display(list3);
   //using swap() function
   list2.swap(list3);
   cout<<"\n\nAfter swapping list2 : ";
   display(list2);
   cout<<"\n\nAfter swapping list3 : ";
   display(list3);
   //Using assign() function
   list3.assign(list2.begin(),list2.end());
   cout<<"\n\nAfter assiging list2 elements to lsit3 :";
   display(list3);
   getch();
   return 0;
}
void display(list <int> & v)
{
   list<int> :: iterator p;
   for(p = v.begin(); p!=v.end(); ++p)
      cout<<" "<<*p;
}
```

Output:

```
size of list1 = 0

List1 elements after push_front() and push_back() : 1 2 3 4

After removing front element of list1 :  2 3 4

After inserting an element at begining of list1 :  1 2 3 4

List2 elements after push_front() and push_back() : 6 5 9 8 7

List2 elements after sorting : 5 6 7 8 9

List1 elements after splicing : 1 2 3 4 5 6 7 8 9

List3 elements after merging : 1 2 3 4 5 6 7 8 9

List3 elements after reversing : 9 8 7 6 5 4 3 2 1

Updated list3 elements : 9 8 7 6 5 4 3 2 1 9 9

After removing duplicates from list3 elements are : 9 8 7 6 5 4 3

After removing all 9's from list3 elements : 8 7 6 5 4 3 2 1

After swapping list2 :  8 7 6 5 4 3 2 1

After swapping list3 :

After assiging list2 elements to lsit3 : 8 7 6 5 4 3 2 1
```

[C] Write a C++ program to implement standard library deque sequence container

```
//Dequeu Sequence Container
#include<iostream>
#include<conio.h>
#include<deque>
using namespace std;
void display(deque <double> &); // display function
prototype
int main()
{
   deque <double> d;

   //insert elements in d
   d.push_front(2.2);
   d.push_front(3.5);
   d.push_back(1.1);  // 3.5 2.2 1.1

   cout<<"\n\nDeque elements after insertion are as
follows :";
   display(d);

   //pop_front() to remove front element
   d.pop_front();     // d.pop_back();

   cout<<"\n\nDeque elements after pop_front() are
as follows :";
   display(d);

   //using [] subsricpt operator to modify elements
   d[1]=3.3;   // 1.1 gets overwritten by 3.3

   cout<<"\n\nDeque elements after subscript
insertion using [] are as follows :";
   display(d);

   getch();
   return 0;
}
void display(deque <double> & d1)
{
   for(int i=0;i<d1.size();i++)
   {
      cout<<" "<<d1[i];
   }
}
```

Output:

```
Deque elements after insertion are as follows : 3.5 2.2 1.1

Deque elements after pop_front() are as follows : 2.2 1.1

Deque elements after subscript insertion using [] are as follows : 2.2
```

[D] Write a C++ program to implement standard library stack adapter class

```
//Standard Library stack adapter class {can be implemented as vector,deque,list}
#include<iostream>
#include<conio.h>
#include<vector>
#include<list>
#include<deque>
#include<stack>
using namespace std;
template<class T> void pushelement(T & s);  //pushelement() function prototype
template<class T> void popelement(T & s);   //popelement() function prototype
int main()
{
   stack <int> dequestack; // stack with default underlying deque
   stack <int,vector<int> > vectorstack;   //stack with underlying int vector
   stack <int,list<int> > liststack;  //stack with underlying int list
   //push 10 elements on each of these stacks (i.e dequestack,vectorstack,liststack)
   cout<<"\n\npushing elements onto dequestack : ";
   pushelement(dequestack);
   cout<<"\n\npushing elements onto vectorstack : ";
   pushelement(vectorstack);
   cout<<"\n\npushing elements onto liststack : ";
   pushelement(liststack);
   cout<<"\n\n-----------------------------------------------";
   cout<<"\n\npopping element from dequestack : ";
   popelement(dequestack);
   cout<<"\n\npopping element from vectorstack : ";
   popelement(vectorstack);
   cout<<"\n\npopping element from liststack : ";
   popelement(liststack);
   getch();
   return 0;
}
//Function definition for pushelement()
template<class T>
void pushelement(T & s)
{
   for(int i=0;i<10;i++)
   {
      s.push(i);
      cout<<s.top()<<" ";
   }
}
//Function definition for popelement()
template<class T>
void popelement(T & s)
{
   while(!s.empty())
   {
      cout<<s.top()<<" ";
      s.pop();
   }
}
```

Output:

```
pushing elements onto dequestack : 0 1 2 3 4 5 6 7 8 9

pushing elements onto vectorstack : 0 1 2 3 4 5 6 7 8 9

pushing elements onto liststack : 0 1 2 3 4 5 6 7 8 9

-----------------------------------------------

popping element from dequestack : 9 8 7 6 5 4 3 2 1 0

popping element from vectorstack : 9 8 7 6 5 4 3 2 1 0

popping element from liststack : 9 8 7 6 5 4 3 2 1 0
```

[E] Write a C++ program to implement standard library queue adapter class template

```
//Standard Library queue adapter class
#include<iostream>
#include<conio.h>
#include<queue>
using namespace std;
int main()
{
   queue <double> q;   // double type queue "q" created (initially empty)
   //push elements onto queue
   q.push(1.1);            // 1.1  2.2  3.3
   q.push(2.2);
   q.push(3.3);
   cout<<"\n\nPopping elements from queue : ";
   while(!q.empty())
   {
      cout<<q.front()<<" ";
      q.pop();
   }
   getch();
   return 0;
}
```

Output:

```
Popping elements from queue : 1.1 2.2 3.3
```

Conclusion: All the concepts and from standard template library were understood and implemented to show different use cases of the library.

Nitesh Naik

(Subject Faculty)