Submitted by jahnvi gupta UE153044 cse- sec 1 ---- 1st year semester 2

Standard Template Library (STL)

# Introduction

The Standard Library is a fundamental part of the C++ Standard. It provides C++ programmers with a comprehensive set of efficiently implemented tools and facilities that can be used for most types of applications.The Standard Template Library, or STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

# *Containers and algorithms*

Like many class libraries, the STL includes container classes: classes whose purpose is to contain other objects. The STL includes the classes vector, list, deque, set, multiset, map, multimap, hash\_set, hash\_multiset, hash\_map, and hash\_multimap. Each of these classes is a template, and can be instantiated to contain any type of object. You can, for example, use a vector<int> in much the same way as you would use an ordinary C array, except that vector eliminates the chore of managing dynamic memory allocation by hand.The STL also includes a large collection of algorithms that manipulate the data stored in containers. You can reverse the order of elements in a vector, for example, by using the reverse algorithm.

reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7

There are two important points to notice about this call to reverse.

• First, it is a global function, not a member function.

• Second, it takes two arguments rather than one: it operates on a range of elements, rather than on a container. In this particular case the range happens to be the entire container v.

# *Iterators*

Pointers themselves are iterators, which is why it is possible to reverse the elements of a C array. Similarly, vector declares the nested types iterator and const\_iterator. In the example above, the type returned by v.begin() and v.end() is vector<int>::iterator. There are also some iterators, such as istream\_iterator and ostream\_iterator, that aren't associated with containers at all.Iterators are the mechanism that makes it possible to decouple algorithms from containers: algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container. Consider, for example, how to write an algorithm that performs linear search through a range. Find takes three arguments: two iterators that define a range, and a value to search for in that range. It examines each iterator in the range [first, last), proceeding from the beginning to the end, and stops either when it finds an iterator that points to value or when it reaches the end of the range.First and last are declared to be of type InputIterator, and InputIterator is a template parameter. That is, there isn't actually any type called InputIterator: when you call find, the compiler substitutes the actual type of the arguments for the formal type parameters InputIterator and T. If the first two arguments to find are of type int\* and the third is of type int, then it is as if you had called the following function.

***Vectors***

One of the basic classes implemented by the Standard Template Library is the vector class. A vector is, essentially, a resizable array; the vector class allows random access via the [] operator.Vectors are sequence containers representing arrays that can change in size. Just like in 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.Vectors are more powerful than arrays because the number of functions that are available for accessing and modifying vectors.

*#include <iostream>*

*#include <vector>*

*using namespace std;*

*int main()*

*{*

*vector<int> vec;*

*int i;*

*cout << "vector size = " << vec.size() << endl;*

*for(i = 0; i < 5; i++) {*

*vec.push\_back(i);*

*}*

*cout << "extended vector size = " << vec.size() << endl;*

*for(i = 0; i < 5; i++) {*

*cout << "value of vec [" << i << "] = " << vec[i] << endl;*

*}*

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

*while( v != vec.end())*

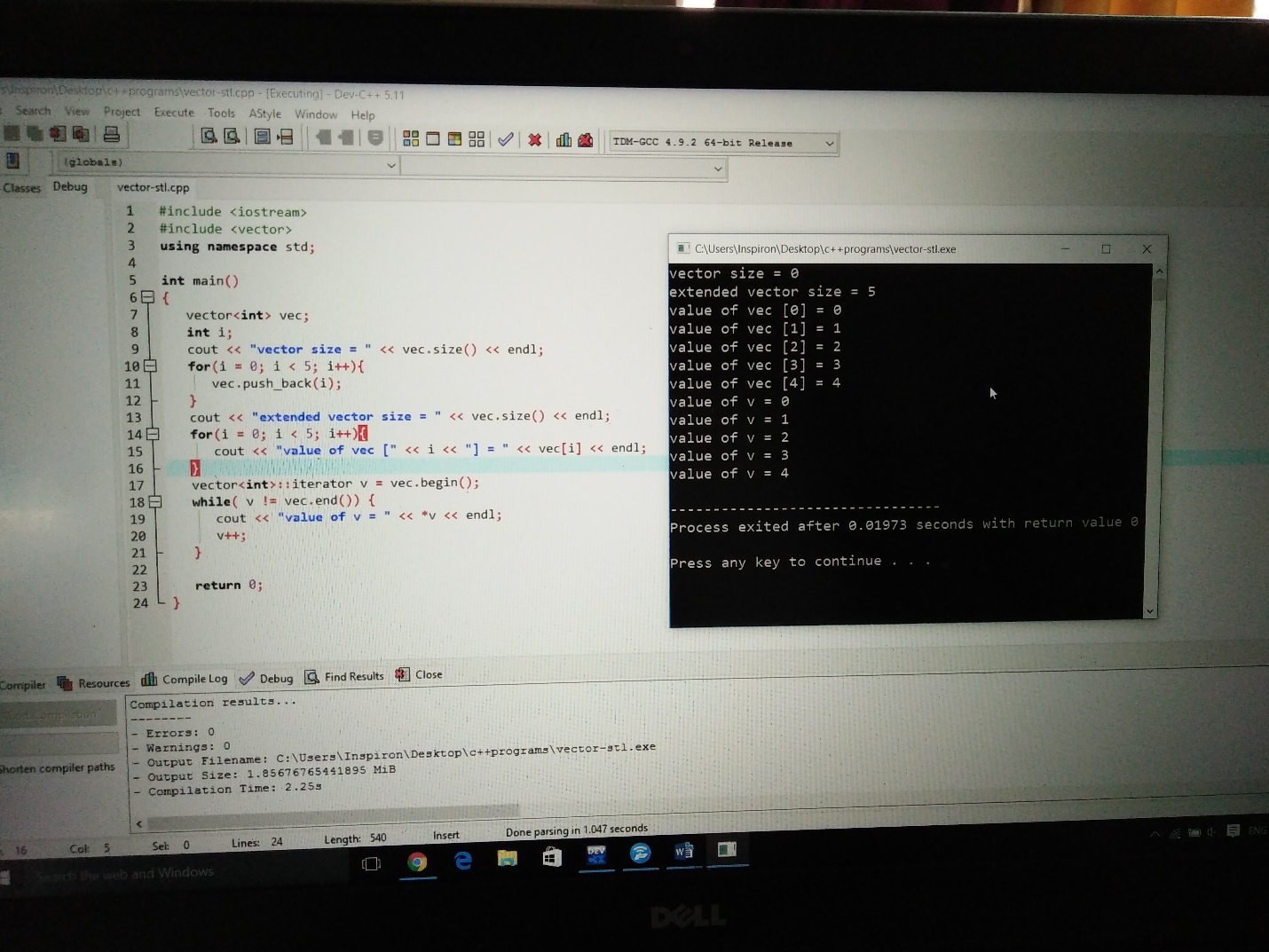
*{*

*cout << "value of v = " << \*v << endl;*

*v++;*

*}*

*return 0; }*



**LISTS**

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; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

std::list is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to[std::forward\_list](http://en.cppreference.com/w/cpp/container/forward_list) this container provides bidirectional iteration capability while being less space efficient.

Addition, removal and moving the elements within the list or across several lists does not invalidate the iterators or references. An iterator is invalidated only when the corresponding element is deleted.

### EXAMPLE 1: REVERSE

*#include <iostream>*

*#include <list>*

*int main ()*

*{*

*std::list<int> mylist;*

*for (int i=1; i<10; ++i) mylist.push\_back(i);*

*mylist.reverse();*

*std::cout << "mylist contains:";*

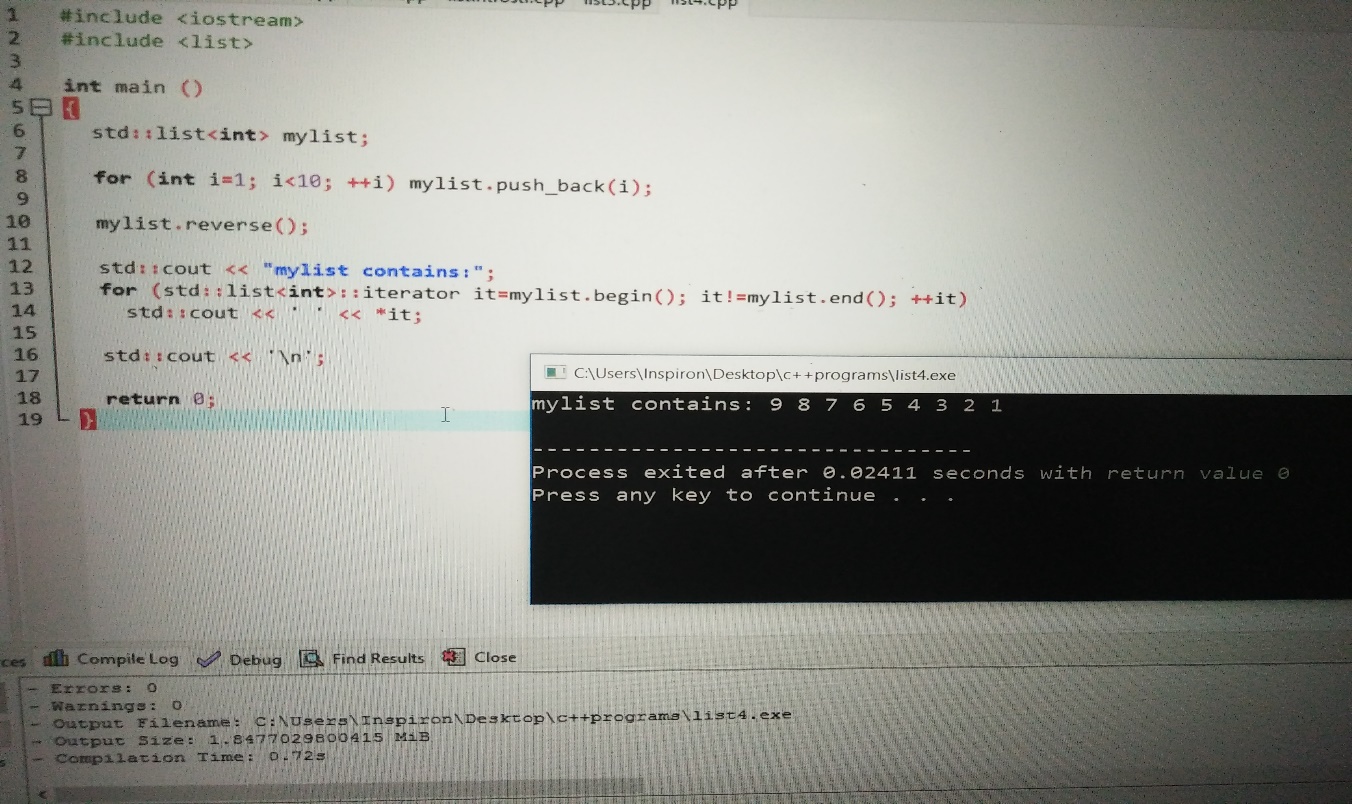
*for (std::list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)*

*std::cout << ' ' << \*it;*

*std::cout << '\n';*

*return 0;*

}



### EXAMPLE 2 : SWAP

#include <iostream>

#include <list>

int main ()

{

std::list<int> first (3,100);

std::list<int> second (5,200);

first.swap(second);

std::cout << "first contains:";

for (std::list<int>::iterator it=first.begin(); it!=first.end(); it++)

std::cout << ' ' << \*it;

std::cout << '\n';

std::cout << "second contains:";

for (std::list<int>::iterator it=second.begin(); it!=second.end(); it++)

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

