Assignment No 13

Demonstration of STL for Deque

AIM : Write C++ program using STL for Deque (Double ended queue)

Objectives:

1. To learn and understand concepts of Standard Template Library.
2. To demonstrate STL for implementation of deque operations

Outcomes:

1. Students will be able to learn and understand concepts of STL.
2. Students will be able to demonstrate various operations on deque using STL

Introduction

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components:

Component	Description
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Containers: Standard Containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate structures very commonly used in programming: dynamic arrays, queues, stacks, heaps, linked lists, trees, associative arrays.

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.

stack, queue and priority_queue are implemented as container adaptors. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the container adaptor independently of the underlying container class used.

containers are implemented via template class definitions. This allows us to define a group of elements of the required type. For example, if we need an array in which the elements are simply integers (as in the example above), we would declare it as:

vector<int> values;

The iterator is container-specific. That is, the iterator's operations depend on what type of container we are using. For that reason, the iterator class definition is inside the container class; this has a good side effect, which is that the syntax to declare an iterator suggests that the definition type belongs in the context of the container definition. Thus, we would declare an iterator for the values object as follows:

vector<int>::iterator current;

With this, we are almost ready to translate the example shown above to find the highest value in an array, using the STL tools. I will first present the implementation, and then explain the details:

```
vector<int>::iterator current = values.begin();
int high = *current++;
while (current != values.end())
{    if (*current > high)
     {    high = *current;
     }
     current++;
}
```

Here, instead of declaring a pointer to int and initialize it pointing to the first element of the array, we declare an iterator for a vector of integers (i.e., an element that will "point" to integers contained in a vector object), and we initialize it "pointing" to the first element in values. Given that values is now an object and not a standard array, we need to ask that object to give us a "pointer" to the first element (more exactly, to give us an iterator that is "pointing" to the first element).

This is done with the member-function begin() , which is provided by all the containers. The value of the iterator pointing to one past the end of the array is also provided by the container object, via the member-function end() .

Other than these two details, the rest is identical.

Following is the algorithm to find the highest value in a linked list
of integers is coded as follows using the STL tools:

```
list<int>::iterator current = values.begin();
int high = *current++;
while (current != values.end())
{     if (*current > high)
      {     high = *current;
      }
      current++;
}
```

Double ended queue : deque(usually pronounced like “deck”) is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Specific libraries may implement deques in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes undefined behavior.

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators

Conclusion: