Introduction

STL provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks. It also defines various routines that access them. Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

An Overview of the STL

Core features of STL are: container, algorithms and iterators. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the vector class defines a dynamic array, deque creates a double-ended queue, and list provides a linear list. These containers are called sequence containers because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines associative containers, which allow efficient retrieval of values based on keys. For example, a map provides access to values with unique keys. Thus a map stores a key/value pair and allows a value to be retrieved given its key.

Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a range of elements within a container.

Iterators

Iterators are objects that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators.

Iterators	Access Allowed
Random Access	Store and retrieve values. Elements may be accessed randomly.
Bidirectional	Store and retrieve values. Forward and backward moving.
Forward	Store and retrieve values. Forward moving only.
Input	Retrieve, but not store values. Forward moving only.
Output	Store, but not retrieve values. Forward moving only.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterate can be used in place of an input iterator. Iterator are handled just like pointers. You can increment and decrement them. You can apply the * operator to them. Iterators are declared using the iterator type defined by the various conatiners.

The STL also support reverse iterators. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in the reverse direction. Thus if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end. When referring to the various iterator types in template descriptions, use the following terms.

Term	Represents
BiIter	Bi-directional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIte	Random access iterator

Container Classes

As explained, containers are the STL objects that actually store data. The containers defined by the STL. Also shown are the headers necessary to use each container. The string class, which manages character strings, is also a container

Container	Description	Required Header
Bits	A set of bits	<bitset>
Deque	A double-ended queue	<deque>
List	A linear list	<list>
Map	Stores key/value pairs in which each key is associated with two or more values	<map>
Multiset	A set in which each element is not necessarily unique	<set>
Priority_queue	A priority queue	<queue>
Queue	A queue	<queue>
Set	A set in which each element is unique	<set>
Stack	A stack	<stack>
Vector	A dynamic array	<vector>

Since the names of the generic placeholder types in a template class declaration are arbitrary, the container classes declare typedef versions of these types. This makes the type name concrete. Some of the most common typedef names are shown here:

Size_type	Some type of integer
Reference	A reference to an element
Const_reference	A const reference to an element
Iterator	An iterator
Const_iterator	A const iterator
Reverse_iterator	A reverse iterator
Const_reverse_iterator	A const reverse iterator
Value_type	The type of a value stored in a container
Allocator_type	The type of the allocator
Key_type	The type of a key
Key_compare	The type of a function that compares two keys
Value_compare	The type of a function that compares two values

vector

The vector class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be adjusted at run time to accommodate changing program condition. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time. The performance of the deque Class container is superior with respect to insertions and deletions at the beginning and end of a sequence. The list Class container is superior with respect to insertions and deletions at any location within a sequence.

Vector reallocation occurs when a member function must increase the sequence contained in the vector object beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.

The template specification for vector is shown below.

```
template <
   class Type,
   class Allocator = allocator<Type>
>
class vector
```

`Type`
: The element data type to be stored in the vector

`Allocator`
: The type that represents the stored allocator object that encapsulates details about the vector's allocation and deallocation of memory. This argument is optional and the default value is allocator<Type>

Code Listing 1 - vector

```
#include <iostream>
#include <vector>

using namespace std ;

void main()
{
  int i;
  vector <int> iv;
  cout << "Vector is empty: " << iv.empty() << endl;

  for(i = 0; i < 5; i++) {
    iv.push_back(i);
    if(i == 0)
      cout << "Elements are : ";
    cout << iv.at(i) << "  ";
  }

  cout << endl;

  const int &j = iv.at(0);
  int &k = iv.at(1);
```

```
  cout << "Vector size is: " << iv.size() << endl;
  cout << "Vector maximum size is: " << iv.max_size()
       << endl;
  cout << "Vector's capacity is: " << iv.capacity() << endl;
  cout << "First element is " << j << endl;
  cout << "Second element is " << k << endl;

  vector <double> dv;

  dv.push_back(10.5);
  dv.push_back(20.6);

  cout << "Vector size is: " << dv.size() << endl;
  cout << "Vector maximum size is: " << dv.max_size() << endl;

  cout << "First element is " << dv.at(0) << endl;
  cout << "Second element is " << dv.at(1) << endl;

  vector <int> iv1;
  for(i = 10; i <= 20; i++)
    iv1.push_back(i);

  iv1.swap(iv);

  cout << "First element of v1 is " << iv1.at(0) << endl;
  cout << "Second element is " << iv1.at(1) << endl;

  // Why does the following statement result in an error?
  // cout << "The eight element is " << iv1.at(8) << endl;

  cout << "First element is " << iv1.front() << endl;
  cout << "Last element is " << iv1.back() << endl;

  vector <int> v1;
  vector <int>::iterator Iter;

  for(i = 50; i < 55; i++)
    v1.push_back(i);

  cout << "v1 = " ;
  for ( Iter = v1.begin(); Iter != v1.end(); Iter++ )
    cout << *Iter << " ";
  cout << endl;

  v1.assign(v1.begin() + 1, v1.begin() + 3);
  cout << "v1 = ";
  for (Iter = v1.begin(); Iter != v1.end(); Iter++)
    cout << *Iter << " ";
  cout << endl;

  v1.assign( 7, 4 ) ;
  cout << "v1 = ";
  for (Iter = v1.begin(); Iter != v1.end(); Iter++)
    cout << *Iter << " ";
  cout << endl;

  iv1.erase(iv1.begin(), iv1.begin() + 2);
  cout << "First element is " << iv1.at(0) << endl;
}
```

list

The STL list class is a template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing a member of some type Type.

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the class deque container is superior when random access is needed and insertions and deletions at both the beginning and the end of a sequence are at a premium.

The list member functions merge, reverse, unique, remove, and remove_if have been optimized for operation on list objects and offer a high-performance alternative to their generic counterparts.

List reallocation occurs when a member function must insert or erase elements of the list. In all such cases, only iterators or references that point at erased portions of the controlled sequence become invalid

The template specification for List is shown below:

```
template <
   class Type,
   class Allocator=allocator<Type>
>
class list

Type
    The element data type to be stored in the vector
Allocator
    The type that represents the stored allocator object that encapsulates
    details about the vector's allocation and deallocation of memory. This
    argument is optional and the default value is allocator<Type>
```

Code Listing 2 - list

```
# include <iostream>
# include <list>

using namespace std;

void main()
{
  int i;
  list <int> lst1;

  for(i = 0; i < 5; i++)
    lst1.push_back(i);

  list <int>::iterator c1_Iter;
  list <int>::const_iterator c1_cIter;
  list <int>::reverse_iterator c1_rIter;
  c1_Iter = lst1.begin();
  cout << "\nThe first element of list 1 is " << *c1_Iter << endl;

  *c1_Iter = 20;
  c1_Iter = lst1.begin();
```

```
  cout << "The first element of list 1 is now " << *c1_Iter << endl;

  // The following line would be an error because iterator is const
  // *c1_cIter = 200;

  c1_rIter = lst1.rbegin();
  cout << "The last element of list 1 is " << *c1_rIter << endl;
  cout << "List 1 data is:";
  for (c1_Iter = lst1.begin(); c1_Iter != lst1.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  // rbegin can be used to start an iteration through a list in reverse order
  cout << "List 1 data in reverse order is:";
  for (c1_rIter = lst1.rbegin(); c1_rIter != lst1.rend(); c1_rIter++)
    cout << " " << *c1_rIter;
  cout << endl;

  cout << "List 1 Max size is " << lst1.max_size() << endl;

  list <int> lst2;
  for(i = 0; i < 10; i++)
    lst2.push_back(i);

  cout << "Elements of List 2 are ";
  for (c1_Iter = lst2.begin(); c1_Iter != lst2.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  lst1.swap(lst2);

  cout << "Elements of List 1 after swapping are ";
  for (c1_Iter = lst1.begin(); c1_Iter != lst1.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  list <int>::iterator Iter;
  Iter = lst2.begin( );
  Iter++;

  lst2.erase(++Iter, lst2.end());

  cout << "Elements of List 2 after erasing are ";
  for (c1_Iter = lst2.begin(); c1_Iter != lst2.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  lst1.pop_front();
  lst1.pop_back();
  lst1.remove(4);

  cout << "Elements of List 1 after popping are ";
  for (c1_Iter = lst1.begin(); c1_Iter != lst1.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;
}
```

Code Listing 3 – List (sorting)

```
# include <iostream>
# include <list>
# include <string>

using namespace std;

void main()
{
  int i;
  list <int> lst;
  for(i = 0; i < 5; i++)
```

```
    lst.push_back(rand());

  list <int>::iterator c1_Iter;
  c1_Iter = lst.begin();
  lst.insert(++c1_Iter, 111);
  lst.push_front(222);
  cout << "Before sorting List:";
  for (c1_Iter = lst.begin(); c1_Iter != lst.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  lst.sort(); // same as lst.sort(less<int>());
  cout << "After sorting List:";
  for (c1_Iter = lst.begin(); c1_Iter != lst.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  lst.sort(greater<int>());
  cout << "After sorting with 'greater than' operation, List:";
  for (c1_Iter = lst.begin(); c1_Iter != lst.end(); c1_Iter++)
    cout << " " << *c1_Iter;
  cout << endl;

  list <string> lststr;

  lststr.push_back("Juno");
  lststr.push_back("Genesis");
  lststr.push_front("DeShaw");
  lststr.push_back("Birla");

  list <string>::iterator c1_Iter1;

  cout << "Before sorting List:";
  for (c1_Iter1 = lststr.begin(); c1_Iter1 != lststr.end(); c1_Iter1++)
    cout << " " << *c1_Iter1;
  cout << endl;

  lststr.sort(); // lststr.sort(less<string>());
  cout << "After sorting List:";
  for (c1_Iter1 = lststr.begin(); c1_Iter1 != lststr.end(); c1_Iter1++)
    cout << " " << *c1_Iter1;
  cout << endl;

  lststr.sort(greater<string>());
  cout << "After sorting with 'greater than' operation, List:";
  for (c1_Iter1 = lststr.begin(); c1_Iter1 != lststr.end(); c1_Iter1++)
    cout << " " << *c1_Iter1;
  cout << endl;

  lststr.reverse();
  cout << "After reversing List:";
  for (c1_Iter1 = lststr.begin(); c1_Iter1 != lststr.end(); c1_Iter1++)
    cout << " " << *c1_Iter1;
  cout << endl;
}
```

stack

A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type. The stack class is used when it is important to be clear that only stack operations are being performed on the container.
The template specification for stack is shown here:

```
template <
   class Type,
   class Container=deque<Type>
>
class stack

Type
    The element data type to be stored in the stack
Container
    The type of the underlying container used to implement the stack. The
    default value is the class deque<Type>
```

Code Listing 4 - Stack

```
#include <iostream>
#include <stack>
#include <string>

using namespace std ;

void main()
{
  string data[5] = {"Juno", "ADP", "Genesis", "Microsoft", "Satyam"};
  stack <string> stk;
  int i;

  cout << "stack size is " << stk.size() << endl;

  for(i = 0; i < 5; i++)
    stk.push(data[i]);

  cout << "Top element: " << stk.top() << endl;
  stk.pop();
  cout << "stack size is " << stk.size() << endl;
  cout << "Top element afer pop: " << stk.top() << endl;

  stack <int> stk1;
  cout << "stack size is " << stk1.size() << endl;

  for(i = 0; i < 5; i++) {
    int j = rand();
    cout << j << " ";
    stk1.push(j);
  }

  cout << "\nTop element: " << stk1.top() << endl;
  stk1.pop();
  cout << "Top element afer pop: " << stk1.top() << endl;
}
```

deque

The STL sequence container deque arranges elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container. However, unlike a vector, the deque class also supports efficient insertion and deletion at the front of the container.

The template specification for deque is shown below:

```
template <
   class Type,
   class Allocator=allocator<Type>
>
class deque

Type
    The element data type to be stored in the vector
Allocator
    The type that represents the stored allocator object that encapsulates
    details about the vector's allocation and deallocation of memory. This
    argument is optional and the default value is allocator<Type>
```

Code Listing 5 - deque

```
#include <iostream>
#include <deque>
#include <string>

using namespace std;

void printcontents (deque <string> dq)
{
  deque <string>::iterator pdeque;
  cout <<"\nData is: ";

  for(pdeque = dq.begin(); pdeque != dq.end(); pdeque++)
    cout << *pdeque << " ";
}

void main()
{
  int i;
  string data[5] = {"Juno", "ADP", "Genesis"};
  deque <string> dq;

  for(i = 0; i < 5; i++)
    dq.push_back(data[i]);

  printcontents (dq);
  cout << "\nData at position 1 is: " << dq.at(1);
  dq.pop_back();
  dq.push_front("Infosys");
  printcontents (dq);

  deque <string>::iterator Iter;

  Iter = dq.begin();
  dq.insert(++Iter, 2, "Barion");
  printcontents (dq);

  deque <string> :: reverse_iterator rIter;

  cout << "\nThe reversed deque is: ";
  for(rIter = dq.rbegin(); rIter != dq.rend(); rIter++)
    cout << *rIter << " ";
```

```
  cout << endl;
}
```

map

The STL map class is used for the storage and retrieval of data from a collection in which the each element is a pair that has both a data value and a sort key. The value of the key is unique and is used to order the data automatically. The value of an element in a map, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The map class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and store that person's telephone number as its value. A map can hold only unique keys. Duplicate keys are not allowed. To create a map that allows non unique keys, use multimap.

The template specification for map is shown below:

```
template <class Key,
   class Type,
   class Traits = less<Key>,
   class Allocator=allocator<pair <const Key, Type>>
>
class map


Key
    The key data type to be stored in the map.
Type
    The element data type to be stored in the map.
Traits
    The type that provides a function object that can compare two element
    values as sort keys to determine their relative order in the map. This
    argument is optional and the binary predicate less<Key> is the default
    value.
Allocator
    The type that represents the stored allocator object that encapsulates
    details about the map's allocation and deallocation of memory. This
    argument is optional and the default value is allocator<pair <const Key,
    Type>>
```

Code Listing 6 - map

```
#include <iostream>
#include <map>

using namespace std;

int main( )
{
  map <int, int>::iterator m1_pIter, m2_pIter;

  map <int, int> m1, m2;
  typedef pair <int, int> Int_Pair;

  m1.insert (Int_Pair(1, 10));
  m1.insert (Int_Pair(2, 20));
```

```
  m1.insert (Int_Pair(3, 30));
  m1.insert (Int_Pair(4, 40));

  cout << "The original key values of m1 =";
  for (m1_pIter = m1.begin(); m1_pIter != m1.end(); m1_pIter++)
    cout << " " << m1_pIter -> first;
  cout << endl;

  cout << "The original mapped values of m1 =";
  for (m1_pIter = m1.begin(); m1_pIter != m1.end(); m1_pIter++)
    cout << " " << m1_pIter->second;
  cout << endl;

  pair< map<int,int> :: iterator, bool > pr;
  pr = m1.insert (Int_Pair(1, 10));

  if(pr.second == true)
  {
    cout << "The element 10 was inserted in m1 successfully." << endl;
  }
  else
  {
    cout << "The element 10 already exists in m1 with a key value of " <<
      (pr.first)->first << endl;
  }

  // The int version of insert
  m1.insert(--m1.end(), Int_Pair(5, 50));

  cout << "After the insertions, the key values of m1 =";
  for (m1_pIter = m1.begin(); m1_pIter != m1.end(); m1_pIter++)
    cout << " " << m1_pIter->first;
  cout << "," << endl;

  cout << "and the mapped values of m1 =";
  for (m1_pIter = m1.begin(); m1_pIter != m1.end(); m1_pIter++)
    cout << " " << m1_pIter->second;
  cout << endl;

  m2.insert (Int_Pair(10, 100));

  // The templatized version inserting a range
  m2.insert(++m1.begin(), --m1.end());

  cout << "After the insertions, the key values of m2 =";
  for (m2_pIter = m2.begin(); m2_pIter != m2.end(); m2_pIter++)
    cout << " " << m2_pIter->first;
  cout << "," << endl;

  cout << "and the mapped values of m2 =";
  for (m2_pIter = m2.begin(); m2_pIter != m2.end(); m2_pIter++)
    cout << " " << m2_pIter->second;
  cout << endl;
}
```

Problems

1. Use the appropriate template class to store countries and capitals. Develop a quiz to test the same.

2. Write a program to accept a set of names and store the same in descending order.