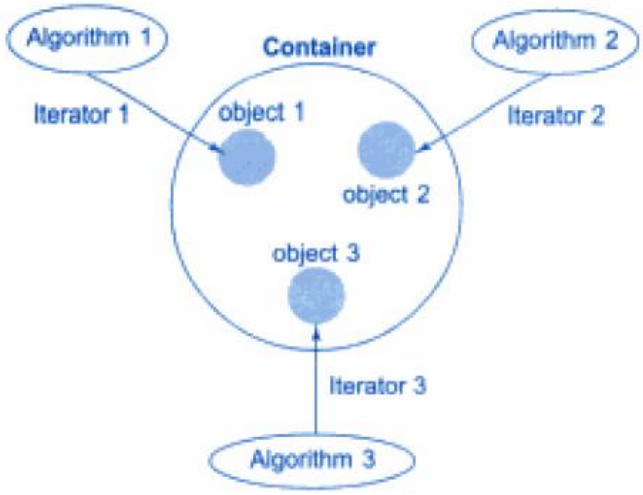
Standard Template Library (STL)

Introduction to STL : Container classes are building blocks used to create object oriented programs and they make the internals of a program much easier to construct. A container



class describes an object that holds other objects. Container classes are so important that they were considered fundamental to early object oriented languages. The C++ approach to containers is based on templates. The containers in the Standard C++ library represent a broad range of data structures designed to work well with the standard algorithms and to meet common software development needs.

The standard template library (STL) is collection of well structured generic C++ classes (templates) and functions.

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, algorithms and iterators :

The standard template library (STL) contains − Containers − Iterators − Algorithms

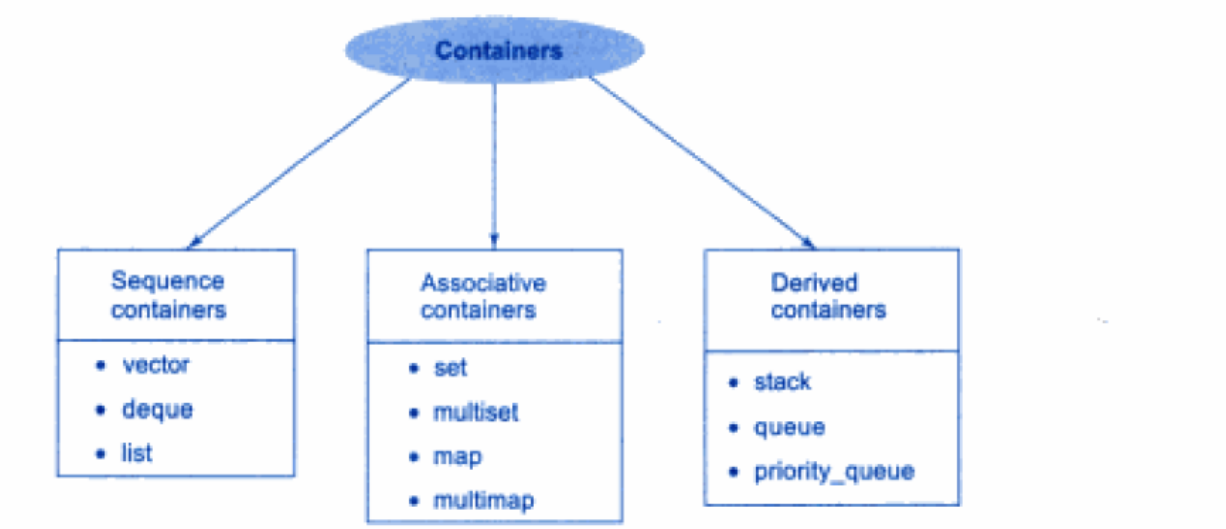
1. A container is a way stored data is organized in memory, for example an array of elements.
2. The container is a collection of objects of different types. These objects store the data. The container can be implemented using template classes.
3. Algorithms in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
4. Iterators are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array.
5. Various types of containers are : Sequence container, Associative container, Derived

container

Containers in Software : A container is usually instantiated as an object of container class. A container class object encapsulates inside it a mechanism for containing other objects. It also provides the necessary behaviour for adding, removing and accessing the objects it contains. A container class gives the opportunity of reuse in different programs: – this frees the programmer from having to recreate complex data structures in every program to manage complex data structure

Containers- Sequence container and associative containers
Some Containers Types:

Sequence container:
Associative containers:



Sequence Containers:

- It consists of all the classes who represent the sequence or linear list. For example vector class defines array.
- vector, list and deque; They store elements in client-visible order
- A sequential container stores elements in a sequence.
- In other words each element (except for the first and last one) is preceded by one specific element and followed by another,<vector>, <lits> and <deque> are sequential containers.
- In an ordinary C++ array the size is fixed and can not change during run-time, it is also tedious to insert or delete elements.
- Advantage: quick random access
- <vector> is an expandable array that can shrink or grow in size, but still has the disadvantage of inserting or deleting elements in the middle
- <list> is a double linked list (each element has two pointers to its successor and predecessor), it is quick to insert or delete elements but has slow random access.
- <deque> is a double-ended queue, that means one can insert and delete objects from both ends, it is a kind of combination between a stack (LIFO) and a queue (FIFO) and constitutes a compromise between a <vector> and a <list>

Associative Containers :

- An associative container is non sequential but uses a key to access elements.
- The keys, typically a number or a string, are used by the container to arrange the stored objects in a specific order, for example in a dictionary the entries are ordered alphabetically.
- A <set>stores a number of items which contain keys.
- The keys are the attributes used to order the items, for example a set might store objects of the class Books which are ordered alphabetically using their title.
- A <map>stores pairs of objects: a key object and an associated value object.
- A <map>is somehow similar to an array except instead of accessing its elements with index numbers, you access them with indices of an arbitrary type.
- <set> and <map> only allow one key of each value, whereas <multiset> and <multimap> and allow multiple identical key values
- It allow efficient retrieval of value based on keys
- map, multimap, set and multiset are examples.
- Containers Adapters: queue, priorityqueue and stack

Derived container:

- The derived containers are created from the sequence containers.
- The derived containers are stack, queue and priority queue.
- These are also known as container adaptors.
- 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 include functions like push and pop values.

List:

- A standard doubly linked container
- supports constant¬time insertion and deletion of elements at any point of the list
- Most list operation are identical to those of a vector
- However, list do not provide random access to elements
- Insert(), erase() run in constant time which makes lists suitable for applications that perform many insertion and deletion

The use of containers:

- They give us control over collections of objects, especially dynamic objects
- Gives a simple mechanism for creating, accessing and destroying without explicitly programming algorithms to do these operation
- Container "Iterator" methods allow us to iterate throw the container

A simple Example : Here's an example using the set class template. This container, modelled after a traditional mathematical set, does not accept duplicate values. The following set was created to work with Int Set

```
#include <iostream>
#include <iterator>
#include <set>
using namespace std;
int main(void)
{
 set<int> intset;
 for(int i = 0; i < 250; i++)
```

```
    for(int  j = 0; j < 24; j++)
       intset.insert(j);
  cout <<"Set Size "<<intset.size()<<endl;
  cout <<"Contents"<<endl;
  copy(intset.begin(),intset.end(),
      ostream_iterator<int>(cout,  ", "));
cout<<"\n";
}
```

Set :

- The insert( ) member does all the work:
- it attempts to insert an element and ignores it if it's already there.
- Often the only activities involved in using a set are simply insertion and testing to see whether it contains an element.
- You can also form a union, an intersection, or a difference of sets and test to see if one set is a subset of another.
- In this example, the values 0¬24 are inserted into the set 250 times, but only the 25 unique instances are accepted.

The Copy Algorithm:

- To print out the contents of the set we use, a special algorithm called copy In conjunction with the ostream_iterator
- We use the copy algorithm to remove the loop we could use to access the elements of the set
- and the begin() and end() methods of the set to gather the extents of the set to copy.
- Following is the simple example of container class for list container which demonstrates sequence container.

```
#include<iostream>
#include<list>
#include<numeric>
using namespace std;
void print(list<double> &lst)
{
list<double>::iterator key; //traverse iterator
for(key=lst.begin();key!=lst.end();++key)
cout<<*key<<"\t";
cout<<endl;
}
int main()
{
double array[4]={30,20,10,40};
list<double> obj;
for(int i=0;i<4;++i)
// inserting elements in the list
obj.push_front(array[i]);
print(obj);
// sorting of the list
obj.sort();
cout<<"the sorted list is:"<<endl;
print(obj);
```

```
cout<<"sum is"<<accumulate(obj.begin(),obj.end(),0.0)<<endl;
return 0;
}
```

OUTPUT:

```
40,10,20,30
The sorted list is:
10,20,30,40
Sum is 100
```

- In above program the list container is used. An array consists of 4 double values. These double values will be pushed into the list container. The print function uses an iterator to print each element of list. The iterator acts like a pointer. The begin() and end () are the member functions of the list container.The begin () represents the starting and end represents the ending location of the container. The sort () function is useful for sorting the elements. This member function basically represents the stable sorting algorithm. The numeric package is include for supporting the accumulate () function.It uses 0.0 as the initial value and then computing the sum of all the elements starting from the location obj.begin () to obj.end ().

Various member functions of container class are enlisted in following table.

	Member	Meaning
1.	CAN::CAN():	CAN holds the item it is a copy constructor
2.	CAN::CAN(C):	Copy constructor for referring beginning location
3.	c.begin() :	For referring beginning location
4.	c.end() :	Ending location of the list
5.	c.rbegin() :	beginning of reverse iterator
6.	c.size() :	number of elements in CAN

Iterator:

- The STL implements five different types of iterators. These are input iterators (that can only be used to read a sequence of values), output iterators (that can only be used to write a sequence of values), forward iterators (that can be read, written to, and move forward), bidirectional iterators (that are like forward iterators, but can also move backwards) and random access iterators (that can move freely any number of steps in one operation).
- It is possible to have bidirectional iterators act like random access iterators, as moving forward ten steps could be done by simply moving forward a step at a time a total of ten times. However, having distinct random access iterators offers efficiency advantages. For example, a vector would have a random access iterator, but a list only a bidirectional iterator.
- Iterators are the major feature that allow the generality of the STL. For example, an algorithm to reverse a sequence can be implemented using bidirectional iterators, and then the same implementation can be used on lists, vectors and deques. User-created containers only have to provide an iterator that implements one of the five standard iterator interfaces, and all the algorithms provided in the STL can be used on the container.
- This generality also comes at a price at times. For example, performing a search on an associative container such as a map or set can be much slower using iterators than by calling member functions offered by the container itself. This is because an associative

container's methods can take advantage of knowledge of the internal structure, which is opaque to algorithms using iterators.

- once we have the values to copy we need to copy it to some place which in this case is the ostream_iterator class template declared in the <iterator> header.
- An ostream_iterator is an Output Iterator that performs formatted output of objects of type T to a particular ostream.
- Output stream iterators overload their copy assignment operators to write to their stream.
- This particular instance of ostream_iterator is attached to the output stream cout
- It is possible to attach it to a file and get the results copied to a file
- Every copy assigns an integer from the set to cout through this iterator, the iterator writes the integer to cout and also automatically writes an instance of the separator string found in its second argument, which in this case contains a ,
- It is just as easy to write to a file by providing an output file stream, instead of cout
- Iterator is a generation of pointer
- It is an object belonging to a class with the prefix * defined on it So that, if p in an iterator over a container, *p is an object in the container.
- You can think of iterator as pointing to a current object at any time
- An Iterator is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented. With many classes (particularly lists and the associative classes), iterators are the primary way elements of these classes are accessed.
- An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

1. Operator* -- Dereferencing the iterator returns the element that the iterator is currently pointing at.
2. Operator++ -- Moves the iterator to the next element in the container. Most iterators also provide Operator-- to move to the previous element.
3. Operator== and Operator!= -- Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, deference the iterators first, and then use a comparison operator.
4. Operator= -- Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is point at, deference the iterator first, then use the assign operator.

Each container includes basic member functions for use with Operator=:

- begin() returns an iterator representing the beginning of the elements in the container.
- end() returns an iterator representing the element just past the end of the elements.
- It might seem weird that end() doesn't point to the last element in the list, but this is done primarily to make looping easy: iterating over the elements can continue until the iterator reaches end(), and then you know you're done.
- All containers provide (at least) two types of iterators:

container::iterator provides a read/write iterator
container::const_iterator provides a read-only iterator

```
Eg: vector<float> L;
for(vector<float>::iterator p=L.begin(); p<L.end(); p++)
    cout << *p <<endl;
```

More Complex example: countw.cpp

```
using namespace std;
void CountUniqueWords(const char* fileName)
{
ifstream source(fileName);
if(!source)
  {  cerr<<"error opening file\n";
     exit(EXIT_FAILURE);
  }
string word;
set<string>  words;
while(source  >> word)
      words.insert(word);
cout << "Number of unique words:"<< words.size() << endl;
copy(words.begin(), words.end(),ostream_iterator<string>(cout, "\n"));
}
int main(int argc, char* argv[])
{
if(argc > 1) CountUniqueWords(argv[1]);
else
  cerr<<"Usage "<<argv[0]<<" file name"<<endl;
}
```

Different Iterators:

- An iterator is an abstraction for genericity.
- It works with different types of containers without knowing the underlying structure of those containers.
- Most containers support iterators, so you can say <ContainerType>::iterator <ContainerType>::const iterator to produce the iterator types for a container.
- Every container has a begin( ) member function that produces an iterator indicating the beginning of the elements in the container, and an end( ) member function that produces an iterator which is the past-the-end marker of the container.
- If the container is const, begin( ) and end( ) produce const_iterators, which disallow changing the elements pointed to (because the appropriate operators are const).
- All iterators can advance within their sequence (via operator++) and allow == and != comparisons.
- Thus, to move an iterator it forward without running it off the end, you say something like:

```
while(it  != pastEnd)
{
        //Do something
         ++it;
}
```

- where pastEnd is the past-the-end marker produced by the container's end( ) member function.
- An iterator can be used to produce the container element that it is currently selecting via the dereferencing operator (operator*).
- This can take two forms. If it is an iterator traversing a container, and f() is a member function of the type of objects held in the container, you can say either:

```
(*it).f();    or    it¬>f();
```

Reverse Iterators:

- A container may also be reversible, which means that it can produce iterators that move backward from the end, as well as iterators that move forward from the beginning.
- All standard containers support such bidirectional iteration.
- A reversible container has the member functions:

− rbegin( ) (to produce a reverse iterator selecting the end)
− rend( ) (to produce a reverse iterator indicating "one past the beginning").

- If the container is const, rbegin( ) and rend( ) will produce const_reverse_iterators.

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main(int argc, char **argv)
{
if(argc>1)
{
 ifstream in(argv[1]);
 string line;
 vector<string> lines;
 while(getline(in, line))
   lines.push_back(line);
vector<string>::reverse_iterator r=lines.rbegin();
 vector<string>::reverse_iterator end=lines.rend();
 for(r; r!=end; r++)
  cout << *r << endl;
  }
}
```

Algorithms:

- A large number of algorithms to perform activities such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container that provides an interface by iterators).
- Searching algorithms like binary_search and lower_bound use binary search and like sorting algorithms require that the type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering.
- Apart from these, algorithms are provided for making heap from a range of elements, generating lexicographically ordered permutations of a range of elements, merge sorted ranges and perform union, intersection, difference of sorted ranges.

Sorting algorithms:

- These algorithms contain the functionalities related to sorting of the list.
- Mutating sequence algorithm: These algorithms modify the contents of the container.
- For example copy() operation will modify the contents of the container.

- Non mutating sequence algorithms: Non-mutating sequence operations perform operations that don't change the elements in a sequence. Example: for_each(), search(), find(), and count().
- Numerical algorithms:These algorithms are useful for performing some computations.
- For instance sum of all the elements can be obtained by the function accumulate().

A simple program of sorting the elements of an array using sorting algorithm.

```
#include<iostream>
#include<algorithm>
#define size 10
using namespace std;
int main()
{
int n,item;
int array[size],i;
cout<<"how many elements you want to enter\n";
cin>>n;
int *limit=array=n;// setting the range for sorting
cout<<"enter the numbers:";
for(i=0;i<n;i++)
{
cin>>item;
array[i]=item;
}
//calling the function from algorithm
sort(array,limit);
//displaying the sorted list of elements
cout<<"\n the sorted list is:"<<endl;
for(i=0;i<n;i++)
cout<<array[i]<<"\t";
cout<<endl;
return 0;
}
```

OUTPUT

How many elements you want to enter
Enter the numbers:
4 3 1 2 7 6
The sorted list is:

1 2 3 4 6 7

In the above program the keyword<algorithm> is used and sort is a function that is supported by sorting algorithm. By this function the quicksort is performed over the range of the array from first element to the last element of the array.

We will summarize various algorithms and supporting functionalities in following tables

Sorting algorithm

Function	Description
• Sort()	using quick sort the elements are sorted
• Stable_sort()	stable sorting method is used
• Merge()	merges the elements
• Sort_heap()	performs sorting on heap
• Min()	it find the min element
• Max()	it finds the max element
• Binary_search()	it performs the binary search on the sorted elements

Mutating Algorithms

Function	Description
• Copy()	copies the sequence of elements
• Copy_backward()	copies the list in backward direction
• Reverse()	this function is used to reverse the given sequence
• Unique() remove	it finds the adjacent duplicate elements and them

Nonmutating Algorithm

Function	Description
• Find()	it will find the position of desired element
• Count()	this function counts the number of elements in the given sequence
• Equal() are	it checks whether the two sequences are equal or not. If two sequences matching then it returns true.
• Search()	This operation is used for searching the desired element from the given sequence.

Applications Of Container Class

Vector :

- The vector container resembles a C++ array in that it holds objects of the same type, and that each of these objects can be accessed individually.
- Vector containers are optimized to provide fast access to their elements by an index.
- The vector container is defined as a template class, meaning that it can be customized to hold objects of any type.
- A vector is great for accessing objects in sequence, but it does not provide an efficient method of accessing objects randomly, outside of sequence.
- For random access, use some type of a map, multimap, set, or multiset container.
- There are 2 arguments that go with a vector container.
- One is class T, and as with all templates, this represent the type of objects being stored in the elements of the vector.
- The other argument is class A, the allocator class.
- Allocators are memory managers responsible for memory allocation and deallocation of elements for containers.
- Here's how we define vectors:
  vector<int> VectorOfInts;
  vector<floats> VectorOfFloats;

- The vector class includes a constructor that accepts the number of elements in the vector as a parameter.

  Here are some examples:

```
#include <iostream.h>
#include <vector>
using namespace std;
//Define a vector of integers. The template name is "vector". The type of object
//it contains "int". The fully specified container data type is "vector<int>".

void main()
{
    vector<int> vec_one;
    int a = 10;
    int b = 20;
    vec_one.push_back(a);  //Add items at end of vector
    vec_one.push_back(15);
    vec_one.push_back(b);
    // vec_one now contains three int values: 2, 10, -5
    unsigned int indx;
    //Display int objects stored in the vector
    for(indx = 0; indx < vec_one.size(); indx++)
    {
       cout << vec_one[indx] << " ";
    }
     cout << endl;
   } // close main()
```

Output: 10 15 20

Member Functions of vector container:

Function	Description
maxsize() -	maximum number of elements allowed in a vector.
capacity() -	tells how many elements a vector can hold.
size() -	returns the number of items currently stored in the vector.
empty() -	returns true if the number of elements is 0.
push_back() -	returns an iterator that references a position just past the end of the vector. For this method to work, the object class must define a copy constructor,
begin() -	returns an iterator that references the beginning of the vector.
end() -	returns an iterator that references a position just past the end
erase() -	remove specified elements from a vector.
clear() -	erases (removes) all elements from a vector.

Iterating through a vector:

```
#include <iostream>
```

```
#include <vector>
int main()
{
   using namespace std;
   vector<int> vect;
   for (int nCount=0; nCount < 6; nCount++)
      vect.push_back(nCount);
   vector<int>::const_iterator it; // declare an read-only iterator
   it = vect.begin(); // assign it to the start of the vector
   while (it != vect.end()) // while it hasn't reach the end
      {
      cout << *it << " "; // print the value of the element it points to
      ++it; // and iterate to the next element
      }
   cout << endl;
}
```

This prints the following:

0 1 2 3 4 5

Iterating through a list:
Now let's do the same thing with a list:

```
#include <iostream>
#include <list>
int main()
{
   using namespace std;
   list<int> li;
   for (int nCount=0; nCount < 6; nCount++)
      li.push_back(nCount);
   list<int>::const_iterator it; // declare an iterator
   it = li.begin(); // assign it to the start of the list
   while (it != li.end()) // while it hasn't reach the end
   {
      cout << *it << " "; // print the value of the element it points to
      ++it; // and iterate to the next element
   }
   cout << endl;
}
```

This prints:

0 1 2 3 4 5

Note the code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

Iterating through a set: In the following example, we're going to create a set from 6 numbers and use an iterator to print the values in the set:

```
#include <iostream>
```

```
#include <set>
int main()
{
    using namespace std;
    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);
    set<int>::const_iterator it; // declare an iterator
    it = myset.begin(); // assign it to the start of the set
    while (it != myset.end()) // while it hasn't reach the end
    {
        cout << *it << " "; // print the value of the element it points to
        ++it; // and iterate to the next element
    }

    cout << endl;
}
```

This program produces the following result:

```
-6 -4 1 2 7 8
```

Note that although populating the set differs from the way we populate the vector and list, the code used to iterate through the elements of the set was essentially identical.

Iterating through a map: This one is a little trickier. Maps and multimaps take pairs of elements (defined as an std::pair). We use the make_pair() helper function to easily create pairs. std::pair allows access to the elements of the pair via the first() and second() member functions. In our map, we use first() as the key, and second() as the value.

```
#include <iostream>
#include <map>
#include <string>
int main()
{
    using namespace std;
    map<int, string> mymap;
    mymap.insert(make_pair(4, "apple"));
    mymap.insert(make_pair(2, "orange"));
    mymap.insert(make_pair(1, "banana"));
    mymap.insert(make_pair(3, "grapes"));
    mymap.insert(make_pair(6, "mango"));
    mymap.insert(make_pair(5, "peach"));
    map<int, string>::const_iterator it; // declare an iterator
    it = mymap.begin(); // assign it to the start of the vector
    while (it != mymap.end()) // while it hasn't reach the end
    {
        cout << it->first << "=" << it->second << " "; // print the value of the element it points to
        ++it; // and iterate to the next element
```

```
    }
    cout << endl;
}
```
This program produces the result:

1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango

Notice here how easy iterators make it to step through each of the elements of the container. You don't have to care at all how map stores its data!

Iterators provide an easy way to step through the elements of a container class without having to understand how the container class is implemented. When combined with STL's algorithms and the member functions of the container classes, iterators become even more powerful.
One point worth noting: Iterators must be implemented on a per-class basis, because the iterator does need to know how a class is implemented. Thus iterators are always tied to specific container classes.
In addition to container classes and iterators, STL also provides a number of generic algorithms for working with the elements of the container classes. These allow you to do things like search, sort, insert, reorder, remove, and copy elements of the container class.

Note that algorithms are implemented as global functions that operate using iterators. This means that each algorithm only needs to be implemented once, and it will generally automatically work for all containers that provides a set of iterators (including your custom container classes). While this is very powerful and can lead to the ability to write complex code very quickly, it's also got a dark side: some combination of algorithms and container types may not work, may cause infinite loops, or may work but be extremely poor performing. So use these at your risk.

Stack Container Class:

- A stack is an adaptor that provides a restricted subset of Container functionality:
- it provides insertion, removal, and inspection of the element at the top of the stack.
- Stack is a "last in first out" (LIFO) data structure: the element at the top of a stack is the one that was most recently added.Stack is a container adaptor, meaning that it is implemented on top of some underlying container type.
- By default that underlying type is deque, but a different type may be selected explicitly.

Following is a list of operations that are supported by the stack derived container class.

Function	Description
• push(item)	this function helps to push an item onto the stack
• pop()	this function pops the top most element from the stack
• size()	this function return the size of the stack
• top()	this function return value which is at the top of the stack

For using these functions in the program we must include the header file <stack> at the top. Following is a simple C++ program in which the derived container class stack is used. Various operations are also performed on this stack using the inbuilt functions of container class library.

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

```
using namespace std;
int main()
{
stack<int>  s;
int     item;
char  ans;
int    choice;
do
{
cout<<"\n main menu";
cout<<"\n 1.push";
cout<<"\n 2.pop";
cout<<"\n 3.display";
cout<<"\n enter your choice\n";
cin>>choice;
switch(choice)
{
case 1:cout<<"\n enter the element to be pushed";
          cin>>item;
          s.push(item);
          cout<<"\n item is pushed";
          break;
case 2:if(!s.empty())
          {
          s.pop();
          cout<<"\npopped an item";
          }
          else
          cout<<"\n stack is empty";
          break;
case 3: if(!s.empty())
          {
          cout<<"top element of the stack is <<s.top();
          }
          else
          cout<<"\n the stack is empty";
          break;
     }
      cout<<"\n do you want to continue?";
     cin>>ans;
  } while(ans=='y');
    return 0;
}
```

Algorithms- basic searching and sorting algorithms: STL provides quite a few algorithms -- we will only touch on some of the more common and easy to use ones here. To use any of the STL algorithms, simply include the algorithm header file.

Sorted Sequences: Once we have collected some data, we often want to sort it. Once the sequence is sorted, our options for manipulating the data in a convenient manner increase significantly. To sort a sequence, we need a way of comparing elements. This is done using a binary predicate. The default comparison is less. The sort() algorithms require random-access

iterators . That is, they work best for vectors and similar containers: template void sort(Ran first, Ran last) ; template void sort(Ran first, Ran last, Cmp cmp) ; template void stable_sort(Ran first, Ran last) ; template void stable_sort(Ran first, Ran last, Cmp cmp) ; The standard list does not provide random-access iterators, so lists should be sorted using the specific list operations

Once a sequence is sorted, however, we can use a binary search to determine whether a value is in a sequence:
template bool binary_search(For first, For last, const T& val) ;
template bool binary_search(For first, For last, const T& value, Cmp cmp) ;

For example:
```
void f(list&  c) {
 if(binary_search(c.begin()  ,c.end() ,7)) {
 / / is 7 in c? / / ...
}
/ / ...
 }
```

A binary_search() returns a boolindicating whether a value was present. As with find(), we often also want to know where the elements with that value are in that sequence

Heaps: The word heap means different things in different contexts. When discussing algorithms, "heap" often refers to a way of organizing a sequence such that it has a first element that is the element with the highest value. Addition of an element (using push_heap()) and removal of an element (using pop_heap()) are reasonably fast.

A heap is implemented by this set of functions:

- template void push_heap(Ran first, Ran last) ;
- template void pop_heap(Ran first, Ran last) ;

Min and Max: The algorithms described here select a value based on a comparison. It is obviously useful to be able to find the maximum and minimum of two values:
```
template <class T> const T& max(const T& a, const T& b)
{
                return ( a<b) ? b : a;
}
```

The max() and min() operations can be generalized to apply to sequences in the obvious manner: template<class For> For max_element(For first, For last);
template<class For> For min_element(For first, For last);

min_element and max_element: The min_element and max_element algorithms find the min and max element in a container class:

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
   using namespace std;
   list<int> li;
```

```
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);
    list<int>::const_iterator it; // declare an iterator
    it = min_element(li.begin(), li.end());
        cout << *it << " ";
    it = max_element(li.begin(), li.end());
        cout << *it << " ";
    cout << endl;
}
```

Prints:
0 5

find (and list::insert): In this example, we'll use the find() algorithm to find a value in the list class, and then use the list::insert() function to add a new value into the list at that point.

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
    using namespace std;
    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);
   list<int>::iterator it; // declare an iterator
    it = find(li.begin(), li.end(), 3); // find the value 3 in the list
    li.insert(it, 8); // use list::insert to insert the value 8 before it
    for (it = li.begin(); it != li.end(); it++) // for loop with iterators
        cout << *it << " ";
    cout << endl;
}
```

This prints the value

0 1 2 8 3 4 5

sort and reverse: In this example, we'll sort a vector and then reverse it.

```
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
    using namespace std;

    vector<int> vect;
    vect.push_back(7);
    vect.push_back(-3);
    vect.push_back(6);
    vect.push_back(2);
    vect.push_back(-5);
    vect.push_back(0);
```

```
    vect.push_back(4);
    sort(vect.begin(),  vect.end());  // sort the list
    vector<int>::const_iterator  it; // declare an iterator
    for (it = vect.begin();  it != vect.end();  it++) // for loop with iterators
       cout << *it << " ";
   cout << endl;
    reverse(vect.begin(),  vect.end());  // reverse the list
    for (it = vect.begin();  it != vect.end();  it++) // for loop with iterators
       cout << *it << " ";
    cout << endl;
}
```

This produces the result:
-5 -3 0 2 4 6 7

7 6 4 2 0 -3 -5

Note that sort() doesn't work on list container classes -- the list class provides its own sort() member function, which is much more efficient than the generic version would be.

Although this is just a taste of the algorithms that STL provides, it should suffice to show how easy these are to use in conjunction with iterators and the basic container classes.