UNIT V

Chapter 1 - Introduction to the Standard Template Library (STL)

1.1 Introduction

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- A collection of generic classes and functions is called the Standard Template Library(STL).STL components are part of C++ standard library are defined in the **namespace std.**
- It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.

1.2 Components of STL:

STL has several components. But as it core are 3 key components.

1.Containers

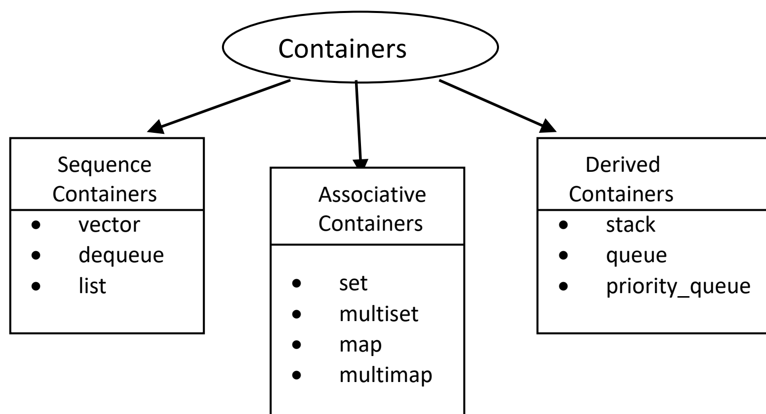
2.Algorithms

3. Iterators

These 3 components work in conjunction with one another to provide support to a variety of programming solutions.

1.3 Containers

- Containers are object that actually hold data of same type(ie.store data).
- It is a way data is organized in memory.
- The STL containers are implemented by template classes.
- The STL defines ten containers which are grouped into three categories.

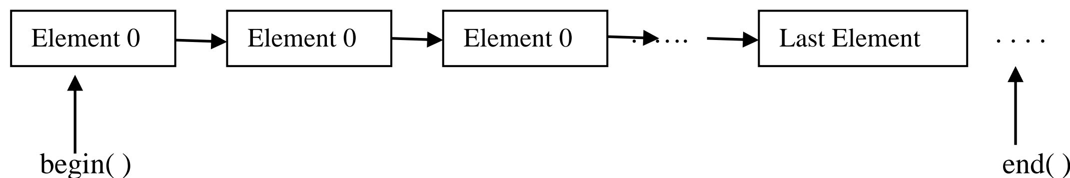


The following table shows the containers supported by the STL.

Container	Description	Header File	Iterator
vector	A dynamic array. Allows insertions and deletions at back.	<vector>	Random access
list	A bidirectional linear list. Allows insertions and deletions at anywhere.	<list>	Bidirectional
dequeue	A double-ended queue. Allows insertions and deletions at both the ends.	<deque>	Random Access
set	Storing unique sets.Allows rapid look up.	<set>	Bidirectional
multiset	Storing nonunique sets.(Duplicates allowed)	<set>	Bidirectional
map	Storing unique key/value pairs. Each key is associated with only one value.	<map>	Bidirectional
multimap	Storing unique key/value pairs. one key may be associated more than one value.	<map>	Bidirectional
stack	A standard stack.Last-in-first-out(LIFO)	<stack>	No iterator
Queue	A standard queue. First-in-first-out(FIFO)	<queue>	No iterator
Priority-queue	A priority queue. Thefirst element out is always the highest priority element.	<queue>	No iterator

i) Sequence Containers:

- It store elements in a linear sequence like a line .It provides 3 containers **vector, list, deque.**
- Each element is related to other elements by its position along the line.



Elements in all these containers can be accessed using an iterator.These 3 sequence containers are differentiated by their performance of

1. Speed of random access (vector and deque are fast random access, list is slow random aceess)

2.Insertions and deletions of elements.

In the middle Insertions and deletions of elements slow in vector and deque, fast in list.
At the end Insertions and deletions of elements ,vector fast at back, list fast at front,deque fast at both the ends.)

ii) Associative Containers :

- It supports direct access to elements using keys.
- They are not sequential. There are 4 types ,
  1. set
  2. multiset
  3. map
  4. multimap
- All these containers store data in a **tree structure.**

- Containers set and multiset can store number of items and provide operations for manipulating them using the values as the keys.

  - **Difference : set does not allow duplicate items while multiset does.**

- Containers map and multimap are used to store pairs of items, one called key, and the other called value.Manipulate the values using the keys associated with them.

  - **Difference: map allows only one key for a given valueto be stored while multimap permits multiple keys.**

iii) Derived Containers:

- It also called container adaptors.There are 3 types,
  1. Stack
  2. Queue
  3. Priority queue
- These are created from different sequence containers.
- It does not support iterators.
- They support **pop( ) and push( )** for **deleting and inserting** operations.

1.4 Algorithms

- Algorithms are a set of functions or methods provided by STL that act on containers. These are built-in functions and can be used directly with the STL containers and iterators instead of writing our own algorithms.

- To access the STL algorithm , we must include **<algorithm>** in our program.

STL supports the following types of algorithms:

- Retrieve or non-mutating algorithms
- Mutating algorithms
- Sorting algorithms
- Set algorithms algorithms
- Relational algorithms

Non-mutating Algorithms:

Operations	Description
adjacent_find( )	Finds adjacent pair of objects that are equal
count( )	Counts occurrence of a value in a sequence
count_if( )	Counts number of elements that matches a predicate
equal( )	True if two ranges are same
find( )	Finds first occurrence of a value in a sequence
find_end( )	Finds last occurrence of a value in a sequence
for_each( )	Apply an operation to each element
mismatch( )	Finds first elements for which two sequences differ.
search( )	Finds subsequence within a sequence
search_n( )	Finds a sequence of a specified number of similar elements.

Mutating Algorithms:

Operations	Description
copy( )	Copies a sequence
fill( )	Fills a sequence with a specified value
generate( )	Replaces all elements with the result of an operation
random_shuffler( )	Places elements in random order
remove( )	Deletes elements of specified value
replace( )	Replaces elements with a specified value
reverse( )	Reverse the order of elements
rotate( )	Rotates elements
swap( )	Swaps two elements
unique( )	Delete equal adjacent elements

Sorting Algorithms :

Operations	Description
binary_search( )	Conducts a binary search on an ordered sequence
equal_range( )	Finds a subrange of elements with a given value
inplace_merge( )	Merges two consecutive sorted sequences
lower_bound( )	Finds the first occurrence of a specified value
make_heap( )	Makes a heap from a sequence
merge( )	Merges two sorted sequences
partition( )	Places elements matching a predicate first
sort( )	Sorts a sequence
sort_heap( )	Sorts a heap
upper_bound( )	Finds the last occurrence of a specified value
stable_sort( )	Sorts maintaining order of equal elements

Set Algorithms

Operations	Description
includes( )	Finds whether a sequence is a subsequence of another
set_difference( )	Constructs a sequence that is the difference of two ordered sets
set_intersection( )	Constructs a sequence that contains the intersection of ordered sets
set_union( )	Produces sorted union of two ordered sets

Relational Algorithms :

Operations	Description
equal( )	Finds whether two sequences are same
lexicographical_compare( )	Compares alphabetically one sequence with other

max( )	Gives maximum of two values
max_elements( )	Finds maximum elements within the sequence
min( )	Gives minimum of two values
min_elements( )	Finds minimum elements within the sequence
mismatch( )	Finds th first mismatch between the elements in two sequences

Numeric Algorithms :

Operation	Description
accumulate( )	Accumulates the result of operation on sequence
adjacent_difference( )	Produces a sequence from another sequence
inner_product( )	Accumulates the result of operation on a pair of sequences
partial_sum( )	Produces a sequence by operation on a pair of sequences

1.5 Iterators

- Iterators are constructs that we use to traverse or step through containers in STL.
- Iterators are very important in STL as they act as a bridge between algorithms and containers.
- Iterators always point to containers and in fact algorithms actually, operate on iterators and never directly on containers.

Iterators are of following types:

- **Input Iterators:** Simplest and is used mostly in single-pass algorithms.
- **Output Iterators:** Same as input iterators but not used for traversing.
- **Bidirectional Iterators:** These iterators can move in both directions.
- **Forward Iterators:** Can be used only in the forward direction, one step at a time.
- **Random Access Iterators:** Same as pointers. Can be used to access any element randomly.

1.6 Application of Container Classes

Three most popular container classes are **vector, list, map**.

i) Vectors

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
- Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.
- Class vector supports a number of constructors for creating vector objects.

```
vector<int>  v1;                // zero-length int vector
vector<double>  v2(10);   // 10 element double vector
vector<int>  v3(v4);        // creates v3 from v4
vector<int>  v(5,4);         // 5 element vector of 2s
```

- The vector class supports several member functions :

at(g) – Returns a reference to the element at position ‘g’ in the vector

front( ) – Returns a reference to the first element in the vector

back( ) – Returns a reference to the last element in the vector

data( ) – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

size( ) – Returns the number of elements in the vector.

max_size( ) – Returns the maximum number of elements that the vector can hold.

capacity( ) – Returns the size of the storage space currently allocated to the vector expressed as number of elements.

resize(n) – Resizes the container so that it contains ‘n’ elements.

empty( ) – Returns whether the container is empty.

shrink_to_fit( ) – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.

reserve( ) – Requests that the vector capacity be at least enough to contain n elements.
begin( ) – Returns an iterator pointing to the first element in the vector

end( ) – Returns an iterator pointing to the theoretical element that follows the last element in the vector

rbegin( ) – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

rend( ) – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

cbegin( ) – Returns a constant iterator pointing to the first element in the vector.

cend( ) – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

crbegin( ) – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

crend( ) – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

assign( ) – It assigns new value to the vector elements by replacing old ones

push_back( ) – It push the elements into a vector from the back

pop_back( ) – It is used to pop or remove elements from a vector from the back.

insert( ) – It inserts new elements before the element at the specified position

erase( ) – It is used to remove elements from a container from the specified position or range.

swap( ) – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.

clear( ) – It is used to remove all the elements of the vector container

emplace( ) – It extends the container by inserting new element at position

emplace_back( ) – It is used to insert a new element into the vector container, the new element is added to the end of the vector

Example :

```
// C++ program to illustrate the iterators in vector
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
   vector<int> g1;

    for (int i = 1; i <= 5; i++)
       g1.push_back(i);

   cout << "Output of begin and end: ";
   for (auto i = g1.begin( ); i != g1.end( ); ++i)
       cout << *i << " ";

   cout << "\nOutput of cbegin and cend: ";
```

```
    for (auto i = g1.cbegin( ); i != g1.cend( ); ++i)
        cout << *i << " ";

    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin( ); ir != g1.rend( ); ++ir)
        cout << *ir << " ";

    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin( ); ir != g1.crend( ); ++ir)
        cout << *ir << " ";

    return 0;
}
```

Output:

Output of begin and end: 1 2 3 4 5

Output of cbegin and cend: 1 2 3 4 5

Output of rbegin and rend: 5 4 3 2 1

Output of crbegin and crend : 5 4 3 2 1

ii) Lists :

- Lists are sequence containers that allow non-contiguous memory allocation.
- As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.
- It supports a bidirectional, linear list. For implementing a singly linked list, we use forward list.
- Class list provides many member functions for manipulating the elements of list
- Functions used with List:

  front( ) – Returns the value of the first element in the list.

  back( ) – Returns the value of the last element in the list .

  begin( ) and end( ) – begin( ) function returns an iterator pointing to the first element of the list

  end( )– end( ) function returns an iterator pointing to the theoretical last element which follows the last element.

  empty( ) – Returns whether the list is empty(1) or not(0).

  insert( ) – Inserts new elements in the list before the element at a specified position.

erase( ) – Removes a single element or a range of elements from the list.

assign( ) – Assigns new elements to list by replacing current elements and resizes the list.

remove( ) – Removes all the elements from the list, which are equal to given element.

remove_if( )– Used to remove all the values from the list that correspond true to the predicate or condition given as parameter to the function.

reverse( ) – Reverses the list.

size( ) – Returns the number of elements in the list.

resize( )– Used to resize a list container.

sort( ) – Sorts the list in increasing order.

max_size( ) – Returns the maximum number of elements a list container can hold.

unique( ) - Removes all duplicate consecutive elements from the list.

clear( )– clear( ) function is used to remove all the elements of the list container, thus making it size 0.

swap( )– This function is used to swap the contents of one list with another list of same type and size.

splice( ) - Used to transfer elements from one list to another.

merge( ) - Merges two sorted lists into one

emplace( ) – Extends list by inserting new element at a given position.

Example :

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;

//function for printing the elements in a list
void showlist(list <int> g)
{
   list <int> :: iterator it;
   for(it = g.begin( ); it != g.end( ); ++it)
       cout << '\t' << *it;
   cout << '\n';
}

int main( )
{

   list <int> gqlist1, gqlist2;
```

```
    for (int i = 0; i < 10; ++i)
    {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist1.front( ) : " << gqlist1.front( );
    cout << "\ngqlist1.back( ) : " << gqlist1.back( );

    cout << "\ngqlist1.pop_front( ) : ";
    gqlist1.pop_front( );
    showlist(gqlist1);

    cout << "\ngqlist2.pop_back( ) : ";
    gqlist2.pop_back( );
    showlist(gqlist2);

    cout << "\ngqlist1.reverse( ) : ";
    gqlist1.reverse( );
    showlist(gqlist1);

    cout << "\ngqlist2.sort( ): ";
    gqlist2.sort( );
    showlist(gqlist2);

    return 0;

}
```

The output of the above program is :

```
List 1 (gqlist1) is :    0    2    4    6
8    10    12    14    16    18

List 2 (gqlist2) is :    27    24    21    18
15    12    9    6    3    0

gqlist1.front( ) : 0
gqlist1.back( ) : 18
gqlist1.pop_front( ) :    2    4    6    8
10    12    14    16    18

gqlist2.pop_back( ) :    27    24    21    18
15    12    9    6    3
```

gqlist1.reverse( ) :   18   16   14   12
10   8   6   4   2

gqlist2.sort( ):   3   6   9   12
15   18   21   24   27

iii) MAPS

- Maps are associative containers that store elements in a mapped fashion.
- Each element has a key value and a mapped value.
- No two mapped values can have same key values.
- Some basic functions associated with Map:
  begin( ) – Returns an iterator to the first element in the map
  end( ) – Returns an iterator to the theoretical element that follows last element in the map
  size( ) – Returns the number of elements in the map
  max_size( ) – Returns the maximum number of elements that the map can hold
  empty( ) – Returns whether the map is empty
  pair insert(keyvalue, mapvalue) – Adds a new element to the map
  erase(iterator position) – Removes the element at the position pointed by the iterator
  erase(const g) – Removes the key value 'g' from the map
  clear( ) – Removes all the elements from the map

Example:

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

using namespace std;

int main( )
{

    // empty map container
    map<int, int> gquiz1;

    // insert elements in random order
    gquiz1.insert(pair<int, int>(1, 40));
    gquiz1.insert(pair<int, int>(2, 30));
    gquiz1.insert(pair<int, int>(3, 60));
    gquiz1.insert(pair<int, int>(4, 20));
    gquiz1.insert(pair<int, int>(5, 50));
    gquiz1.insert(pair<int, int>(6, 50));
    gquiz1.insert(pair<int, int>(7, 10));

    // printing map gquiz1
    map<int, int>::iterator itr;
    cout << "\nThe map gquiz1 is : \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz1.begin( ); itr != gquiz1.end( ); ++itr)
```

```
{
    cout << '\t' << itr->first
         << '\t' << itr->second << '\n';
  }
  cout << endl;

  // assigning the elements from gquiz1 to gquiz2
  map<int, int> gquiz2(gquiz1.begin( ), gquiz1.end( ));

  // print all elements of the map gquiz2
  cout << "\nThe map gquiz2 after"   << " assign from gquiz1 is : \n";
  cout << "\tKEY\tELEMENT\n";
  for (itr = gquiz2.begin( ); itr != gquiz2.end( ); ++itr) {
    cout << '\t' << itr->first  << '\t' << itr->second << '\n';
  }
  cout << endl;

  // remove all elements up to  element with key=3 in gquiz2
  cout << "\ngquiz2 after removal of" " elements less than key=3 : \n";
  cout << "\tKEY\tELEMENT\n";
  gquiz2.erase(gquiz2.begin( ), gquiz2.find(3));
  for (itr = gquiz2.begin( ); itr != gquiz2.end( ); ++itr) {
    cout << '\t' << itr->first  << '\t' << itr->second << '\n';
  }

  // remove all elements with key = 4
  int num;
  num = gquiz2.erase(4);
  cout << "\ngquiz2.erase(4) : ";
  cout << num << " removed \n";
  cout << "\tKEY\tELEMENT\n";
  for (itr = gquiz2.begin( ); itr != gquiz2.end( ); ++itr) {
    cout << '\t' << itr->first
         << '\t' << itr->second << '\n';
  }

  cout << endl;

  // lower bound and upper bound for map gquiz1 key = 5
  cout << "gquiz1.lower_bound(5) : "   << "\tKEY = ";
  cout << gquiz1.lower_bound(5)->first << '\t';
  cout << "\tELEMENT = "  << gquiz1.lower_bound(5)->second << endl;
  cout << "gquiz1.upper_bound(5) : "  << "\tKEY = ";
  cout << gquiz1.upper_bound(5)->first << '\t';
  cout << "\tELEMENT = " << gquiz1.upper_bound(5)->second << endl;

  return 0;
}
```

Output:

The map gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 20

5 50

6 50

7 10

The map gquiz2 after assign from gquiz1 is :

KEY ELEMENT

1 40

2 30

3 60

4 20

5 50

6 50

7 10

gquiz2 after removal of elements less than key=3 :

KEY ELEMENT

3 60

4 20

5 50

6 50

7 10

gquiz2.erase(4) : 1 removed

KEY ELEMENT

3 60

5 50

6 50

7 10

gquiz1.lower_bound(5) : KEY = 5 ELEMENT = 50

gquiz1.upper_bound(5) : KEY = 6 ELEMENT = 50

1.7 Function Objects:

- A function object is a function that has been wrapped in a class, like an object.
- The class has only one member functions, the overloaded ( ) operator and no data.
- Function objects are used as arguments to certain containers and algorithms.Example:

  Sort(array, array+5,greater<int>( ))

- STL provides many predefined function objects for performing arithmetical and logical operations. Here the variables x and y represent objects of class T.

  i) Arithmetic function object

  divides< T > - x/y
  minus< T > - x-y
  modulus< T > - x%y
  negate< T > - -x
  plus< T > - x+y
  multiplies< T > - x*y

  ii) Relational function object
  equal_to<T> - x==y
  greater<T> - x>y
  greater_equal<T>- x>=y
  less<T> - x<y
  less_equal<T> - x<=y
  not_equal_to<T>- x!=y

  iii) Logical Function Object

logical_and<T> - x&&y
logical_not<T> - !x
logical_or<T> - x||y

Example:

```
#include<iostream>
#include<algorithm>
#include<functional>
using namespace std;
int main()
{
   int x[ ] = { 10,50,30,40,20};
   int y[ ] = { 70,90,60,80};
sort(x,x+5,greater<int>());
sort(y,y+4);
for(int i=0;i<5;i++)
 cout<<x[i]<<" ";
 cout<<"\n";

for(int i=0;i<4;i++)
 cout<<y[i]<<" ";
cout<<"\n";

int z[9];
merge(x,x+5,y,y+4,z);

for(int i=0;i<9;i++)
 cout<<z[i]<<" ";
cout<<"\n";

return(0);
}
```

OUTPUT:

```
 50   40   30   20   10

 60   70   80   90

50   40   30   20   10     60   70   80   90
```