UNIT-I

STANDARD TEMPLATE LIBRARY

* Standard Template Library (STL) is a set of C++ template classes, that provide generic classes and functions that can be used to implement data structures and algorithms.
* STL is mainly composed of :

(i) Containers
(ii) Algorithms
(iii) Iterators.

} Components of STL

* Containers
  - Container library in STL provides containers that are used to create data structures like arrays, linked list, trees, etc.
  - These containers are generic and can hold elements of any data type.
* Algorithms
  - Algorithms library in STL contains built-in functions that peyorms complex algorithms on data structures.
  - provides abstraction → not necessary to know how the algorithm works.
* Iterators
  - Iterators are used for working upon a sequence of values on containers
  - Acts as a bridge between containers and algorithms.

* Use and Application of STL

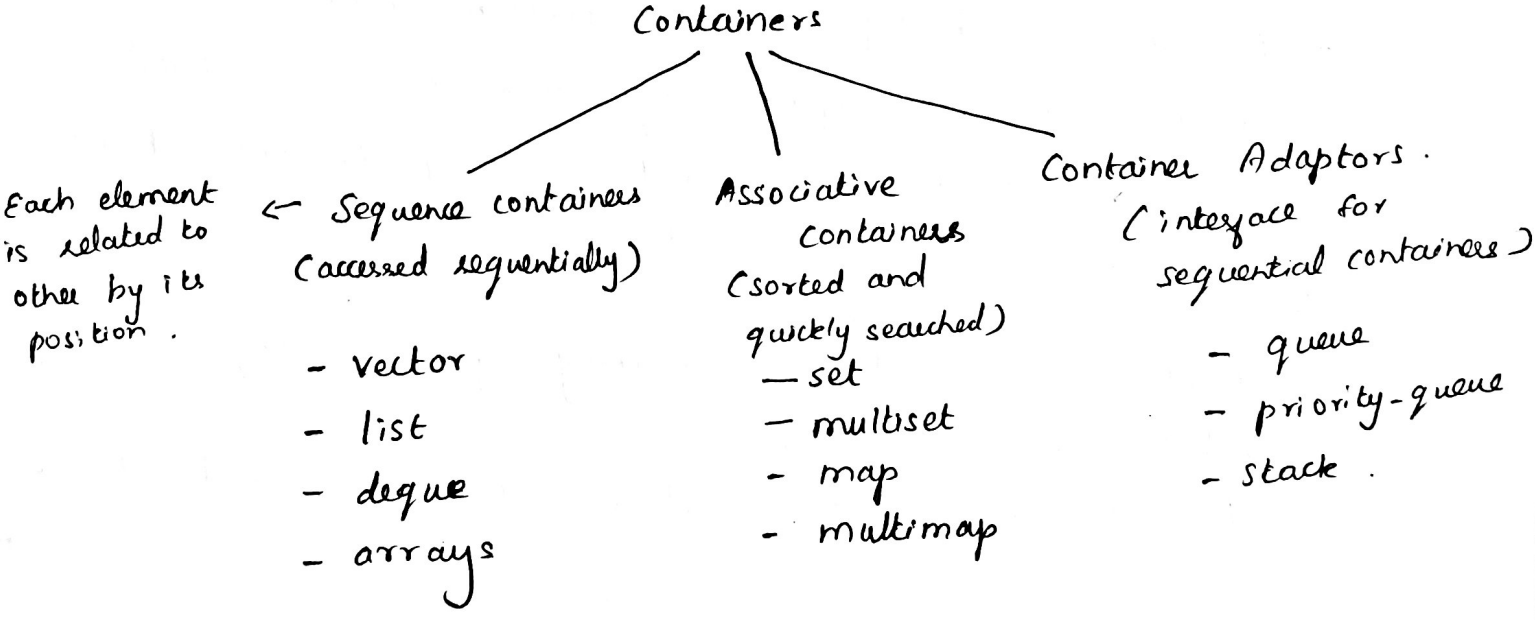
* STL being generic library provide containers and algorithms that can be used to store and manipulate different types of data

* Does not require to redefine data structures and algorithms for each data type.
* Saves lot of time, code and effort during programming.
* Reliable and fast.

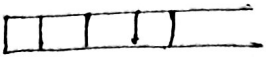
Containers

* Containers or container classes are used to store class objects and data consisting of built-in types.

* Containers are classified into 7 standard container classes and 3 container adaptor classes.



1) Vector



- Vectors are dynamic arrays that resize itself automatically during runtime while insertion or deletion.
- Vector elements are placed in contiguous storage.
- Vector grows & shrink at the end.
- Similar to but more powerful than built-in C++ arrays.

Syntax for creating vector:

```
vector <obj type> vector_name;
```

Adv:
* Quick random access
* Quick to insert & erase at end

Disadv
* Slow to insert or erase in middle (bcoz need to shift all elements)

(Eg)
```
#include <vector>
void main()
{
  vector <int> vec;   ⇒ creates a blank vector
}
```

Functions of vector class

1) assign() – Assigns values to a vector from beginning to end. Previous contents are removed.
2) at() – Returns a reference to a vector element at specified location
3) back() – Returns reference to last element of vector.
4) begin() or – Returns reference to first element of vector.
5) clear() – deletes all elements of vector
6) empty() – returns true, if there is no element in vector, false otherwise.
7) erase() – deletes element at indexed location
8) insert() – insert elements in vector a) one value b) a no. of copies of value at specified location c) insert from start and end before location.
9) max_size – returns maximum no. of element a vector can hold.

10) pop_back() - removes last element of vector

11) push_back(g) - appends an element 'g' at back of vector.

12) swap() - exchanges elements of two vectors.

13) size() - Returns current number of elements in vector.

(Eg)
```
#include <iostream.h>
#include <vector.h>
using namespace std.
void 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 << " Values : " << vec[i] << endl;
// Using iterator to acces the values
  vector <int> :: iterator v;v=vec.begin();
  while (v != vec.end())
    {
     cout << " Values using iterator:" << *v << endl;
     v++;
    }
}
```

o/p
```
Vector size : 0
Extended vector size : 5
 Values : 0
  Values : 1
    "    : 2
    "    : 3
         : 4
Values with iterator : 0
                     : 1
                     2
                     3
                     4
```

Note: - STL classes are part of std namespace
- Include "using namespace std;" at top of program (or)
- prefix every container class type with std::
(Eg) std::vector <int> vec;

For insertion of element in vector

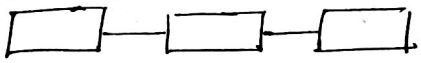
v => vector

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

v.insert(i, 70); => insert 70 before the position pointed by i.

v.insert(i, 2, 70); => insert 70 twice before v.begin()

(ii) List



* Lists are sequence containers that allow non-contiguous memory allocation.
* Used to implement linked list.
* List has slow traversal but quick insertion and deletion of elements once the position is found.

Syntax for creating new linked list:

```
#include <iostream.h>
#include <list.h>
void main()
{
  list <int> l; // Creates a new empty list l.
}
```

instead of

```
#include <iostream.h>
struct node
{ int data;
  struct node *next;
}
void main()
{
struct node *list1
  = NULL;
}
```

Functions of list class.

1. front() - returns reference to first element in the list
2. back() - returns reference to last element in list.
3. push_front(g) - Adds a new element 'g' at beginning of the list.
4. push_back(g) - Adds new element 'g' at end of list.

5) pop_front() - Removes first element of list and size of list by 1.

6) pop_back() - Removes last element and " "

7) begin() - Returns an iterator pointing to first element.

8) end() - Returns an iterator pointing to theoretical last element which follows last element.

9) empty() - Returns whether the list is empty or not.

10) insert() - inserts new element in list before element at specified position.

11) erase() - removes single or range of elements.

12) assign() - assigns new element by replacing current element.

13) size() - returns no. of elements.

14) reverse() - reverses the list.

15) sort() - sort the list in increasing order.

(Eg)
```
#include <iostream.h>
#include <list>
#include <iterator>
using namespace std;
void show (list <int> g)
{
list <int> :: iterator it;
for (it = g.begin(); it != g.end(); it++)
    cout << *it << " ";
}
```

```
void main()
{
list <int> list1, list2;
for (int i=0; i<5; i++)
 {
  list1.push_back (i*2);
  list2.push_front (i*3);
}
cout << "List1 is :\n";
 show (list1);
cout << " List2 is :\n";
 show (list2);
 cout << " Front of list1 : " << list1.front();
 cout << " Back of list1 :" << list1.back();
  cout << " Pop from front of list1 :";
  list1.pop_front ();
  show (list1);
  cout << " Pop from back of list2 :";
  list2.pop_back ();
  show (list2);
 cout << " Reverse list1 \n";
  list1.reverse ();
   show (list1);
  cout << "Sort list2 \n";
   list2.sort ();
   show (list2);
}
```

O/P

List1 is : 0 2 4 6 8

List2 is : 12 9 6 3 0

Front of list1 : 0

Back of List1 : 8

Pop from front of list1

2 4 6 8

Pop from back of list2

12 9 6 3

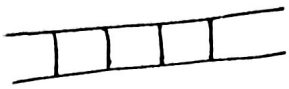
Reverse List 1

8 6 4 2 0

Sort List2

3 6 9 12

(iii) Deque



* Shorthand for doubly ended queue.
* Allows fast insertion and deletion at both ends of the queue.
* Contiguous storage allocation may not be guaranteed.
* Functions for deque are same as vector, with addition of push and pop operations for both front and back.

(Eg)

```
#include <iostream.h>
#include <deque>
using namespace std;
void show (deque <int> g)
{
 deque <int> :: iterator it;
 for (it = g.begin(); it != g.end(); it++)
      cout << *it;
}
void main
{
 deque <int> dq;
 dq.push_back(10);
 dq.push_front(20);
 dq.push_back(30);
 dq.push_front(15);
cout << "Deque is : ";
 show (dq);
cout << "Size : " << dq.size();
cout << "Max size : " << dq.max_size();
cout << "At (2) : " << dq.at(2);
cout << "Front : " << dq.front();
cout << "Back : " << dq.back();
cout << "Pop front : " << dq.pop_front();
 show(dq);
cout << "Pop back : " << dq.pop_back();
 show(dq);
}
```

o/p:

```
Deque is : 15 20 10 30
Size : 4
Max size : 1073741823
At(2) : 10
Front : 15
Back : 30
Pop front : 20 10 30
Pop back : 20 10
```

Array

- Array container in STL provides static arrays. (ie) size must be specified in source code.

(Eg)
```
#include <array>
void main()
{
 array <int, 4> a1 ;   // creates an array a1 of size 4.
}
```

Member functions of array

1) at() - returns value at given position
2) front() - returns first element
3) back()
4) swap() - swap contents of 2 arrays of same type & size
5) size()
6) max_size()
7) begin()
8) end()
9) empty()

Eg)
```
#include <array>
#include <iostream.h>
void main()
{
 array <int, 5> a = {1, 2, 3, 4, 5} ;
 array <int, 5> b = {10, 20, 30, 40, 50} ;
 a.swap(b) ;
 cout << "a is \n" ;
 for(int i = 0 ; i < 5 ; i++)
    cout << a.at(i) << " " ;
 cout << " b is \n";
 for(int i = 0 ; i < 5 ; i++)
    cout << b.at(i) << " " ;
}
```

o/p

a is :
10 20 30 40 50

b is :
1 2 3 4 5

ASSOCIATIVE CONTAINERS

* Associative container is not sequential.
* Uses keys to access data.
* The keys are numbers or strings → used automatically by the container to arrange stored elements in specific order.
* Two types of associative containers:
  (i) Sets
  (ii) Maps
* Both store data in a tree structure → offers fast searching, insertion and deletion.
* Sets and Maps are versatile data structures suitable for wide variety of applications.

(i) Set

- each element has to be unique because the element is identified by its value.
- Value of element cannot be modified in a set.
  → but element can be removed, modified & added.

Functions:

1) find (g) → returns an iterator to the element 'g' in the set if found, else returns iterator to end
2) count (g) → returns no. of matches to element 'g' in set.
3) lower_bound (g) → returns iterator to first element equivalent to g
4) upper_bound ( ) → returns iterator to upper bound.

(F7)

```
#include <iostream>
#include <set>
using namespace std;
void main()
{
  set <int> S1;
  S1.insert(40);
  S1.insert(20);
  S1.insert(10);
  S1.insert(20);
  S1.insert(30);

set <int> :: iterator itr;
cout << "The set is : ";
for(itr = S1.begin(); itr != S1.end(); itr++)
  {
   cout << *itr << " ";          // prints 40 30 20 10.
  }
S1.erase(S1.begin(), S1.find(30));   // remove all elements upto element with value 30
for(itr = S1.begin(); itr != S1.end(); itr++)
  {
  cout << *itr << " ";           // Prints 30 20 10.
  }
}
```

erase:

S1.erase(30); => removes all elements with value 30.

S1.lower_bound(20); => prints 20.

S1.upper_bound(20); => prints 10.

(ii) Multiset

- Similar to set except that.
- but multiple elements can have same values

(Eg)
```
#include <iostream.h>
#include <set>
using namespace std;
void main()
{
 multiset <int> m1;
 m1.insert(40);
 m1.insert(20);
 m1.insert(10);
 m1.insert(20);
 m1.insert(30);
multiset <int> :: iterator itr;
cout << "The multiset is \n";
for(itr = m1.begin(); itr != m1.end(); itr++)
    cout << *itr << " ";        // prints 40 30 20 20 10.
 m1.erase(20);  // remove all elements with value 20
for(itr = m1.begin(); itr != m1.end(); itr++)
    cout << *itr << " ";  // prints 40 30 10
}
```

Map :

* 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 keys.

(Eg)

```
#include <iostream>
#include <map>
using namespace std;
void main()
{
 map <int, int> m1;
 m1.insert (pair <int, int> (1, 40));
 m1.insert (pair <int, int> (2, 30));
 m1.insert (pair <int, int> (3, 80));
 map <int, int> :: iterator itr;
 cout << "The mapping is : \n";
 for (itr = m1.begin(); itr != m1.end(); itr++)
    {
     cout << itr -> first << "\t" << itr -> second << "\n";
    }
}
```

o/p :

```
1   40
2   30
3   80.
```

Note :

* pair is a container, consists of two data elements or objects.
* Used to store two heterogenous objects as a single unit.
* First element is referenced as 'first' and second element as 'second'.

(iv) Multimap :

- Similar to map.
- but multiple elements can have same keys.
- The key value and mapped value pair has to be unique in this case.

(Eg)

```
#include <iostream>
#include <map>
using namespace std;
void main()
 {
  multimap <int, int> mm;
  mm.insert (pair <int,int> (1,40));
  mm.insert (pair <int,int> (2,30));
  mm.insert (pair <int,int> (2,50));
  mm.insert (pair <int,int> (6,80));
 multimap <int,int> ::iterator itr;
 cout << "The mapping is : \n";
 for (itr = mm.begin(); itr != mm.end(); itr++)
    cout << itr -> first << "\t" << itr -> second << "\n";
  int n;
n=mm.erase(2); // removes all elements with key = 2
cout << n << " removed";
cout << " After removal \n";
for (itr = mm.begin(); itr != mm.end(); itr++)
    cout << itr -> first << "\t" << itr -> second << "\n";
}
```

o/p :

```
The mapping is
1   40
2   30
2   50
6   80
```

```
2 removed.
After removal
1   40
6   80
```

Container Adaptors

- provide a different interface for sequential containers.

(i) Stack

* Stack is a type of container adaptors with LIFO (Last In First Out) type of working.
* Insertion and deletion is always performed at top of stack.

Member functions

1) empty() - returns whether stack is empty
2) size() - returns size of stack.
3) top() - returns reference to top most element of stack.
4) push(g) - Adds the element 'g' at top of stack.
5) pop() - Deletes top most element of stack.

Eg)
```
#include <iostream>
#include <stack>
using namespace std;
void show (stack <int> s)
  {
    while (!s.empty())
      {
        cout << s.top() << "\t";
        s.pop();
      }
  }
void main()
{
  stack <int> s1;
  s1.push(10);
  s1.push(30);
  s1.push(20);
  s1.push(5);
  cout << "The stack is \n";
  show (s1);
  cout << "\nstack size : " << s1.size();
  cout << "\nStack top : " << s1.top();
  cout << "\n After popping \n";
  s1.pop();
  show (s1);
}
```

o/p:

The stack is

5 20 30 10

Stack size : 4

Stack top : 4

After popping

20 30 10.

(ii) Queue

- Queue is a type of container adaptors with FIFO (First In First Out) type of working.
- Elements are inserted at back (end) and deleted from front.

Member functions

1) empty() 2) size()
3) front() - returns reference to first element of queue
4) back() - " " " last element of queue.
5) push(g) - Adds element 'g' at end of queue
6) pop() - Deletes first element of queue

(Eg)

```
#include <iostream>
#include <queue>
using namespace std;
void show (queue <int> q1)
 {
 while (!q1.empty())
   {
    cout << q1.front();
    q1.pop();
   }
 }
void main()
 {
 queue <int> q;
 q.push(10);
 q.push(20);
 q.push(40);
 q.push(50);
 cout << "The queue is:\n";
 show(q);
 cout << "\n Queue size : " << q.size();
 cout << "\n Front of Queue : " << q.front();
 cout << "\n Last ele. of Queue : " << q.back();
 q.pop();
 cout << "After deleting an element\n";
 show(q);
}
```

O/P:

The queue is : 10 20 40 50

Queue size : 4

Front of Queue : 10

Last ele. of Queue : 50

After deleting an element

20 40 50

Priority Queue

* Type of container adaptor
* Similar to queue, except that the first element in priority queue is always the greatest of all the elements.
* Used to replicate Max Heap data structure.

Member functions

1) empty() 2) size()

3) top() - Returns a reference to top most (first) element of the queue.

4) push(g) - Adds the element 'g' to the queue.

5) pop() - Deletes first element of queue.

(Eg)

```
#include<iostream>
#include<queue>
using namespace std;
void show(priority_queue<int> pq)
  {
   while(!pq.empty())
     {
      cout<<pq.top();
      pq.pop();
     }
  }
void main()
 {
 priority_queue<int> p;
 p.push(30);
 p.push(40);
 p.push(90);
 p.push(60);
 cout<<"The priority queue is\n";
 show(p);
 cout<<"\nQueue size :"<<p.size();
 cout<<"\n First element :"<<p.top();
 cout<<"\nAfter deleting an element\n";
 p.pop()
 show(p);
}
```

o/p:

The priority queue is

90 60 40 30

Queue size : 4

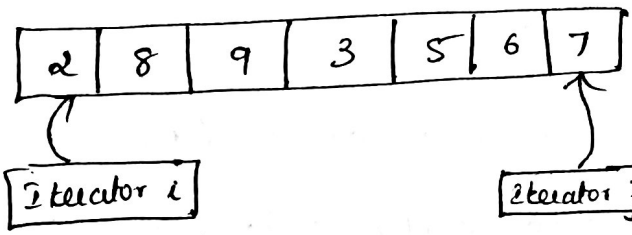
First element : 90

After deleting an element

60 40 30.

ITERATORS

* Iterators are used to point the memory addresses of STL containers and to access the data stored.
* They are used in sequence of numbers, characters. etc.
* They reduce the complexity and execution time of program.



points to beginning of vector

points to end of vector.

Defining an iterator

Syntax:

```
container_type <parameter_list> :: iterator iterator_name;
```

(Eg)

```
# include <list>
# include <iostream.h>
# include <vector>
using namespace std
void main()
{
  vector <int> :: iterator vi;   // creates an iterator vi to a vector of integers.
  vector <char> :: iterator ci;  // creates iterator ci to a vector of characters.
  list <int> :: iterator li;     // creates iterator li to a list of integers.
}
```

Operations of Iterators

1) begin() - used to return iterator, pointing to beginning position of container

2) end() - " " end position of container

(Eg)
```
#include <iostream>
#include <vector>
using namespace std;
void main()
 {
  vector <int> ar = {1, 2, 3, 4, 5};
  vector <int> :: iterator ptr;
  cout << "Vector elements \n";
  for( ptr = ar.begin(); ptr < ar.end(); ptr++)
      cout << *ptr << " ";   // to de reference the
 }                                iterator to get element
                                  pointed by it.
```

3) advance() - used to increment iterator position by the value of distance.

Syntax:

```
advance (iterator i, int distance);
```

- If distance is -ve, iterator will be decremented.

(Eg)
```
ptr = ar.begin();   // ptr pts to beg. of vector ar
advance (ptr, 3);   // ptr pts to third element of vector.
```

4) distance() - returns no. of elements or distance between first and last iterator.

(Eg) distance (ar.begin(), ar.end());

5) next() → ptr1 = next (ptr, 3); ⇒ returns iterator after advancing 3 positions (ie. 4th pos)

6) prev() → ptr2 = prev (ptr1, 3); ⇒ decrements 3 positions.

ALGORITHMS IN STL

* STL provides different types of algorithms that can be implemented upon any of the containers with the help of iterators.
* Use of STL algorithms saves time, effort; reliable & promotes code reusability.

(i) Sorting algorithm in STL

Syntax:

```
sort (start_iterator, end_iterator)
```

(Eg)

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void show (vector <int> v)
{
  vector <int> :: iterator it;
  for (it = v.begin(); it != v.end(); it++)
      cout << *it << "\t";
}
int main()
{
vector <int> v1 = {90, 70, 35, 60, 80};
cout << "Before sorting\n";
show (v1);
cout << "After sorting\n";
sort (v1.begin(), v1.end());
show (v1);
return 0;
}
```

o/p

```
Before sorting
90  70  35  60  80
After sorting
35  60  70  80  90
```

(ii) Searching algorithm in STL

(Eg) Syntax

```
binary_search (first, last, value)
```

- returns true if the value is present in given large
- i/p should be sorted before search fn is applied

(Eg)

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void main()
{
int n;
vector <int> vec = { 20, 40, 50 } ;
vector <int> :: iterator i;
cout << "The vector is \n";
for (i = vec.begin() ; i != vec.end(); i++)
    cout << *i << "\t";
cout << "Enter the element to search :\n";
cin >> n ;
if ( binary_search (vec.begin(), vec.end(), n))
    cout << "Element found \n";
else
   cout << "Element not found \n";
}
```

O/P :

```
The vector is
20 40 50
Enter the element to
search
40
Element found
```