ShreeSoft INFORMATICS

Standard Template Library in C++

Introduction to STL

Standard Template Library (STL) is a collection of standard C++ template classes. It consists of generic methods and classes to work with different forms of data. Standard Template Library is basically a generic library i.e. a single method/class can operate on different data types. So, as understood, we won’t have to declare and define the same methods/classes for different data types. Thus, STL saves a lot of effort, **reduces** the redundancy of the code and therefore, leads to the **increased** optimization of the code blocks.

Components of Standard Template Library

1. **Containers**
2. **Algorithms**
3. **Iterator**

Containers in Standard Template Library

In Standard Template Library, Containers create and store data and objects. Containers create generic data structures that can hold values of different data type’s altogether.
Following are the different types of Containers in STL:

1. **Sequence Containers**
2. **Associative Containers**
3. **Container Adapters**
4. **Unordered Associative Containers**

Sequence Containers in STL

Sequential Containers basically inculcate and implement the classes/functions containing the data structures wherein the elements can be accessed in a sequential manner.
The Sequential Containers consists of the following commonly used data structures:

- **array:** Static contiguous array (class template)
- **vector:** Dynamic contiguous array (class template)
- **deque:** Double-ended queue (class template)
- **forward_list:** Singly-linked list (class template)
- **list:** Doubly-linked list (class template)

Arrays

The Array class of Sequential Containers is much more efficient than the default array structure. Moreover, elements in an array are accessible in a sequential manner.
Syntax:

```
array<data_type, size>  array_name = {value1, value2, ...., valuen};
```

The Array class includes a huge number of functions to manipulate the data.

1. **front():** It is used to access and print the first element of the array.
2. **back():** It is used to access and print the last element of the array.
3. **empty():** Checks whether the array is empty or not.

4. **at():** Access and print the elements of the array.
5. **get():**This **function** is also used to access the elements of array. This function is not the member of array class but overloaded function from class tuple.
6. **size()**: It returns the number of elements in array. This is a property that C-style arrays lack.
7. **max_size()**: It returns the maximum number of elements array can hold i.e, the size with which array is declared. The size() and max_size() return the same value.
8. **swap():** Function to swap two input arrays.
9. **fill():** It fills the array with a specific/particular input element.

Example: To demonstrate the print or access array element in different way and size(), max_size().

```
#include<iostream>
#include<array>
using namespace std;
int main()
{
   int i;
   array<int,5> x = {10,20,30,40,50};

   cout<<"Print Array Elements using operator[]:";
   for(i=0;i<5;i++)
   {
         cout<<x[i]<<" ";
   }
   cout<<endl<<"Print Array Elements using at():";
   for(i=0;i<5;i++)
   {
         cout<<x.at(i)<<" ";
   }
   cout<<endl<<"Print Array Elements using get():";
   cout<<get<0>(x)<<" ";
   cout<<get<4>(x);

   cout<<endl<<"First Element:";
   cout<<x.front();
   cout<<endl<<"Last Element:";
   cout<<x.back();

   cout<<"The number of array elements is:";
   cout<<x.size();
   cout<<endl<<"Maximum Elements in Array:";
   cout<<x.max_size() << endl;
}
```

Example 2: Swapping elements of two array.

```
#include<iostream>
#include<array>
using namespace std;
```

```
int main()
{
   int i;
   array<int,5> arr1 = {10, 20, 30, 40, 50};
   array<int,5> arr2 = {11, 21, 31, 41, 51};

   cout<<"The first array elements before swapping are:";
   for(i=0;i<5;i++)
   cout<< arr1[i]<<" ";

   cout<<endl<<"The second array elements before swapping are:";
   for (int i=0;i<5;i++)
   cout << arr2[i] <<" ";

   // Swapping arr1 values with arr2
   arr1.swap(arr2);

   cout<<endl<<"The first array elements after swapping are:";
   for (i=0;i<5;i++)
   cout<<arr1[i]<<" ";
   cout<<endl<<"The second array elements after swapping are:";
   for (i=0;i<5;i++)
   cout<<arr2[i]<<" ";
}
```

Example 3: To demonstrate the empty() and fill()

```
#include<iostream>
#include<array>
using namespace std;
int main()
{
   int i;
   array<int,5> arr1={10,20,30,40,50};
   array<int,0> arr2;

   // To check first array is empty
   if(arr1.empty())
      cout<<endl<<"Array empty";
   else
      cout<<endl<<"Array not empty";

   // To check second array is empty
   if(arr2.empty())
      cout<<endl<<"Array empty";
   else
      cout<<endl<<"Array not empty";
   arr1.fill(0);
   cout<<endl<<"Array after filling operation is : ";
   for (i=0;i<5;i++)
      cout<<arr1[i]<<" ";
}
```

Vector

Vectors in C++ function the same way as Arrays in a dynamic manner i.e. vectors can resize itself automatically whenever an item is added/deleted from it.
The data elements in Vectors are placed in contagious memory locations and Iterator can be easily used to access those elements. Moreover, insertion of items in takes place at the end of Vector.
Syntax:

vector<data_type> vector_name;

Most commonly used functions of Vector:

1. **begin():** It returns an iterator element which points to the first element of the vector.
2. **end():** It returns an iterator element which points to the last element of the vector.
3. **rbegin():** Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. **rend():** Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. **cbegin():** Returns a constant iterator pointing to the first element in the vector.
6. **cend():** Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. **crbegin():** Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
8. **crend():** Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
9. **push_back():** It inserts the element into the vector from the end.
10. **pop_back():** It deletes the element from the end of the vector.
11. **reference operator [g]:** Returns a reference to the element at position 'g' in the vector
12. **at(g):** Returns a reference to the element at position 'g' in the vector
13. **front():** Returns a reference to the first element in the vector
14. **back():** Returns a reference to the last element in the vector
15. **data():** Returns a direct pointer to the memory array used internally by the vector to store its owned elements.
16. **size():** This function gives the size i.e. the number of elements in the vector.
17. **max_size():** Returns the maximum number of elements that the vector can hold.
18. **capacity():** Returns the size of the storage space currently allocated to the vector expressed as number of elements.
19. **resize(n):** Resizes the container so that it contains 'n' elements.
20. **empty():** Checks whether the vector is empty or not.
21. **front():** It returns the first element of the vector.
22. **back():** It returns the last element of the vector.
23. **assign():** It assigns new value to the vector elements by replacing old ones
24. **insert():** This function adds the element before the element at the given location/position.

25. **erase():** It is used to remove elements from a container from the specified position or range.
26. **clear():** It is used to remove all the elements of the vector container.
27. **emplace():** It extends the container by inserting new element at position.
28. **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
29. **swap():** Swaps the two input vectors.
30. **shrink_to_fit():** Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
31. **reserve():** Requests that the vector capacity be at least enough to contain n elements.

Example 1: To print vector in forward and backward direction

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
        int i;
        vector<int> v1;
        cout<<endl<<"Print vector using push_back():";
        for (i=1; i<=5; i++) {
           v1.push_back(i);
        }
        cout<<endl<<"Print vector using begin and end: ";
        for(auto i=v1.begin(); i!=v1.end(); i++) {
           cout<<*i<<" ";
        }
        cout << "\nPrint vector cbegin and cend: ";
        for (auto i=v1.cbegin(); i!=v1.cend(); i++)
                cout << *i << " ";
        cout << "\nOutput of rbegin and rend: ";
        for (auto i=v1.rbegin(); i!=v1.rend(); i++)
                cout << *i << " ";
        cout << "\nOutput of crbegin and crend : ";
        for (auto i=v1.crbegin(); i!=v1.crend(); i++)
                cout << *i << " ";
}
```

Example 2: To show the use of size(), max-size(), capacity(), empty().

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
        int i;
        vector<int> v1;
        for (i = 1; i <= 5; i++) {
            v1.push_back(i);
        }
        cout<<endl<<"Size:"<<v1.size();
        cout<<endl<<"Capacity:"<<v1.capacity();
        cout<<endl<<"Max_Size:"<<v1.max_size();

        // resizes the vector size to 3
        v1.resize(3);
        cout<<endl<<"Prints the vector size after resize():"<< v1.size();
        // checks if the vector is empty or not
        if (v1.empty() == false)
                cout<<endl<<"Vector is not empty";
        else
                cout<<endl<<"Vector is empty";

        // Shrinks the vector
        v1.shrink_to_fit();
        cout<<endl<<"Vector elements are: ";
        for (auto i=v1.begin(); i!=v1.end(); i++)
                cout<<*i<<" ";
}
```

Example 2: To show the use of element accessing functions.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
        int i;
        vector<int> v1;
        for (i=1; i<=10; i++)
                v1.push_back(i * 10);
        cout<<endl<<"Reference operator[v]:v1[2]="<<v1[2];
        cout<<endl<<"at:v1.at(4)="<<v1.at(4);

        cout<<endl<<"First Element="<<v1.front();
        cout<<endl<<"Last Element="<<v1.back();

        // pointer to the first element
        int* pos=v1.data();
        cout<<endl<<"First element using pointer:"<<*pos;
}
```

Example 3: To show the use of Modifiers in vector.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   int i;
         // Assign vector
         vector<int> v1;
         // fill the vector 7 times with value 13
         v1.assign(7, 13);
         cout<<endl<<"The vector elements are:";
         for (i=0; i<v1.size(); i++) {
            cout<<v1[i]<<" ";
         }
         // inserts 15 to the last position
         v1.push_back(15);
         int n = v1.size();
         cout<<endl<<"The last element:"<<v1[n - 1];
         // removes last element
         v1.pop_back();
         // prints the vector
         cout<<endl<<"The vector elements are: ";
         for(i=0; i<v1.size(); i++) {
            cout<<v1[i] << " ";
         }
         // inserts 5 at the beginning
         v1.insert(v1.begin(), 5);
         cout<<endl<<"The first element is:"<<v1[0];
         // removes the first element
         v1.erase(v1.begin());
         cout<<endl<<"The first element is: " << v1[0];
         // inserts at the beginning
         v1.emplace(v1.begin(), 5);
         cout<<endl<<"The first element is:"<<v1[0];
         // Inserts 20 at the end
         v1.emplace_back(20);
         n = v1.size();
         cout<<endl<<"The last element is:"<<v1[n - 1];

         // erases the vector
         v1.clear();
         cout << "\nVector size after erase(): " << v1.size();
         // two vector to perform swap
         vector<int> v2, v3;
         v2.push_back(11);
```

```
        v2.push_back(22);
        v3.push_back(33);
        v3.push_back(44);

        cout<<endl<<"Before Swap Vector 2:";
        for(i=0; i<v1.size(); i++) {
           cout<<endl<<v2[i]<<" ";
        }
        cout<<endl<<"Before Vector 3:";
        for(i=0; i<v3.size(); i++) {
           cout<<v3[i]<<" ";
        }
        // Swaps v1 and v2
        v1.swap(v2);
        cout<<endl<<"After Swap Vector 2:";
        for(i=0; i<v2.size(); i++) {
           cout << v2[i] << " ";
        }
        cout<<endl<<"After Swap Vector 3:";
        for (i=0; i<v3.size(); i++) {
           cout << v3[i] << " ";
        }
}
```

Deque:

Double-ended queues are sequence containers with the feature of expansion and contraction on both ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.
Double Ended Queues are basically an implementation of the data structure double-ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

Example: Program to implement Deque in STL

```
#include <deque>
#include <iostream>
using namespace std;
void showdq(deque<int> d)
{
        //deque<int>::iterator it;
        for (auto it = d.begin(); it != d.end(); it++)
                cout<<'\t'<< *it;
        cout<<endl;
}
```

```
int main()
{
	deque<int> d1;
	d1.push_back(10);
	d1.push_front(20);
	d1.push_back(30);
	d1.push_front(15);
	cout<<endl<<"The deque is:";
	showdq(d1);

	cout<<endl<<"Deque size:"<<d1.size();
	cout<<endl<<"Deque Maximum size:"<<d1.max_size();

	cout<<endl<<"Deque value at location 2:"<<d1.at(2);
	cout<<endl<<"First Element of Deque:" <<d1.front();
	cout<<endl<<"Last Element of Deque:" <<d1.back();

	cout<<endl<<"After Delete First Element:";
	d1.pop_front();
	showdq(d1);
	cout<<endl<<"After Delete Last Element:";
	d1.pop_back();
	showdq(d1);
}
```

forward_list: The *forward_list* is implemented as a singly linked-list where each node contains a data element and a reference pointer to point to the next element in the list. Due to its forward moving structure, each keeps information about its next element and hence traversal can be done in forward direction only. The container is useful for algorithms that require insertion, deletion, and forward iteration of elements. Because it is implemented with linked-list, it is completely dynamic in nature and the size of the list is immaterial.

Functions of forward_list:

Function Name	Usage or Description
assign()	This function is used to assign values to the forward list, its other variant is used to assign repeated elements.
push_front()	This function is used to insert the element at the first position on forward list. The value from this function is copied to the space before first element in the container. The size of forward list increases by 1.
emplace_front()	This function is similar to the previous function but in this no copying operation occurs, the element is created directly at the memory before the first element of the forward list.
pop_front()	This function is used to delete the first element of the list.
insert_after()	This function gives us a choice to insert elements at any position in forward list. The arguments in this function are copied at the desired position.
emplace_after()	This function also does the same operation as the above function but the elements are directly made without any copy operation.

erase_after()	This function is used to erase elements from a particular position in the forward list.
remove()	This function removes the particular element from the forward list mentioned in its argument.
remove_if()	This function removes according to the condition in its argument.
splice_after()	This function transfers elements from one forward list to other.
front()	This function is used to reference the first element of the forward list container.
begin()	This function is used to return an iterator pointing to the first element of the forward list container.
end()	This function is used to return an iterator pointing to the last element of the list container.
cbegin()	Returns a constant iterator pointing to the first element of the forward_list.
cend()	Returns a constant iterator pointing to the past-the-last element of the forward_list.
before_begin()	Returns an iterator that points to the position before the first element of the forward_list.
cbefore_begin()	Returns a constant random access iterator which points to the position before the first element of the forward_list.
max_size()	Returns the maximum number of elements that can be held by forward_list.
resize()	Changes the size of forward_list.
unique()	Removes all consecutive duplicate elements from the forward_list. It uses a binary predicate for comparison.
reverse()	Reverses the order of the elements present in the forward_list.

Example 1: To assign value to forward list and print forward list.

```
#include <iostream>
#include <forward_list>
using namespace std;
int main()
{
   forward_list<int> flist1;
   forward_list<int> flist2;
   flist1.assign({ 1, 2, 3 });
   //assigning value
   flist2.assign(7, 13);
   cout << "The elements of first list are : ";
   for(auto it=flist1.begin(); it!= flist1.end(); it++)
   cout << *it << " ";
   cout<<endl<<"The elements of second list are : ";
   for(auto it=flist2.begin(); it!= flist2.end(); it++)
   cout << *it << " ";
}
```

Example 2: To store value in to forward list from user and use of function emplace_front(), pop_front() and print forward list.

```
#include <iostream>
#include <forward_list>
using namespace std;
int main()
{
        int k,n,i;
        forward_list<int> flist;
        cout<<endl<<"Enter how many numbers to be read:";
        cin>>n;
        for(i=1;i<=n;i++) {
           cout<<endl<<"Enter Number:";
           cin>>k;
           flist.push_front(k);
        }
        // print the forward list
        cout<<endl<<"The forward list after push_front operation:";
        for(auto it=flist.begin();it!=flist.end(); ++it)
                cout<<*it<< " ";
        // Inserting value using emplace_front()
        flist.emplace_front(90);
        cout<<endl<<"The forward list after emplace_front operation:";
        for(auto it=flist.begin();it!=flist.end(); ++it)
                cout<<*it<< " ";
        // Deleting first value using pop_front()
        flist.pop_front();
        cout<<endl<<"The forward list after pop_front operation:";
        for(auto it=flist.begin();it!=flist.end(); ++it)
                cout<<*it<< " ";
}
```

Algorithms in Standard Template Library

Standard Template Library provides us with different **generic algorithms**. These algorithms contain built in generic functions which can be directly accessed in the program. The Algorithm and its functions can be accessed with the help of Iterators only.

Types of Algorithms offered by Standard Template Library:

1. Sorting Algorithms
2. Search algorithms
3. Non-modifying algorithms
4. Modifying algorithms
5. Numeric algorithms
6. Minimum and Maximum operations

Iterators in Standard Template Library

Iterators are basically used to point or refer to a particular memory location of the element. They point at the containers and help them manipulate the elements in an efficient manner. Following are some of the commonly used functions offered by Iterators in Standard Template Library:

1. **iterator.begin():** It returns the reference to the first element of the container.
2. **iterator.end():** It returns the reference to the last element of the container.
3. **iterator.prev(iterator_name, index):** It returns the position of the new iterator that would point to the element before the specified index value in the parameter.
4. **iterator.next(iterator_name, index):** It returns the position of the new iterator that would point to the element after the specified index value in the parameter.
5. **iterator.advance(index):** It increments the iterator’s position to the specified index value.