

# Standard Template Library

Ali Haider

[syedalihaider.ciit@gmail.com](mailto:syedalihaider.ciit@gmail.com)

Department of Computer Science IUB

# Overview

- Standard Template Library
- Containers
- Algorithms
- Iterators

# Standard Template Library

- 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.
- It is a library of container classes, algorithms, and iterators.
- It is a generalized library and so, its components are parameterized.
- A working knowledge of template classes is a prerequisite for working with STL.

# The three most important Entities of STL

- **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** act on containers.
- They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
- **Iterators** are used to step through the elements of collections of objects.
- These collections may be containers or subsets of containers.

# Containers-1

- Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.
- **Sequence Containers:** implement data structures which can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward\_list( Introduced in C++11)
- **Container Adaptors :** provide a different interface for sequential containers.
  - queue
  - priority\_queue
  - stack

# Containers-2

- **Associative Containers** : implement sorted data structures that can be quickly searched ( $O(\log n)$  complexity).
  - set
  - multiset
  - map
  - multimap
- **Unordered Associative Containers** : implement unordered data structures that can be quickly searched
  - unordered\_set (Introduced in C++11)
  - unordered\_multiset (Introduced in C++11)
  - unordered\_map (Introduced in C++11)
  - unordered\_multimap (Introduced in C++11)

# Algorithms

- The header algorithm defines a collection of functions especially designed to be used on ranges of elements.
- They act on containers and provide means for various operations for the contents of the containers.
- Algorithm
- Sorting
- Searching
- Important STL Algorithms
- Useful Array algorithms
- Partition Operations
- Numeric
- valarray class

# Sort Algorithm

- Sorting is one of the most basic functions applied to data.
- It means arranging the data in a particular fashion, which can be increasing or decreasing.
- There is a built-in function in C++ STL by the name of `sort()`.
- **`sort(startaddress, endaddress)`**
- `startaddress`: the address of the first element of the array
- `endaddress`: the address of the next contiguous location of the last element of the array.
- So actually `sort()` sorts in the range of `[startaddress, endaddress)`



# Example

```
#include <iostream>
#include <algorithm>
using namespace std;
void show(int a[]) {
    for(int i = 0; i < 10; ++i)
        cout << a[i] << " ";
}
int main() {
    int a[10]= {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    cout << "\n The array before sorting is : ";
    show(a);

    sort(a, a+10);

    cout << "\n\n The array after sorting is : ";
    show(a);

    return 0;
}
```

# Iterators

- Iterators are used for working upon a sequence of values.
- They are the major feature that allow generality in STL.
- Iterators are used to point at the memory addresses of [STL](#) containers.
- They are primarily used in sequence of numbers, characters etc.
- They reduce the complexity and execution time of program.

## Operations of iterators :-

- **1. begin()** :- This function is used to return the **beginning position** of the container.
- **2. end()** :- This function is used to return the ***after* end position** of the container.

## Example

```
// C++ code to demonstrate the working of
// iterator, begin() and end()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
    vector<int> ar={1, 2, 3, 4, 5}; |
    // Declaring iterator to a vector
    vector<int>::iterator ptr;
    // Displaying vector elements using begin() and end()
    cout << "The vector elements are : ";
    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
    {
        cout << *ptr << " ";
    }
    return 0;
}
```

# Operations of iterators

- **3. advance()** :- This function is used to **increment the iterator position** till the specified number mentioned in its arguments.
- **4. next()** :- This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.
- **5. prev()** :- This function **returns the new iterator** that the iterator would point **after decrementing the positions** mentioned in its arguments.
- **6. inserter()** :- This function is used to **insert the elements at any position** in the container.
- It accepts 2 arguments, the container and iterator to position where the elements have to be inserted.

# Example of advance()

```
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main() {
    vector<int> ar = { 1, 2, 3, 4, 5 };
    // Declaring iterator to a vector
    vector<int>::iterator ptr = ar.begin();
    // Using advance() to increment iterator position
    // points to 4
    advance(ptr, 3);
    // Displaying iterator position
    cout << "The position of iterator after advancing is : ";
    cout << *ptr << " ";
    return 0; |
}
```

# Example of next() and prev()

```
#include<iostream> |
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main() {
    vector<int> ar = { 1, 2, 3, 4, 5 };
    // Declaring iterators to a vector
    vector<int>::iterator ptr = ar.begin();
    vector<int>::iterator ftr = ar.end();
    // Using next() to return new iterator
    // points to 4
    auto it = next(ptr, 3);
    // Using prev() to return new iterator
    // points to 3
    auto it1 = prev(ftr, 3);
    // Displaying iterator position
    cout << "The position of new iterator using next() is : ";
    cout << *it << " " << endl;
    // Displaying iterator position
    cout << "The position of new iterator using prev() is : ";
    cout << *it1 << " " << endl;
    return 0;
}
```

# Example of inserter()

```
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int> ar1 = {10, 20, 30};
    // Declaring iterator to a vector
    vector<int>::iterator ptr = ar.begin();
    // Using advance to set position
    advance(ptr, 3);
    // copying 1 vector elements in other using inserter()
    // inserts ar1 after 3rd position in ar
    copy(ar1.begin(), ar1.end(), inserter(ar,ptr));
    // Displaying new vector elements
    cout << "The new vector after inserting elements is : ";
    for (int &x : ar)
        cout << x << " ";
    return 0;
}
```