A Brief Introduction to C++ STL

Standard Template Library (STL)

- **STL** is a software library for the C++ programming language.
- It provides four components called:
  - *algorithms*: The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.
  - *containers*: pair, list, vector, map, etc.
  - iterators: are objects that traverse inside a container
  - *function objects or functors*:
    - The header <functional> is required.
    - This is a object that can be used with **()** in the manner of a function.

Examples of STL Containers

Simple Examples of STL Container Objects

```
#include<iostream>
#include<utility>
#include<iterator>
#include<vector>
#include<array>
#include <map>
#include <string>
using namespace std;
int main(){
  array<int,5> a = {1, 2, 3, 6};
  a.back() = 9;
  vector<int> ar = {1, 2, 3,4 , 5};
  ar.push_back(34);
    map<string, int> mp;
  mp.insert(pair<string,int> ("abc", 10001));
  mp.insert(pair<string,int> ("abcd",10002));
    ...
    return 0;
}
```

An Example for Using STL Algorithms

Simple Example for STL algorithm

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace stud;
int main(){
    vector<int> v;
    v.push_back(4);;
    v.push_back(-1);
    v.push_back(3);
    v.push_back(0);
    v.push_back(2);
    sort(v.begin(), v.end());
    reverse(v.begin(), v.end());
    for (size_t i = 0; i < v.size(); ++i)
        cout << v.at(i) << " ”;
    return 0;
}
```

What is Iterator?

Iterator

- An iterator is an object that provides a general method of successively accessing each element of a container type such as vectors or lists. In other words it's an object that enables a programmer to travers linear containers such as arrays, vectors and lists.

Simple Example of Using STL Iterator for array

```
int main(){
  array<int, 5> a = {1, 2, 3, 7};
  a.back() = 9;
    array<int>::iterator itr2;

    // Displaying array elements
    cout << "The array elements are : \n";

    for (itr2 = a.begin(); itr2 < a.end(); itr2++)
    cout << *itr2  << "  ";

    return 0;
}
```

Simple Example of Using STL Iterator for vector

```
int main(){
    ...
    ...
    vector<int> ar = { 1, 2, 3, 4, 5 };
    ar.push_back(34);
    vector<int>::iterator itr;

    // Displaying vector elements
    cout << "The vector elements are : \n";
    for (itr = ar.begin(); itr < ar.end(); itr++)
        cout << *itr << ";

        return 0;
}
```

Simple Example of Using STL Iterator for map

```
int main(){
   ...
   ...
  map<string, int> mp;
   mp.insert(pair<string, int> ("abc", 10001));
   mp.insert(pair<string, int> ("abcd", 10002));
   // Displaying map elements
   map<string, int>::iterator ptr3;
    cout << "The map elements are : \n";
   for (ptr3 = mp.begin(); ptr3 != mp.end(); ptr3++)
      cout << ptr3->first << " " << ptr3->second << endl;

     return 0;
}
```

How does iterator work.

How Does Iterator Work

- An iterator class is normally a class with a pointer that points to the data within a container class.
- C++ operators such as ++ and – can be used to move the pointer forward/backward and retrieve the data within the container.

```
template <class T>
class X {
 friend Iterator<T>
 private:
     …
   public:
            …
 };
```

```
template<class T>
class Iterator {
private:
     X <T>* ptr;
public:
     Iterator(X& arg): ptr(&arg){
          …
     }
          …
 };
```

An Example

- Let's write a C++ class iterator called ***ArrayIterator***, that provides overloaded operators to allow access to the elements of a container class called **Array**.
- Assuming we have an object of ***ArrayIterator*** called ***iter*** that initially is associated with the first element of an object of class ***Array***, then a statement such as:

```
cout  << iter++;
```

Is supposed to display the value of the first element of the array, then the pointer within the iterator object advances to the next element of the array. Obviously it means that we need to overload operator ++ (postfix) in the class ArrayIterator.

- Similarly, you can overload other operators to achieve different type of operations on a data within the container class.

One Possible Solution

```
template <class T> class Array;
template<class T>
class ArrayIterator;
template <class T>
class Array {
   friend ArrayIterator<T>;
   T *storage;
   int size;
public:
   Array(int s): size(s){
      storage = new T[size];
      //...
   }
   T& operator [] (int i) {
      return storage[i];
    }
};
```

```
template<class T>
class ArrayIterator {
   Array<T>* ptr;
   int index;
public:
   ArrayIterator(Array<T>& ar):ptr(&ar), index(0){}
   T operator++();
   T operator--();
};
```

Possible solution continued

```
// Prefix ++

templae <class T>
T ArrayIterator<T>::operator++()
{
    index++;
    if(index >= ptr->size)
        index = 0;
    return ptr->storage[index];
}
```

```
int main(){
    cout << "Hello World ..." << endl;
    Array <int> x(3);
    x[0] = 100;
    x[1] = 200;
    x[2] = 110;
   ArrayIterator<int> itr(x);
   cout << x[0] << endl;
   cout << ++itr << endl;
   return 0;
}
```