Preview

- What is STL?
- Iterator and Containers
- Vector Container
- Vector Container Members

What is the Standard Template Library

- The Standard Template Library (STL) was developed by Alexander Stepanov and Meng Lee (1995) at HP.
- The **STL** is a fundamental part of the C++ which is defined as standard in the 1997.
- The **STL** is a collection C++ libraries that allow you to use several well known kinds of data structures (Stack, Queue, Linked List, Vector,…)without having to program them.

What is the Standard Template Library

Library	Description
<**vector**>	A dynamic array
<**list**>	A randomly changing sequence of items
<**stack**>	A sequence of items with **pop** and **push** at one end only
<**queue**>	A Sequence of items with **pop** and **push** at opposite ends
<**deque**>	Double Ended Queue with pop and push at both ends
<**bitset**>	A subset of a fixed and small set of items
<**set**>	An unordered collection of items
<**map**>	An collection of pairs of items indexed by the first one

What is the Standard Template Library

- The STL has been adopted as a standard by the ISO/IEC and ANSI.
- However current implementation of the STL are not totally potable.
- The concept of STL come from reusability
- Using STL, can save time and effort to develop a software.
- There are components of STL:
  - Containers
  - Iterators

Iterators and Containers

Containers

- A Container is a data structure that holds a number of object of the same type or class.
- Ex) **Lists**, **Vectors**, **Stacks**, **Queues**, etc are all **Containers**
- STL has been carefully designed so that each containers provides space for data

Iterators and Containers

Iterators

- Items in **Containers** are referred to be special objects called: *iterator*s.
- **iterators** are generalization of pointers.

Vector Container

- Vectors are a kind of **sequence containers** - like regular array, their elements stored in contiguous storage locations (that means elements can be accessed not only using iterators but also using offsets on regular pointers
- Vector containers are implemented as **dynamic arrays** – unlike regular array, size of vector automatically being expanded and contracted

Vector Container

- Vector provide almost the same performance as array plus have ability to easily resized.
- Vector usually consume more memory than arrays in order to accommodate for extra storage space for future growth

Vector Container Member Function

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Copy vector content (public member function)
Iterators:	
begin	Return iterator to beginning (public member type)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
Capacity:	
size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)

Vector Container Member Function

Element access:	
operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
Modifiers:	
assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
Allocator:	
get_allocator	Get allocator (public member function)


```
// vector.cpp Testing Standard Library vector class template
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   const int SIZE = 100;
   vector< int > v, w;

   cout << "The initial size of v is: " << v.size() <<endl;
   cout << "The initial capacity of v is: " << v.capacity()<<endl;

   for (int i = 1; i <= SIZE; i++)
   {
            v.push_back(i);
            cout <<"Now, the capacity of v becomes " << v.capacity() <<endl;
            cout <<"Now, the size of v becomes " << v.size()<<endl <<endl;
   }
          // using iterator for display element of vector
   cout << "Contents of vector v using iterator notation: ";
   vector <int>::iterator p3;
   for (p3=v.begin(); p3 != v.end(); p3++)
        cout << *p3 << ' ';
   cout <<endl <<endl;
   // using reversed iterator for display element of vector
   cout << "Reversed contents of vector v using reverse_iterator notation: ";
   vector< int >::reverse_iterator p2;
   for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
        cout << *p2 << ' ';
   cout <<endl;

   w = v; // use overloaded operator in vector STL;
   cout << "Contents of vector w using iterator notation: ";
   for (p3=w.begin(); p3 != w.end(); p3++)
           cout << *p3 << ' ';

   cout << endl;
   return 0;
}
```



```
// vector1.cpp Testing Standard Library vector class template
#include <iostream>
#include <vector>
using namespace std;

template < class T >
void printVec(vector< T > &vec );

template < class T >
void printVecRev(vector< T > &vec );

int main()
{
   const int SIZE = 100;
   vector< int > v, w;

   cout << "The initial size of v is: " << v.size() <<endl;
   cout << "The initial capacity of v is: " << v.capacity()<<endl;

   for (int i = 1; i <= SIZE; i++)
   {
            v.push_back(i);
            cout <<"Now, the capacity of v becomes " << v.capacity() <<endl;
            cout <<"Now, the size of v becomes " << v.size()<<endl <<endl;
   }
          // using iterator for display element of vector
   cout << "Contents of vector v using iterator notation: ";
   printVec(v);
          // using reversed iterator for display element of vector
   cout << "Reversed contents of vector v using reverse_iterator notation: ";
   printVecRev(v);

   w = v; // use overloaded operator in vector STL;
   cout << "Contents of vector w using iterator notation: ";
   printVec(w);
   return 0;
}
```

Vector Container Member Function

```
template < class T >
void printVec(vector< T > &vec )
{
   vector <int>::iterator p;
   for ( p = vec.begin(); p != vec.end(); p++ )
      cout << *p << ' ';
   cout <<endl;
}

template < class T >
void printVecRev(vector< T > &v )
{
  vector< T >::reverse_iterator p2;

   for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
      cout << *p2 << ' ';
   cout << endl;
}
```


Vector Container Member Function

```
// vector2.cpp comparing size, capacity and max_size
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
  vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  cout << "size: " << (int) myvector.size() << endl;
  cout << "capacity: " << (int) myvector.capacity() << endl;
  cout << "max_size: " << (int) myvector.max_size() << endl;
  return 0;
}
```



```
// vector3.cpp vector container member functions resize(), push_back()
#include <iostream>
#include <vector>
using namespace std;

template <class T>
void printVec(vector <T> &);

int main ()
{
   vector<int> myvector;

  // set some initial content:
  for (int i=1;i<10;i++)
             myvector.push_back(i); // 1, 2, 3, 4, 5, 6, 7, 8, 9

  printVec(myvector);
  myvector.resize(5); // 1, 2, 3, 4, 5
  printVec(myvector);

  myvector.resize(8,100); // 1, 2, 3, 4, 5, 100, 100, 100
  printVec(myvector);

  myvector.resize(12); // 1, 2, 3, 4, 5, 100, 100, 100, 0, 0, 0, 0
  printVec(myvector);

  return 0;
}
// function for print vector
template <class T>
void printVec(vector <T> &v)
{
          for (int i=0; i<v.size(); i++)
            cout << " " <<v[i];
          cout <<endl;
}
```


Vector Container Member Function

```
// vector4.cpp front(): access front
//             back(): access back
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
  vector<int> myvector;

  myvector.push_back(77); // 77
  myvector.push_back(16); // 77, 16


  myvector.front() -= myvector.back(); // 61, 16

  cout << "myvector.front() is now " << myvector.front() << endl;

  return 0;
}
```



```
// vector5.cpp insert() inserting into a vector
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
  vector<int> myvector (3,100); //create myvector with initial values 100 100 100
  vector<int>::iterator it;

  it = myvector.begin();
  it = myvector.insert ( it , 200 ); // now  myvector is 200 100 100 100

  myvector.insert (it,2,300); // now vecot is 300 300 200 100 100 100

  // "it" no longer valid, get a new one:
  it = myvector.begin();

  vector<int> anothervector (2,400); // new vecor with 400 400
  myvector.insert (it+2,anothervector.begin(),anothervector.end());
  // now myvector is 300 300 400 400 200 100 100 100
  int myarray [] = { 501,502,503 };
  myvector.insert (myvector.begin(), myarray, myarray+3);
  // now myvector is 501 502 503 300 300 400 400 200 100 100 100
  cout << "myvector contains:";
  for (it=myvector.begin(); it<myvector.end(); it++)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```


Vector for Structured Data Type

```
// vector6.cpp vector container member functions with structured data type
#include <iostream>
#include <vector>
using namespace std;
struct Student {
          char LastName[20];                //last name
          char FirstName[20];               // First Name
          int IDNumber;                     // Student ID
          Student();                        // Constructor
          bool operator == (const Student);
          bool operator > (const Student);
          bool operator < (const Student);
          friend ostream& operator << (ostream &stream, const Student &student);
};
// Student Constructor
Student::Student()
{
          cout << "What is the student's Last Name: ";
          cin >> LastName;
          cout << "What is the student's First Name: ";
          cin >> FirstName;
          cout << "What is the student's ID#: ";
          cin >> IDNumber;
}
```

```
// == Operator Overloader: checks if the numbers are equal
bool Student::operator == (Student x)
{
            return (x.IDNumber == IDNumber);
}
// > Operator Overloader: checks if the number is greater than
bool Student::operator > (Student x)
{
            return (IDNumber > x.IDNumber);
}
// < Operator Overloader: checks if the number is less than
bool Student::operator < (Student x)
{
            return (IDNumber < x.IDNumber);
}
// << Operator Overloader: displays the structures information
ostream& operator << (ostream &stream, const Student &student)
{
            stream << "ID#: " << student.IDNumber <<" - " << student.LastName << ", " <<
student.FirstName << endl;
            return(stream);
}
int main ()
{
    // vector for Student structure
            vector<Student> StdVec;

  for (int i= 1; i <3; i++)
  {
             Student *ptr;
             ptr = new Student;
             StdVec.push_back(*ptr);
  }

  for (int i =1; i < 3; i++)
             cout << StdVec[i-1];


}
```