Week 8-2 : C++ Templates and Standard Library

2. STL (Standard Template Library)

■ Collection of Template which include often used data structure and algorithm

■ Container(contain several different data structure), iterator(access the elements), Algorithm(composed of algorithm classes which operate jobs at container)

Containers

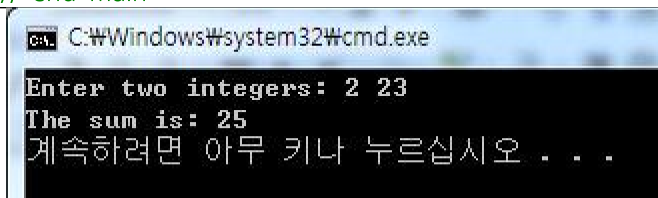
Standard Library container class	Description
Sequence containers	
`vector`	rapid insertions and deletions at back direct access to any element
`deque`	rapid insertions and deletions at front or back direct access to any element
`list`	doubly linked list, rapid insertion and deletion anywhere
Associative containers	
`set`	rapid lookup, no duplicates allowed
`multiset`	rapid lookup, duplicates allowed
`map`	one-to-one mapping, no duplicates allowed, rapid key-based lookup
`multimap`	one-to-many mapping, duplicates allowed, rapid key-based lookup
Container adapters	
`stack`	last-in, first-out (LIFO)
`queue`	first-in, first-out (FIFO)
`priority_queue`	highest-priority element is always the first element out

Common member functions for all STL containers	Description
default constructor	A constructor to provide a default initialization of the container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.
`empty`	Returns `true` if there are no elements in the container; otherwise, returns `false`.
`insert`	Inserts an item in the container.
`size`	Returns the number of elements currently in the container.
`operator=`	Assigns one container to another.
`operator<`	Returns `true` if the first container is less than the second container; otherwise, returns `false`.
`operator<=`	Returns `true` if the first container is less than or equal to the second container; otherwise, returns `false`.
`operator>`	Returns `true` if the first container is greater than the second container; otherwise, returns `false`.
`operator>=`	Returns `true` if the first container is greater than or equal to the second container; otherwise, returns `false`.
`operator==`	Returns `true` if the first container is equal to the second container; otherwise, returns `false`.
`operator!=`	Returns `true` if the first container is not equal to the second container; otherwise, returns `false`.
`swap`	Swaps the elements of two containers.

Common member functions for all STL containers	Description
Functions found only in first-class containers	
`max_size`	Returns the maximum number of elements for a container.
`begin`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the first element of the container.
`end`	The two versions of this function return either an `iterator` or a `const_iterator` that refers to the next position after the end of the container.
`rbegin`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the last element of the container.
`rend`	The two versions of this function return either a `reverse_iterator` or a `const_reverse_iterator` that refers to the next position after the last element of the reversed container.
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

- Iterators

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iterator> // ostream_iterator and istream_iterator

int main()
{
	cout << "Enter two integers: ";

	//  create istream_iterator for reading int values from cin
	std::istream_iterator< int > inputInt( cin );

	int  num
ber1 = *inputInt; // read int from standard  input
	++inputInt; // move iterator to next input value
	int  number2 = *inputInt; // read int from standard  input

	//  create ostream_iterator for writing int values to cout
	std::ostream_iterator< int > outputInt( cout );

	cout << "The sum is: ";
	*outputInt = number1 +  number2; // output result to cout
	cout << endl;
	return  0;
} // end main
```



Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

-Iterator Opration

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.

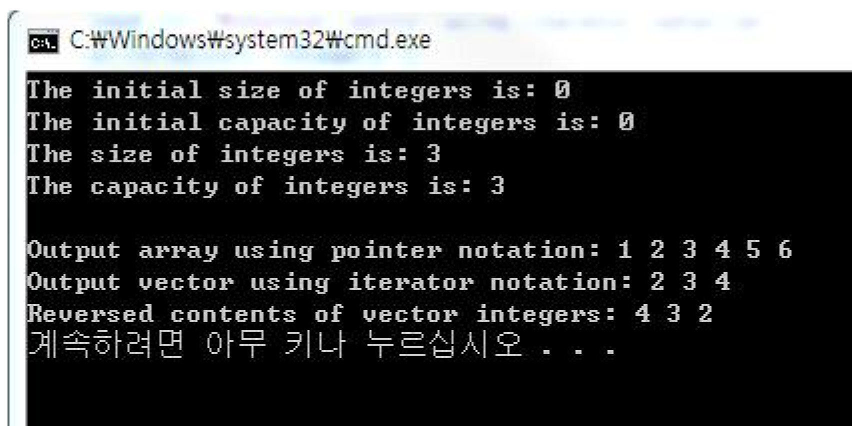
Iterator operation	Description
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` or `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

-Example

```
#include <iostream>

using std::cout;

using std::endl;


#include <vector> // vector class-template definition

using std::vector;


// prototype for function template printVector

template < typename T > void printVector( const   vector< T > &integers2 );


int main(){

        const   int SIZE = 6; //   define array size

        int   array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
```

```
vector< int > integers; //   create vector of ints

cout << "The initial size of integers is: "   << integers.size()
<< "₩nThe initial capacity of integers is: "   << integers.capacity();

//   function push_back is in every sequence collection
integers.push_back( 2   );
integers.push_back( 3   );
integers.push_back( 4   );

cout << "₩nThe size of integers is: " <<   integers.size()
<< "₩nThe capacity of integers is: "   << integers.capacity();
cout << "₩n₩nOutput array using pointer notation: ";

//   display array using pointer notation
for   ( int *ptr = array;   ptr != array + SIZE; ptr++ )
cout << *ptr   << ' ';

cout << "₩nOutput vector using iterator notation: ";
printVector( integers );
cout << "₩nReversed contents of vector integers: ";

//   two const reverse iterators
vector< int >::const_reverse_iterator reverseIterator;
vector< int >::const_reverse_iterator tempIterator =   integers.rend();

//   display vector in reverse order using reverse_iterator
for   ( reverseIterator = integers.rbegin();
reverseIterator!=   tempIterator; ++reverseIterator )
cout <<   *reverseIterator << ' ';
```

```
        cout << endl;
        return   0;
} // end main
// function template for outputting vector   elements
template < typename T > void printVector( const   vector< T > &integers2 )
{
        typename   vector< T >::const_iterator constIterator; //   const_iterator

        //   display vector elements using const_iterator
        for   ( constIterator = integers2.begin();
        constIterator !=   integers2.end(); ++constIterator )
        cout <<   *constIterator << ' ';
} // end function printVector
```



[Exercise]

- See the final print out value and complete the Stack.h and main.cpp
- Generate object based on Class Template

  - Stack.h

```
#ifndef STACK_H
#define STACK_H

template< typename T >
class Stack
{
private:
        int   size; // # of elements in the Stack
        int   top; // location of the top element (-1 means   empty)
        T *stackPtr; // pointer to internal representation of the Stack

public:
        Stack( int = 10 ); // default   constructor (Stack size 10)

        //   destructor
        ~Stack()
        {
        delete [] stackPtr; //   deallocate internal space for Stack
        } //   end ~Stack destructor

        bool   push( const T& ); // push an element onto the Stack
        bool   pop( T& ); // pop an element off the Stack

        //   determine whether Stack is empty
        bool   isEmpty() const
        {
```

```
        return top == -1;
        } //   end function isEmpty

        //   determine whether Stack is full
        bool   isFull() const
        {
        return top == size - 1;
        } //   end function isFull
}; // end class template Stack

// constructor template
template< typename T  >
Stack< T >::Stack( int s ) : size(   s > 0 ? s : 10 ), // validate size
top( -1 ), // Stack initially empty
stackPtr( new T[ size ] ) // allocate memory for elements
{
        //   empty body
} // end Stack constructor template

// Blank : push element onto Stack;
// if successful, return true; otherwise, return false

// Blank : pop element off Stack;
// if successful, return true; otherwise, return false

#endif
```

- main.cpp

```
#include <iostream>
using std::cout;
using std::endl;
```

```
#include "Stack.h"   // Stack class template definition

int main()
{
        //   Blank : create size 5   double type doubleStack
        double   doubleValue = 1.1;

        cout << "Pushing elements onto doubleStack₩n";

        //   Blank : push 5   doubles onto doubleStack


        cout << "₩nStack is full. Cannot push "   << doubleValue
        << "₩n₩nPopping elements from doubleStack₩n";

        //   Blank : pop elements   from doubleStack

        cout << "₩nStack is empty. Cannot pop₩n";

        //   Blank : create default   size int type intStack
        int   intValue = 1;
        cout << "₩nPushing elements onto intStack₩n";

        //   Blank : push 10   integers onto intStack


        cout << "₩nStack is full. Cannot push "   << intValue
        << "₩n₩nPopping elements from intStack₩n";

        //   Blank : pop elements   from intStack
```

```
        cout << "₩nStack is empty. Cannot pop"   << endl;


        return   0;

} // end main
```