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

<table>
<thead>
<tr>
<th>Standard Library container class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence containers</strong></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>rapid insertions and deletions at back</td>
</tr>
<tr>
<td></td>
<td>direct access to any element</td>
</tr>
<tr>
<td>deque</td>
<td>rapid insertions and deletions at front or back</td>
</tr>
<tr>
<td></td>
<td>direct access to any element</td>
</tr>
<tr>
<td>list</td>
<td>doubly linked list, rapid insertion and deletion anywhere</td>
</tr>
<tr>
<td><strong>Associative containers</strong></td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>rapid lookup, no duplicates allowed</td>
</tr>
<tr>
<td>multiset</td>
<td>rapid lookup, duplicates allowed</td>
</tr>
<tr>
<td>map</td>
<td>one-to-one mapping, no duplicates allowed, rapid key-based lookup</td>
</tr>
<tr>
<td>multimap</td>
<td>one-to-many mapping, duplicates allowed, rapid key-based lookup</td>
</tr>
<tr>
<td><strong>Container adapters</strong></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>last-in, first-out (LIFO)</td>
</tr>
<tr>
<td>queue</td>
<td>first-in, first-out (FIFO)</td>
</tr>
<tr>
<td>priority_queue</td>
<td>highest-priority element is always the first element out</td>
</tr>
</tbody>
</table>
### Common member functions for all STL containers

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

### Functions found only in first-class containers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_size</td>
<td>Returns the maximum number of elements for a container. The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the first element of the container.</td>
</tr>
<tr>
<td>begin</td>
<td>The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the first element of the container.</td>
</tr>
<tr>
<td>end</td>
<td>The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the next position after the end of the container.</td>
</tr>
<tr>
<td>rbegin</td>
<td>The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the last element of the container.</td>
</tr>
<tr>
<td>rend</td>
<td>The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the next position after the last element of the reversed container.</td>
</tr>
<tr>
<td>erase</td>
<td>Erases one or more elements from the container.</td>
</tr>
<tr>
<td>clear</td>
<td>Erases all elements from the container.</td>
</tr>
</tbody>
</table>
- Iterators

```cpp
#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 number1 = *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
```

```
C:\Windows\system32\cmd.exe
Enter two integers: 2 23
The sum is: 25
계속하려면 아무 키나 누르십시오 . ..
```
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>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.</td>
</tr>
<tr>
<td>output</td>
<td>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.</td>
</tr>
<tr>
<td>forward</td>
<td>Combines the capabilities of input and output iterators and retains their position in the container (as state information).</td>
</tr>
<tr>
<td>bidirectional</td>
<td>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.</td>
</tr>
<tr>
<td>random access</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Container Type of iterator supported

<table>
<thead>
<tr>
<th>Container</th>
<th>Type of iterator supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence containers (first class)</strong></td>
<td></td>
</tr>
<tr>
<td>vector</td>
<td>random access</td>
</tr>
<tr>
<td>deque</td>
<td>random access</td>
</tr>
<tr>
<td>list</td>
<td>bidirectional</td>
</tr>
<tr>
<td><strong>Associative containers (first class)</strong></td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>bidirectional</td>
</tr>
<tr>
<td>multiset</td>
<td>bidirectional</td>
</tr>
<tr>
<td>map</td>
<td>bidirectional</td>
</tr>
<tr>
<td>multiset</td>
<td>bidirectional</td>
</tr>
<tr>
<td><strong>Container adapters</strong></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>no iterators supported</td>
</tr>
<tr>
<td>queue</td>
<td>no iterators supported</td>
</tr>
<tr>
<td>priority_queue</td>
<td>no iterators supported</td>
</tr>
</tbody>
</table>
## Iterator Operation

<table>
<thead>
<tr>
<th>Iterator operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All iterators</td>
<td></td>
</tr>
<tr>
<td>++p</td>
<td>Preincrement an iterator.</td>
</tr>
<tr>
<td>p++</td>
<td>Postincrement an iterator.</td>
</tr>
<tr>
<td>Input iterators</td>
<td></td>
</tr>
<tr>
<td>*p</td>
<td>Dereference an iterator.</td>
</tr>
<tr>
<td>p = p1</td>
<td>Assign one iterator to another.</td>
</tr>
<tr>
<td>p == p1</td>
<td>Compare iterators for equality.</td>
</tr>
<tr>
<td>p != p1</td>
<td>Compare iterators for inequality.</td>
</tr>
<tr>
<td>Output iterators</td>
<td></td>
</tr>
<tr>
<td>*p</td>
<td>Dereference an iterator.</td>
</tr>
<tr>
<td>p = p1</td>
<td>Assign one iterator to another.</td>
</tr>
<tr>
<td>Forward iterators</td>
<td></td>
</tr>
<tr>
<td>**p</td>
<td>Predecrement an iterator.</td>
</tr>
<tr>
<td>p--</td>
<td>Postdecrement an iterator.</td>
</tr>
<tr>
<td>Bidirectional iterators</td>
<td></td>
</tr>
<tr>
<td>--p</td>
<td>Predecrement an iterator.</td>
</tr>
<tr>
<td>p--</td>
<td>Postdecrement an iterator.</td>
</tr>
</tbody>
</table>
#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

    // Example

    // Random-access iterators
    // Increment the iterator p by i positions.
    p += i;
    // Decrement the iterator p by i positions.
    p -= i;
    // Expression value is an iterator positioned at p incremented by i positions.
    p + i or i + p;
    // Expression value is an iterator positioned at p decremented by i positions.
    p - i;
    // Expression value is an integer representing the distance between two elements in the same container.
    p - p1;
    // Return a reference to the element offset from p by i positions
    p[i];
    // 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.
    p >= p1;
}
vector<int> integers; // create vector of ints

cout << "The initial size of integers is: " << integers.size();
<< "The 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 << "The size of integers is: " << integers.size();
<< "The capacity of integers is: " << integers.capacity();
cout << "WnWnOutput array using pointer notation: ";

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

cout << "WnOutput vector using iterator notation: ";
printVector(integers);
cout << "WnReversed contents of vector integers: ";

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

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

The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 3

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
계속하려면 아무키나 누르십시오...
[Exercise]
- See the final print out value and complete the Stack.h and main.cpp
- Generate object based on Class Template

```cpp
#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

#include <iostream>
using std::cout;
using std::endl;
```cpp
#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
    << "\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 << "Pushing elements onto intStack\n";

    // Blank : push 10 integers onto intStack

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

    // Blank : pop elements from intStack
```
```cpp
cout << "WnStack is empty. Cannot pop" << endl;

return 0;

} // end main
```

Pushing elements onto doubleStack
1 2 2 3 3 4 4 5 5
Stack is full. Cannot push 6 6

Popping elements from doubleStack
5 5 4 4 3 3 2 2 1 1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

계속하려면 아무 키나 누르실시오...