```
/* *********************************************************************
    Demonstrates a generic stack ADT.
    Uses composition with std::deque<T> as its underlying storage.
    Mimics std::stack<T> for member function signatures.
    Advice: always choose std::stack<T> when in need of a stack.
********************************************************************/
#ifndef STACK_H
#define STACK_H

#include <deque>
#include <exception>

template <class T>
class Stack {
protected:
   std::deque<T> container;  // container for the stack elements
public:
   Stack() = default;
   virtual ~Stack() = default;

   Stack(const Stack<T>& other)               = default; // copy constructor
   Stack<T>& operator=(const Stack<T>& other) = default; // copy assignment

   Stack(Stack<T>&& other)               = default; // move constructor
   Stack<T>& operator=(Stack<T>&& other) = default; // move assignment

   // Returns the number of elements in the stack
   typename std::deque<T>::size_type size() const;

   // Returns whether the stack is empty
   bool empty() const;

   // Inserts a new element at the top of the stack
   void push(const T& elem);

   // Removes the element on top of the stack.
   void pop();

   // Returns a reference to the top element in the stack.
   T& top();

   // Returns a constant reference to the top element in the stack.
   const T& top() const;
};
// Include Stack.cpp. Only for class templates.
// Be sure to remove Stack.cpp from your IDE's project.
#include "Stack.cpp"
#endif // STACK_H
```

```
//#include "Stack.h"      // our own Stack<T> class
// The include line above is commented out because this entire
// stack.cpp file is already included at the bottom of stack.h

// Returns the number of elements in the stack
template<class T>
inline typename std::deque<T>::size_type Stack<T>::size() const
{
   return container.size();
}

// Returns whether the stack is empty
template<class T>
inline bool Stack<T>::empty() const
{
   return container.empty();
}

// Inserts a new element at the top of the stack
template<class T>
inline void Stack<T>::push(const T & elem)
{
   container.push_back(elem);
}

// Removes the element on top of the stack.
template<class T>
inline void Stack<T>::pop()
{
   if (container.empty()) {
      throw std::underflow_error("Empty stack - no pop()");
   }
   container.pop_back();
}

// Returns a reference to the top element in the stack.
template<class T>
inline T & Stack<T>::top()
{
   if (container.empty()) {
      throw std::underflow_error("Empty stack - no top()");
   }
   return container.back();
}

// Returns a constant reference to the top element in the stack.
template<class T>
inline const T & Stack<T>::top() const
{
   if (container.empty()) {
      throw std::underflow_error("Empty stack - no top()");
   }
   return container.back();
}
```

COMP 5421/1 BB Test Driver: A sample class template using std::deque July 16, 2018

```
#include <iostream>
#include "Stack.h"      // use our own Stack<T> class
using namespace std;

// test drive for our stack class
int main()
{
   try {
      Stack<int> intStack;

      // push five elements onto the stack
      for(int x = 1; x <=5; ++x)
         intStack.push(x);

      // pop and print three elements from the stack
      for (int x = 1; x <= 3; ++x)
      {
         cout << intStack.top() << " "; // process top element
         intStack.pop();                // then pop it
      }
      cout << endl;

      // modify top element
      intStack.top() = 111;

      // push three new elements
      for (int x = 6; x <= 8; ++x)
         intStack.push(x);

       // print and pop six elements, one element too many
      for (int x = 1; x <= 6; ++x)
      {
         cout << intStack.top() << " "; // process top element
         intStack.pop();                // then pop it
      }
      cout << endl;
   }
   catch (const exception& e)
   {
      cerr << "\nException: " << e.what() << endl;
   }
}
```

Output

```
5 4 3
8 7 6 111 1
Exception: Empty stack - no top()
```