**CSCI 241 Assignment 5  
100 points**

**Purpose**

This assignment is an exercise in implementing the stack ADT using a dynamically-allocated array, as well as techniques for managing dynamically-allocated storage in C++..

**Assignment**

In this assignment, you will write a class called Stack that will encapsulate a dynamically-allocated array of integers.

A *driver program* is provided for this assignment to test your implementation. You don't have to write the tests.

**Program**

You will need to write a single class for this assignment, the Stack class. You will need to implement several methods and functions associated with this class.

The class should be implemented as two separate files. The class declaration should be placed in an appropriately named header (.h) file. The implementations of the class methods and any other associated functions should be placed in a separate .cpp file for the class.

**class Stack**

*Data members*

* a pointer to an int. This data member will be referred to as the *stack array pointer*. It will be used to dynamically allocate an array of int (the *stack array*).
* an unsigned integer or size\_t variable used to keep track of the number of elements in the stack array. This data member will be referred to as the *stack capacity*.
* an unsigned integer or size\_t variable used to store the current number of items stored in the stack array. This data member will be referred to as the *stack size*. The stack size must always be less than or equal to the stack capacity.

You may also choose to have a data member to keep track of the subscript of the top item in the stack (the *stack top subscript*). Since the subscript of the top item is always equal to the stack size - 1, doing so is optional, but if you make this a separate data member it should be an integer.

In addition to the data members described above, your class declaration will need prototypes for the methods described below.

*Methods and associated functions*

* Stack::Stack()

The class should have a default constructor that takes no arguments. The constructor should set the stack size and stack capacity to 0 and the stack array pointer to nullptr.

* Stack::~Stack()

The class should have a destructor that deletes the dynamic memory for the stack array. The destructor should NOT call the clear() method.

* Stack::Stack(const Stack& other)

The class should also have a proper copy constructor. Your code should account for the possibility that you might be copying an empty Stack object.

* Stack& Stack::operator=(const Stack& other)

The assignment operator should be properly overloaded to allow one Stack object to be assigned to another. Your code should account for the possibility that you might be copying an empty Stackobject.

* ostream& operator<<(ostream& lhs, const Stack& rhs)

The output operator should be overloaded so that a Stack can be printed on the standard output. As in Assignment 4, this will need to be a standalone friend function rather than a method.

The items stored in the stack should be printed starting with the first array element (subscript 0) and ending with the last valid element (subscript stack size - 1). Print a space after each stack element.

* void Stack::clear()

This method should set the stack size to 0. It should not change the stack capacity or the stack array.

* size\_t Stack::size() const

This method should return the stack size.

* size\_t Stack::capacity() const

This method should return the stack capacity.

* bool Stack::empty() const

This method should return true if the stack size is equal to 0; otherwise it should return false.

* int Stack::top() const

This method should return the top element of the stack array (the one at the subscript stack size - 1). You may assume this method will not be called if the stack is empty.

* void Stack::push(int val)

This method takes an integer argument, the value to insert into the stack. If the stack is full (the stack size is equal to the stack capacity), this method will need to call the reserve() method to increase the capacity of the stack array and make room for the value to insert. If the stack capacity is currently 0, pass a new capacity of 1 to the reserve() method. Otherwise, pass a new capacity of twice the current stack capacity to the reserve() method.

Using the stack size as the subscript, copy the value to be inserted into the stack array. The stack size should then be incremented by 1.

* void Stack::pop()

This method should decrement the stack size by 1, which effectively removes the top (last) value from the stack array. No changes need be made to the stack array contents. You may assume this method will not be called if the stack is empty.

* void Stack::reserve(size\_t n)

This method increases the capacity of the stack array. It takes a single integer argument, the new capacity. The logic for this method should look something like this:

* 1. If n is less than the stack size or n is equal to the current stack capacity, simply return.
  2. Set the stack capacity to n.
  3. Declare a temporary array pointer (a pointer to an int).
  4. If the stack capacity is 0, set the temporary array pointer to nullptr. Otherwise, use the temporary array pointer to allocate an array of int. The number of elements in the new temporary array should be equal to the updated stack capacity.
  5. Copy the contents of the stack array into the temporary array.
  6. Delete the stack array.
  7. Set the stack array pointer equal to the temporary array pointer.
* int Stack::operator[](size\_t n) const

This method should return the element of the stack array at subscript n. You may assume this method will not be called with an argument n that is out of range.

* int& Stack::operator[](size\_t n)

This method should return the element of the stack array at subscript n. You may assume this method will not be called with an argument n that is out of range.

* bool Stack::operator==(const Stack& rhs) const

This method should return true if the values stored in the stack array of the object that called the method are identical to the values stored in the stack array of the Stack object passed in as rhs.

The logic for this method is less difficult that it might initially appear to be. The first step is to compare the stack sizes of the two stacks. If they are different, return false (two stacks of different lengths can not be equal).

Otherwise, loop through the elements of both stack arrays, starting at 0 and stopping when you reach the stack size. (The size of *which stack* doesn't matter, since you know they're the same by this point.) Compare the current element from each stack array. If elements are different, return false. If the elements are the same, don't return true; do nothing and let the loop continue.

Once the loop ends and you've reached the end of both stacks, return true.

**Output**

A driver program, assign5.cpp is provided for this assignment. The purpose of a driver program is to test other pieces that you code. You do not need to write the driver program yourself. A copy of the driver program can also be found on turing at /home/turing/t90kjm1/CS241/Code/Fall2017/Assign5/assign5.cpp.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PROGRAM: CSCI 241 Assignment 5

PROGRAMMER: your name

LOGON ID: your z-ID

DUE DATE: due date of assignment

FUNCTION: This program tests the functionality of the Stack class.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include "Stack.h"

using std::cout;

using std::endl;

int main()

{

cout << "Testing default constructor\n\n";

Stack s1;

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "Testing push()\n\n";

for (int i = 10; i < 80; i+= 10)

s1.push(i);

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

cout << endl;

for (int i = 15; i < 85; i+= 10)

s1.push(i);

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "Testing copy constructor()\n\n";

Stack s2 = s1;

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "Testing top()\n\n";

cout << "Top item of s1: " << s1.top() << endl << endl;

cout << "Testing pop()\n\nTop item of s1: ";

while (!s1.empty())

{

cout << s1.top() << ' ';

s1.pop();

}

cout << endl << endl;

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "Testing assignment operator\n\n";

Stack s3;

s3 = s2;

cout << "s2 (size " << s2.size() << "): " << s2 << endl;

cout << "s3 (size " << s3.size() << "): " << s3 << endl << endl;

cout << "Testing clear()\n\n";

s2.clear();

cout << "s2: " << s2 << endl;

cout << "s2 size: " << s2.size() << ", capacity: " << s2.capacity() << endl;

cout << "s2 is " << ((s2.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "s3: " << s3 << endl;

cout << "s3 size: " << s3.size() << ", capacity: " << s3.capacity() << endl;

cout << "s3 is " << ((s3.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "Testing assignment to self and swap\n\n";

s3 = s3;

s2 = s3;

s3.clear();

cout << "s2 (size " << s2.size() << "): " << s2 << endl;

cout << "s3 (size " << s3.size() << "): " << s3 << endl << endl;

cout << "Testing chained assignment\n\n";

Stack s4;

s4 = s3 = s2;

cout << "s2 (size " << s2.size() << "): " << s2 << endl;

cout << "s3 (size " << s3.size() << "): " << s3 << endl;

cout << "s4 (size " << s4.size() << "): " << s4 << endl << endl;

Stack s5 = s4;

cout << "s5 (size " << s5.size() << "): " << s5 << endl << endl;

cout << "Testing write version of subscript operator\n\n";

for (size\_t i = 0; i < s5.size(); ++i)

s5[i] += 5;

cout << "s5 (size " << s5.size() << "): " << s5 << endl << endl;

cout << "Testing read version of subscript operator\n\nv5: ";

for (size\_t i = 0; i < s5.size(); ++i)

cout << s5[i] << ' ';

cout << endl << endl;

cout << "Testing const correctness\n\n";

const Stack& r4 = s4;

cout << "s4: " << r4 << endl;

cout << "s4 size: " << r4.size() << ", capacity: " << r4.capacity() << endl;

cout << "s4 is " << ((r4.empty()) ? "empty\n" : "not empty\n");

cout << "Top item of s4: " << r4.top() << endl;

cout << "Element 2 of s4: " << r4[2] << endl << endl;

s1 = r4;

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

cout << endl;

cout << "Testing equality operator\n\n";

cout << "s1 and s4 are " << ((s1 == s4) ? "equal\n" : "not equal\n");

cout << "s5 and s4 are " << ((s5 == s4) ? "equal\n" : "not equal\n");

s1.pop();

cout << "s1 and s4 are now " << ((s1 == s4) ? "equal\n" : "not equal\n");

cout << endl;

s1.clear();

cout << "s1: " << s1 << endl;

cout << "s1 size: " << s1.size() << ", capacity: " << s1.capacity() << endl;

cout << "s1 is " << ((s1.empty()) ? "empty\n" : "not empty\n");

return 0;

}

**Output**

A successful run of the entire program should produce the output shown below.

Testing default constructor

s1:

s1 size: 0, capacity: 0

s1 is empty

Testing push()

s1: 10 20 30 40 50 60 70

s1 size: 7, capacity: 8

s1 is not empty

s1: 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s1 size: 14, capacity: 16

s1 is not empty

Testing copy constructor()

s1: 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s1 size: 14, capacity: 16

s1 is not empty

Testing top()

Top item of s1: 75

Testing pop()

Top item of s1: 75 65 55 45 35 25 15 70 60 50 40 30 20 10

s1:

s1 size: 0, capacity: 16

s1 is empty

Testing assignment operator

s2 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s3 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

Testing clear()

s2:

s2 size: 0, capacity: 16

s2 is empty

s3: 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s3 size: 14, capacity: 16

s3 is not empty

Testing assignment to self and swap

s2 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s3 (size 0):

Testing chained assignment

s2 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s3 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s4 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s5 (size 14): 10 20 30 40 50 60 70 15 25 35 45 55 65 75

Testing write version of subscript operator

s5 (size 14): 15 25 35 45 55 65 75 20 30 40 50 60 70 80

Testing read version of subscript operator

v5: 15 25 35 45 55 65 75 20 30 40 50 60 70 80

Testing const correctness

s4: 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s4 size: 14, capacity: 16

s4 is not empty

Top item of s4: 75

Element 2 of s4: 30

s1: 10 20 30 40 50 60 70 15 25 35 45 55 65 75

s1 size: 14, capacity: 16

s1 is not empty

Testing equality operator

s1 and s4 are equal

s5 and s4 are not equal

s1 and s4 are now not equal

s1:

s1 size: 0, capacity: 16

s1 is empty

**Other Points**

* A makefile is required. Same as always. Make sure it has appropriate rules for all the pieces involved.
* The driver program assumes that your header file is called Stack.h.
* Write this program in a fashion similar to the last assignment. Start off at the beginning of the driver program and try to get it working piece by piece. Maintain a working program as you go.
* Programs that do not compile on turing/hopper automatically receive 0 points.
* Submit your program using the electronic submission guidelines posted on the course web site.