CSCI 2 - Assignment 7

Subject<u>: Writing and Testing a Self-Growing Stack Class Using Pointer and Dynamic Memory Allocation</u>

Preparation Work Due Date: 1 November 2017

Coding Testing Grading Final Due Date: 6 November, 2017 Monday

Assignment must be submission ready before your lab starts. Thanks.

Sorry. No debugging support is available on the day of submission! Plan in advance so that your program is debugged the day before or earlier than the submission day.

Preparation Work

This can be done on computer and showed to me on computer. Implement the rule of three for the following class:

```
class Employee{
private:
string firstName;
string * lastName;
int hourWorked;
double PayRate;
double Salary;
public:
//1. write separate default and explicit constructors. The memory allocation for
//pointer lastName is done inside the constructor.
//2. Write a copy constructor that makes deep copy.
//3. Write an assignment operator that makes deep copy and deletes the memory
//allocated to the pointer which is being given a new pointee.
//4. Write a virtual destructor that de-allocates the memory allocated by
//constructors.
```

//5. Write a toString function that causes no mutation in the object and returns a
//const object.
//All constructors, copy constructor, assignment operator, and destructor must
//have an output statement. Example is:
//cout<<"From Employee destructor"<<endl;
};
Write separate .h and .cpp files. In main function create an object of Employee, print it. Then create two employee objects, one created by default and other by explicit constructor call. Set first object equal to second using assignment operator and then print both. Write the following stand-alone function:
void print(Employee E);
print, simply prints E to console.

Place all your main function code in a pair of curly braces as below:

{

//code in main function

}

Prove to yourself that number of constructor, and copy constructor calls are equal to the number of destructor calls. If not then something is not correct in code.

Background for Assignment

The stack classes you used so far are either provided by C++ or provided by me (instructor), which was limited in capacity. By now you have learned that using pointers you can allocate memory for an array dynamically at runtime and de-allocate that array as needed. In this program you will modify my stack class to alter and add the required functions to fulfill following goals.

1. This new stack will start out at some default initial capacity, but will never run out of capacity when elements more than the initial capacity are added. Instead the stack will grow to expand and accommodate new additions.
2. This new/modified stack will enforce rule of three that we have discussed in number of ways in lecture, e-book, and also later in this document. Correct coding to satisfy rule of three would ascertain that class is designed to have:
    a. No memory leaks.
    b. Makes deep copies when objects are copied using assignment operator or default call to copy constructor.

3. After completing the coding of new Stack class, you would demonstrate the use of this class to fulfill its correct functioning as a self-growing stack, and its conformance to the goals of rule of three.

Guidance is provided to you in this project by the design and code of self-growing queue class that I have done and a copy of that is provided to you.

Files you will get from me
You will get exactly the same files for ItemType, StackInterface, and Stack classes from me that you got in assignment 4. No changes are to be made to ItemType and StackInterface files. [Any fact making any changes to them disqualifies your project for credit]. You would also get Stack.h file from me that has the new specifications and additions. You are responsible for writing algorithms for, coding, and testing Stack.cpp and main function that would fulfill testing and demonstration rules for this project.

Changes would only be made to the Stack.h and Stack.cpp, according to the new specifications given below. Understand that uncovering specifications and coding to them is a Programmer's task! You are free to ask questions that would clarify specifications beyond this document. It is not client's job to write your algorithm for you. I will be available to guide you in the right directions, when you are in journey to write code and test members of Stack class.

Stack Class Specifications

Stack class implements StackInterface abstract class. This version is self-growing when the array on to which items are pushed is filled. The self-expanding array is implemented by using a pointer of ItemType that points to a dynamically allocated and growing array. The memory allocation for this array is done inside the constructor of Stack Class.

Since this class contains a pointer data member for which memory allocation is done inside the constructor, the rule of three kicks in. The rule of three requires that when a class has one or more pointer data members for which the memory allocation is done inside the constructor, then the class must provide the below to replace defaults:
1. A destructor that will de-allocate the memory that was allocated by the constructor.
Understand that destructor are:
    A. Not overloadable
    B. Are NEVER called in programmer code by the programmer. They work behind the scene automatically.
    C. Are NOT ALLOWED taking arguments.
    D. Have same name as class name but with a ~ in front.
2. A user defined copy constructor that makes deep copies.
3. A user defined Assignment operator that makes a deep copy when one object is copied into other using the assignment operator.

Since class would have an array that would expand by itself when push is attempted after array is filled, major changes are required to push function that was there in the limited array class earlier used by you in assignments 4 and 5.

Documentation for the rest of the class is below which is more or less the copy of Stack.h file. The table below contains the description of class member description and then their C++ declaration in the header file

```
class Stack :public StackInterface
{
private:
        /*
        Constant storing initial MAX capacity of the array. For revealing bugs in code,
        this must be set to 1.
        */
        const static int MAX = 1;
        /*
        Constant storing the number by which array must grow when previous array is filled and
additional elements are pushed on to the stack..
        For revealing bugs in code, this must be set to 1.
        */
        const static int GROWBY = 1;

      /*
      Pointer showing the position of pointer pointing to the top element on the stack.
     */
        int top_position;
        /*
        ItemType pointer data member which points to dynamically growing and managed array
        that holds the stack.
        */
        ItemType * items;
        /*
        Number of items in the stack;
        */
        size_t numItems;
        /*
        Capacity of the array holding stack elements.
        */
        size_t arrayCapacity;
public:
        /*
        Major changes are made to constructor of Stack class. It's responsibilities are:
        1.  Dynamically assign memory to ItemType array pointed to by pointer items.
        2.  Sets proper initial values for class members' numItems and arrayCapacity.
            For me to check if your code follows rule of three correctly, below code MUST
            be placed as the last line in your constructor body!
            cout << "From Stack Constructor.\n";
```

```
 */
 Stack();

  /*
   Push would be the most important function you would need to write in this project. For
   Some assistance and ideas are provided by the manner in which I implemented enqueue
   function in self-growing queue class, whose copy you have and you have studied it.
   Basically the push function adds the item on the top of stack if number of items in the stack
   Are smaller than the capacity of the array holding the stack. Otherwise the function grows
   the stack by magnitude stored in class constant GROW and then adds the item to stack.
  Look carefully how we have done enqueue function to ascertain that there are no memory
  leaks caused by push function, and yet it fulfills all responsibilities that a push function  is
  required to fulfill.
*/
void push(ItemType item);

/*
No change is made to this function compared to the file you got in assignment 4. You can
Copy and paste same code that you were given for stack class in assignment 4
*/
void pop();




/*
No change is made to this function compared to the file you got in assignment 4. You can
Copy and paste same code that you were given for stack class in assignment 4
*/
ItemType top() const;

/*
No change is made to this function compared to the file you got in assignment 4. You can
Copy and paste same code that you were given for stack class in assignment 4
*/
bool isEmpty() const;


/*
Programmer defined destructor de-allocates the memory that was assigned inside the
constructor. The programmer defined constructor is another part of rule of three. In order for
me to confirm that your Stack class has no memory leaks, you MUST place the below code line
as last line in your destructor code.
cout << "From Stack Destructor.\n";
*/
~Stack();
```

```
/*
Copy Constructor is a part of rule of three implementation. Default
copy constructor provided by C++ only makes shallow copies. To make deep
copies of data pointed to by a pointer and allocated dynamically at runtime
a programmer defined copy constructor that would do that is needed.
Copy Constructor is called automatically when a function would return a value
of type Stack or when Stack object is passed by value to a function. Copy
constructor should NOT be called by user code. Goal of copy constructor
is to make deep copies of data which is pointed to by pointer when object
is either passed to a function by value or returned as a value from a function.
In order for me to confirm that your code works correctly, the below MUST be the
Last line in your Copy Constructor code.
cout << "From Stack Copy Constructor.\n";
@param st is the Stack object being copied inside the copy constructor.
*/
Stack(const Stack & st);

/*
Assignment operator is a part of rule of three implementation.
Default assignment operator provided by C++ only makes shallow copies. To make deep
copies of data pointed to by a pointer and allocated dynamically at runtime
a programmer defined assignment operator that would do that is needed.
Assignment operator is used when one object is copied into another using the
= operator. This must also work if = is used in a cascaded manner.
For example it must work if one does the following:
Stack X, Y, Z, T;
X= Y = Z = T;
 The assignment operator MUST work correctly even when user attempts a self-copy. Example of
Self-copy is below.
 Stack ST;
 ST = ST;

For me to ascertain that your code follows rule of three and has no memory leak, please place
The code line below before the return statement in the assignment operator.
cout << "From Stack Assignment operator.\n";
@param Stk is the stack object on right side of assignment operator
@return the value to be copied in the left side of assignment operator.
*/
const Stack & operator = (const Stack & Stk);

/*
Functoion getArrayCapacity returns the capacity of the array used for stack when call to
this function is made.
@return the capacity of array used to hold stack elements.
*/
size_t getArrayCapacity() const;

/*
```

```
        Function getNumItems returns the number of items in the stack at the time when this function
        is called.
        @return the number of elements stored in the stack.
        */
        size_t getNumItems() const;

protected:
        /*
        copy function is optional. However it condenses the copy code that would be used in both
        the copy constructor and in the assignment operator. Use of this function advances
        the code re-use technology.
        @param Stk is the stack object to be copied to this Stack object.
        */
        void copy(const Stack & Stk);

private:
        /*
        Function isFull is not needed in the dynamically growing stack. However,
        we must give it a body because it was a pure virtual function in StackInterface
        class and since Stack class derives from StackInterface class, it must be
        implemented to allow to create objects of type Stack. [Recall that if a base class
        pure virtual function is not implemented by the derived class, then derived class
        will also become abstract, and cannot be instantiated]. You can just have the
        function return false as the dynamically growing stack will never be full.
        @return false because dynamically growing stack always has capacity to add more elements.
        */
        bool isFull() const;
};
```

Please write one function at a time and make sure it works before moving to next. Use of debugger (after compiling, press F10 to start the debugger and then press F11, when function or constructor execution is in the code) allows you to walk through all code lines and ascertain that that function or constructor works correctly. Additional code testing will certify the line by line debugger walk through.

Main Function:

Purpose of main function is to prove that your Stack class fulfills following goals.

1. Works as a self-expanding stack data structure and can be used in all those software situations where stack data structure is usable. [For example try re-doing an additional version of assignment 5 with this new stack you developed. Simple way of doing this would be to make copy of your entire Assignment 5 project and paste it on your hard drive. Then delete code in your old Stack.h and Stack.cpp, and copy and paste new Stack.h, and Stack.cpp in respective files. If your includes have been done correctly, then your assignment 5 must work exactly the way it did last time with old Stack class].
2. Its use has no memory leaks.

3. Copy constructor works correctly. That means that when a Stack object is passed to a function, or returned as a value from a function then it makes deep copies.
4. Deep copy is made when One Stack class Object is copied into another.

Table below gives my code and output for the main function. You can use my main function code to start testing your program. However, I may have additional main functions to test your final program. This will demo the mechanics of tests of Stack class. [Code in main function must be placed in (non-Signature) pair of curly braces as the style below:

```
int main()
{
    {
            //Main function code
     }
return 0;
}
```

Line #	Main Function Source Code
1	#include **"Stack.h"**
2	#include **<iostream>**
3	using namespace std**;**
4	Stack FillEmptyStackWithIntegersAndReturnIt**();**
5	void PopAndPrintStack**(**Stack Stk**);**
6	int main**()**
7	**{**
8	{
9	cout << "Creating a new Stack.\n";
10	Stack IntStack = FillEmptyStackWithIntegersAndReturnIt();
11	cout << "Making a copy of original stack.\n";
12	Stack CopyOfIntStack;
13	CopyOfIntStack = IntStack;
14	cout << "Topping and Popping the original Stack:\n";
15	PopAndPrintStack(IntStack);
16	cout << "Topping and Popping the copy stack.\n";
17	PopAndPrintStack(CopyOfIntStack);
18	}
19	system**("pause");**
20	return **0;**
21	**}**
22	//--
23	void PopAndPrintStack**(**Stack Stk**)**
24	**{**
25	cout **<<** **"Now printing the integers pushed on the stack.\n";**
26	
27	while **(!**Stk.isEmpty**())**
28	**{**
29	cout **<<** Stk.top**()**.Integer **<<** **" ";**

```
                Stk.pop();
        }
        cout << endl;
}
//-----------------------------------------------------------------
Stack FillEmptyStackWithIntegersAndReturnIt()
{
        Stack Stk;
        cout << "We will push some integers on the stack now.\n";
        bool done = false;
        int val = int();
        while (!done)
        {
                cout << "Please enter an integer to be pushed on stack : ";
                cin >> val;
                ItemType item;
                item.Integer = val;
                Stk.push(item);
                cout << "The number of items on stack :" << Stk.getNumItems() << endl;
                cout << "The capacity of stack array is: " << Stk.getArrayCapacity() << endl;
                cout << "More data? Enter 0 to continue and 1 to exit: ";
                cin >> done;
        }

        return Stk;
}
```

OUTPUT

```
Creating a new Stack.
From Stack CTOR.
We will push some integers on the stack now.
Please enter an integer to be pushed on stack : 10
Using the existing array to push item on stack.
The number of items on stack :1
The capacity of stack array is: 1
More data? Enter 0 to continue and 1 to exit: 0
Please enter an integer to be pushed on stack : 9
Expanding the existing array to push item on stack.
The number of items on stack :2
The capacity of stack array is: 2
More data? Enter 0 to continue and 1 to exit: 0
Please enter an integer to be pushed on stack : 8
Expanding the existing array to push item on stack.
The number of items on stack :3
The capacity of stack array is: 3
More data? Enter 0 to continue and 1 to exit: 0
Please enter an integer to be pushed on stack : 7
Expanding the existing array to push item on stack.
The number of items on stack :4
The capacity of stack array is: 4
More data? Enter 0 to continue and 1 to exit: 0
Please enter an integer to be pushed on stack : 6
Expanding the existing array to push item on stack.
```

```
The number of items on stack :5
The capacity of stack array is: 5
More data? Enter 0 to continue and 1 to exit: 0
Please enter an integer to be pushed on stack : 5
Expanding the existing array to push item on stack.
The number of items on stack :6
The capacity of stack array is: 6
More data? Enter 0 to continue and 1 to exit: 1
From Stack Copy Constructor.
From Stack Destructor.
Making a copy of original stack.
From Stack CTOR.
From Stack Assignment operator.
Topping and Popping the original Stack:
From Stack Copy Constructor.
Now printing the integers pushed on the stack.
5  6  7  8  9  10
From Stack Destructor.
Topping and Popping the copy stack.
From Stack Copy Constructor.
Now printing the integers pushed on the stack.
5  6  7  8  9  10
From Stack Destructor.
From Stack Destructor.
From Stack Destructor.
```

Demonstrating the Proof of No Memory Leak

After ascertaining that all calls to new are matching with all delete calls in all functions other than constructor, destructor, and in absence of memory leak from properly written constructor and destructors the number of constructor and copy constructor calls would match the number of destructor calls. [Note that this proof would not work if analysis of the code of followings reveals logical inconsistencies in them:

1. Constructor
2. Copy Constructor
3. Assignment operator
4. Destructor

Special techniques would be needed in large software projects to uncover memory leaks.]

However in simple situations like this project (after analyzing and looking at constructor and rule of three component codes), this test will be adequate.

In Table above you can count the number of constructor and copy constructor calls and match them numerically with the number of destructor calls. In electronic copy all constructor and copy constructor calls are in blue and all destructor calls are in red. There are seven calls to constructor and copy constructor and seven to destructor. Thus this test is consistent. The code in line 13 calls the overloaded assignment operator, whose output is shown in line 93.

Rule of three conformances is demonstrated by calls to copy constructor, destructor and assignment operator. Line 58 to 87 demonstrates that stack class works like a normal stack data structure, as items pushed on to the stack in order 10, 9, 8, 7, 6, and 5 are popped in order 5, 6, 7, 8, 9, and 10. (Please see output on lines 97 and 102).