CS 202 - Computer Science II

Project 10

Due date (FIXED): Tuesday, 5/1/2019, 11:59 pm

Objectives: The main objectives of this project are to test your ability to create and use stack-based dynamic data structures, and generalize your code using templates. A review of your knowledge to manipulate dynamic memory, classes, pointers and iostream to all extents, is also included. You may from now on freely use **square bracket**-indexing, **pointers**, **references**, all **operators**, the **<cstring>** library, and the **std::string** type as you deem proper.

Description:
For this project you will create a templated Stack class, with an Array-based and a Node-based variant. A Stack is a Last In First Out (LIFO) data structure. A Stack exclusively inserts data at the top (**push**) and removes data from the top (**pop**) as well. The Stack's **m_top** data member is used to keep track of the current Stack size, and through that also infer the position of the last inserted element (the most recent one).

The following provided specifications refer to Stacks that work with DataType class objects, similarly to the previous project. For the this Project's requirements, you will have to make the necessary modifications so that your ArrayStack and NodeStack and all their functionalities are generalized templated classes.

Templated Array-based Stack:
The following header file excerpt is used to explain the required specifications for the class. This only refers an Array-based Stack that holds elements of type class DataType. You **have to template** this class, and provide the necessary header file with the necessary declarations and implementations:

```
const size_t MAX_STACKSIZE = 1000;

class ArrayStack{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const ArrayStack & arrayStack);
  public:
    ArrayStack();                                                         //(1)
    ArrayStack(size_t count, const DataType & value);                     //(2)
    ArrayStack(const ArrayStack & other);                                 //(3)
    ~ArrayStack();                                                        //(4)

    ArrayStack & operator= (const ArrayStack & rhs);                      //(5)

    DataType & top();                                                    //(6a)
    const DataType & top() const;                                        //(6b)

    void push(const DataType & value);                                    //(7)
    void pop();                                                           //(8)

    size_t size() const;                                                  //(9)
    bool empty() const;                                                  //(10)
    bool full() const;                                                   //(11)
    void clear();                                                        //(12)
    void serialize(std::ostream & os) const;                             //(13)

  private:
    DataType m_container[MAX_STACKSIZE];
    size_t m_top;
};
```

The **ArrayStack** Class will contain the following **private** data members:

- **m_container,** the array that holds the data. Note: Here it is given to hold DataType class objects and have a maximum size of MAX_STACKSIZE. For this Project's requirements, both of these parameters will have to be determined via Template Parameters.
- **m_top,** a size_t, tracking the size of the currently existing elements in the ArrayStack. Through that, the position of the most recently inserted element of the m_container can be inferred. *Note*: This should never exceed MAX_STACKSIZE.

,will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new ArrayStack object with no valid data.
- **(2) Parametrized Constructor** – will instantiate a new ArrayStack object, which will hold size_t count number of elements in total, all of them initialized to be equal to the parameter value.
- **(3) Copy Constructor** – will instantiate a new ArrayStack object which will be a separate copy of the data of the **other** ArrayStack object which is getting copied. *Note*: Consider whether you actually need to implement this.
- **(4) Destructor** – will destroy the instance of the ArrayStack object. *Note*: Consider whether you actually need to implement this.
- **(5) operator=** will assign a new value to the calling ArrayStack object, which will be an exact copy of the rhs object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Note*: Consider whether you actually need to implement this.
- **(6a,6b) top** returns a Reference to the last inserted element of the stack.
  Note1: Consider carefully how this can be inferred from the m_top member.
  Note2: Since it returns a Reference, before calling this method the user must ensure that the stack is not empty.
- **(7) push** inserts at the top of the stack an element of the given value.
  Note1: Consider carefully how this relates to the m_top member.
  Note2: Since m_top can never exceed MAX_STACKSIZE, checking if the stack is full prior to pushing a new element makes sense.
- **(8) pop** removes the top element of the stack.
  Note1: Consider carefully how this relates to the m_top member.
  Note2: Since m_top is an unsigned (size_t) type value, checking if the stack is empty prior to popping an element makes sense.
- **(9) size** will return the current size of the ArrayStack.
- **(10) empty** will return a bool, true if the ArrayStack is empty.
- **(11) full** will return a bool, true if the ArrayStack is full.
- **(12) clear** performs the necessary actions, so that after its call the ArrayStack will be semantically considered empty.
- **(13) serialize** outputs to the parameter ostream os the complete content of the calling ArrayStack object (ordered from top to bottom).

as well as a non-member overload for:

- **(i) operator<<** will output (to terminal or file) the complete content of the arrayStack object passed as a parameter.

Templated Node-based Stack:

The following header file excerpt is used to explain the required specifications for the class. This only refers a Node-based Stack that holds elements of type class DataType. You will additionally **have to template** this class, and provide the necessary header file with the necessary declarations and implementations:

```
class NodeStack{
  friend std::ostream & operator<<(std::ostream & os,              //(i)
                                   const NodeStack & nodeStack);
  public:
    NodeStack();                                                    //(1)
    NodeStack(size_t count, const DataType & value);                //(2)
    NodeStack(const NodeStack & other);                             //(3)
    ~NodeStack();                                                   //(4)

    NodeStack & operator= (const NodeStack & rhs);                  //(5)

    DataType & top();                                              //(6a)
    const DataType & top() const;                                  //(6b)

    void push(const DataType & value);                              //(7)
    void pop();                                                     //(8)

    size_t size() const;                                            //(9)
    bool empty() const;                                            //(10)
    bool full() const;                                             //(11)
    void clear();                                                  //(12)
    void serialize(std::ostream & os) const;                       //(13)

  private:
    Node * m_top;
};
```

The following references a Node class that holds elements of type class DataType. You will also have to template this class as well, and preferably put the necessary declarations and implementations in the same header file as the NodeStack templated class:

```
class Node{
  friend class NodeStack;

  public:
    Node();
    Node(const DataType & data, Node * next = NULL);

    DataType & data;
    const DataType & data() const;

  private:
    Node * m_next
    DataType m_data;
};
```

The **NodeStack** Class will contain the following **private** data members:

- **m_top,** a templated (you need to template this) Node Pointer type, pointing to the top (most recently inserted) element of the Stack.

,will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new NodeStack object with no elements (Nodes).
- **(2) Parametrized Constructor** – will instantiate a new NodeStack object, which will be dynamically allocated at instantiation to hold size_t count number of elements (Nodes), all of them initialized to be equal to the parameter value.
- **(3) Copy Constructor** – will instantiate a new NodeStack object which will be a separate copy of the data of the **other** NodeStack object which is getting copied. *Note*: Consider why now you do need to implement this.

- **(4) Destructor** – will destroy the instance of the NodeStack object. *Note*: Consider why now you do need to implement this.
- **(5) operator=** will assign a new value to the calling NodeStack object, which will be an exact copy of the rhs object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Note*: Consider why now you do need to implement this.
- **(6a,6b) top** returns a Reference to the top element of the stack. *Note*: Since it returns a Reference, before calling this method the user must ensure that the stack is not empty.
- **(7) push** inserts at the top of the stack an element of the given value. *Note*: No imposed maximum size limitations exist for the Node-based stack variant.
- **(8) pop** removes the top element of the stack. *Note*: Checking if the stack is empty prior to popping an element still makes sense.
- **(9) size** will return the current size of the NodeStack.
- **(10) empty** will return a bool, true if the NodeStack is empty.
- **(11) full** will return a bool, true if the NodeStack is full. *Note*: Kept for compatibility, should always return false.
- **(12) clear** performs the necessary actions, so that after its call the NodeStack will become empty.
- **(13) serialize** outputs to the parameter ostream os the complete content of the calling NodeStack object (ordered from top to bottom).

as well as a non-member overload for:

- **(i) operator<<** will output (to terminal or file) the complete content of the nodeStack object passed as a parameter.

You will create the necessary ArrayStack.h and NodeStack.h files that contain the **templated class declarations and implementations**. You should also create a source file proj10.cpp which will be a test driver for your classes.

Templates are special! Do not try to separate declaration & implementation in header and source files as you were used to doing. Follow the guidelines about a single header file ArrayStack.h and a single NodeStack.h, each holding both declarations and respective implementations.
Suggestion(s): First try to implement the two Stack variants as non-templated classes (will closely resemble a simplified version of Project_9). Create a test driver for your code and verify that it works. Then move on to write template-based generic versions of your classes.

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully on the department machines using Makefile(s), which will be invoking the g++ compiler. Instructions how to remotely connect to department machines are included in the Projects folder in WebCampus.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed Header & Source files, Makefile(s), and project documentation.

Submission Instructions:

- You will submit your work via WebCampus
- Name your code file proj10.cpp
- If you have header file, name it proj10.h
- If you have class header and source files, name them as the respective class (ArrayStack.h NodeStack.h) This source code structure is not mandatory, but advised.
- Compress your:
  1. Source code
  2. Makefile(s)
  3. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  Ex: PA10_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the NoMachine virtual machines or directly on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

Late Submission:

A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.