CS 202 - Computer Science II

Project 10

Due date (FIXED): Wednesday, 4/29/2020, 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 Stack class template, with both an Array-based and a Node-based backend 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 projects. For this Project's requirements, you will have to make the necessary modifications such that your ArrayStack and NodeStack and all their functionalities are generalized class templates.

Array-based Stack Class Template:
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 ARRAY_CAPACITY = 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[ARRAY_CAPACITY];
    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 ARRAY_CAPACITY. 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 ARRAY_CAPACITY.

,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.

Node-based Stack Class Template:
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:

```
class Node{
  friend class NodeStack;
  public:
    Node();
    Node(const DataType & data, Node * next = nullptr);
    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.hpp, Node.hpp, NodeStack.hpp files that contain the **class template declarations and implementations**. You should also create a source file proj10.cpp which will be a test driver for your classes.

Templates are special! As mentioned in class:

a) **Do not separate declaration & implementation in (.h) header and (.cpp) source files as you were used to doing. Follow the guidelines about unified header files ArrayStack.hpp, Node.hpp, NodeStack.hpp, each holding both declarations and respective implementations.**

b) **Do not try to compile a Class Template (.hpp) header file on its own. Instead, use `#include` directives in whichever source file requires it, and compile that source file as usual.**

Treat your class template headers (.hpp) like regular headers in terms of file organization. Place them under locations such as:

```
project_10 ->| include ->|-> ArrayStack ->| ArrayStack.hpp
             |           |
             |           |-> NodeStack  ->| Node.hpp
             |                              | NodeStack.hpp
             ...
```

and there will obviously be no source files to be placed under the `src` location.

The completed project should have the following properties:

- Your code is required to follow the **file organization structure** demonstrated in your Labs, with subfolders for headers, source files, and a final build products location (generated during build).
- Your project's build should be based on a **CMakeLists.txt** script, which will be included with your deliverables.
- It must compile successfully using the g++ compiler on department machines or the provided Xubuntu VM image.
- 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 file structure (with .hpp, .cpp, and .h files, and your CMakeLists.txt). Also, your project documentation file.

REMEMBER: Use debugging tools, GDB and Valgrind, as you were instructed to detect the origin of such memory access violation errors and memory leaks!

IMPORTANT: Creating a build configuration for Debugging:

- When requiring a build with debug symbols, usually you would specify this using the g++ command with the appropriate flag:
  `g++ -g ...`

  But **CMake** provides a convenient functionality for configuring a "standardized" debug build of your project with the required compiler flags and settings automatically handled by CMake. It does so via a configuration option called: `CMAKE_BUILD_TYPE`.
  If you want to enforce configuring your project build to have debug symbols enabled (because you intend to debug it using `gdb` for instance), you may run the cmake configuration command and specify this option as follows:

  ```
  cmake -D CMAKE_BUILD_TYPE=Debug ..
  ```

  Extra: Other possible values are:
  `-D CMAKE_BUILD_TYPE=Release` which enables recommended compiler optimizations to refactor the compiled code to make it more efficient,
  `-D CMAKE_BUILD_TYPE=RelWithDebInfo` which generates a "Release" (optimized) build but retains debug symbols for debugging (be careful with "Release" builds your code is optimized by the compiler and thus refactored).
- It is recommended to use such a configuration together with `gdb` (as demonstrated in your Labs) to trace any dynamic memory management bugs of your code.

The following are a list of restrictions:

- Your code may use the C++11 standard (or any standard higher or lower).

  Note: Usually, you would specify using the g++ command with some flags:
  `g++ -std=c++11 ...`

  But **CMake** provides the functionality of *autodetecting* your system's C++ compiler and generating the Makefiles to invoke the appropriate commands to be used when you eventually `make` your project.
  If you want to enforce configuring your project build to use a particular standard, you either do so everytime you run the cmake configuration command by:

  ```
  cmake -D CMAKE_CXX_STARDARD=11 ..
  ```

  Or you could put a line like the following inside your **CMakeLists.txt** script:

  ```
  set(CMAKE_CXX_STANDARD 11)
  ```

  You do not need to worry about either of these however, and for now just running the usual `cmake ..` for configuration should do the trick.
- No libraries except **`<iostream>`** and **`<fstream>`** and **`<string>`** or **`<cstring>`** / **`<string.h>`** allowed.
- No global variables except `const` ones.
- You are expected to employ code abstraction and reuse by implementing and using functions. The already provided code structure in the project description will be considered sufficient.
- You are expected to implement **`const`** correctness in your program design. This refers to class method qualifications, function parameter qualifications, etc.

Submission Instructions:

- You will submit your work via WebCampus
- Compress your:
  1. Code file structure (containing Source code files, Header files, CMakeLists.txt)
  2. Documentation

  Do not include executable or library files, nor any build, devel, or other non-required folders.
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  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 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.

Instructions to remotely test your project configuration and build on the ECC systems:

a) Download your Webcampus submission on your local computer. Let's say you submitted a file named **PAx_Smith_John.zip**, and you now downloaded it into your **Downloads** folder.

b) Navigate to that directory using your terminal, and check the file you downloaded is there.
   a. On **Ubuntu** you can open a terminal with Ctrl+Alt+T and then do:
      ```
      cd Downloads
      ls -al
      ```
   b. On a **Mac** you can open Spotlight and type "Terminal" and hit Enter, then do:
      ```
      cd Downloads
      ls -al
      ```
   c. On **Windows** in your Start Menu type "cmd" and click on Command Prompt, then:
      ```
      cd Downloads
      dir
      ```

c) Remotely copy your submission file from you local Downloads folder to your CSE Ubuntu user account inside its home/$USER folder.
   ```
   scp FILENAME NETID@ubuntu.cse.unr.edu:/nfs/home/NETID/FILENAME
   ```
   For example if the user NetID is **jsmith** and the submission file **PA7_Smith_John.zip**:
```
scp PAx_Smith_John.zip jsmith@ubuntu.cse.unr.edu:/nfs/home/jsmith/PAx_Smith_John.zip
```

d) Login to your CSE Ubuntu user account.
   ```
   ssh NETID@ubuntu.cse.unr.edu
   ```
   For example if the user NetID is **jsmith**:
   ```
   ssh jsmith@ubuntu.cse.unr.edu
   ```

e) Once you are in, check the contents to verify that you have successfully transferred the file:
   ```
   ls -al
   ```

f) Unzip the file into a folder with the same name:

a. If it is a **.zip** file then:
   `unzip -o FILENAME.zip -d FILENAME`
   Example:
   `unzip -o PAx_Smith_John.zip -d PAx_Smith_John`
b. If it is a **tar.gz** file then:
   `mkdir FILENAME`
   `tar -xzvf FILENAME.tar.gz -C FILENAME`
   Example:
   `mkdir PAx_Smith_John`
   `tar -xzvf PAx_Smith_John.tar.gz -C PAx_Smith_John`

g) The above will create a folder with the same name as your submission file, which will contain the unzipped content. Enter the directory and execute the known configuration and build sequence:

```
cd PAx_Smith_John
mkdir build
cmake ..
make
```