CS 202 - Computer Science II

Project 9

Due date (FIXED): Wednesday, 4/22/2020, 11:59 pm

Objectives: The main objectives of this project are to test your ability to create and use queue-based dynamic data structures. 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 Queue class, both an Array-based and a Node-based variant. A Queue is a First In First Out (FIFO) data structure. A Queue exclusively inserts data at the back (**push**) and removes data from the front (**pop**). The Queue's **front** data member points to the first inserted element (the front one), and the **back** data member points to the last (the rear one).

Static *Circular Array*-based Queue:
The following ArrayQueue.h file extract is used to explain the required specifications for the class (it implements a Queue handling DataType objects):

```
const size_t ARRAY_CAPACITY = 1000;
class ArrayQueue{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const ArrayQueue & arrayQueue);

  public:
    ArrayQueue();                                                         //(1)
    ArrayQueue(size_t count, const DataType & value);                     //(2)
    ArrayQueue(const ArrayQueue & other);                                 //(3)
    ~ArrayQueue();                                                        //(4)

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

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

    DataType & back();                                                   //(7a)
    const DataType & back() const;                                       //(7b)

    void push(const DataType & value);                                    //(8)
    void pop();                                                           //(9)

    size_t size() const;                                                //(10)
    bool empty() const;                                                 //(11)
    bool full() const;                                                  //(12)
    void clear();                                                       //(13)
    void serialize(std::ostream & os) const;                            //(14)

  private:
    DataType m_array[ARRAY_CAPACITY];
    size_t m_front;
    size_t m_back;
    size_t m_size;
};
```

IMPORTANT note about Static *Circular Array*-based Queue:

The required implementation is to implement a *Circular Array*-backed Queue. This means that you have to study the modified m_front and m_back update rules and any other pertinent requirements, as discussed and presented in the corresponding Lecture.
Note: Since the container maximum capacity is statically allocated, you do not need to concern yourselves with Dynamic Memory Allocation for this class.

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

- **m_array,** the array that holds the data. *Note*: This will be a Statically Allocated array that can hold ARRAY_CAPACITY objects, be careful *not* to treat is as a Dynamically Allocated one.
- **m_size**, a size_t, keeps track of how many elements are currently stored in the Queue (and therefore considered valid). *Note*: This cannot exceed ARRAY_CAPACITY.
- **m_front,** a size_t, with the respective m_array index of the front element of the Queue.
- **m_back**, a size_t, with the respective m_array index of the back element of the Queue.

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

- **(1) Default Constructor** – will instantiate a new Queue object with no valid data.
- **(2) Parametrized Constructor** – will instantiate a new Queue 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 Queue object which will be a separate copy of the data of the **other** Queue object which is getting copied. *Note*: Consider whether you actually need to implement this.
- **(4) Destructor** – will destroy the instance of the Queue object. *Note*: Consider whether you actually need to implement this.
- **(5) operator=** will assign a new value to the calling Queue 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) front** returns a Reference to the front element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(7a,7b) back** returns a Reference to the back element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(8) push** inserts at the back of the Queue an element of the given value. *Note*: Since m_size can never exceed ARRAY_CAPACITY, checking if the Queue is full prior to pushing a new element makes sense.
- **(9) pop** removes from the front element of the Queue. *Note*: Since m_size is an unsigned value, checking if the Queue is empty prior to popping an element makes sense.
- **(10) size** will return the size of the current Queue.
- **(11) empty** will return a bool, true if the Queue is empty (m_size==0).
- **(12) full** will return a bool, true if the Queue is full (m_size==ARRAY_CAPACITY).
- **(13) clear** performs the necessary actions, so that after its call the Queue will be semantically considered empty.
- **(14) serialize** outputs to the parameter ostream os the complete content of the calling Queue object.

as well as a non-member overload for:

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

Node-based Queue:
The following NodeQueue.h file extract is used to explain the required specifications for the class (the corresponding Node class handles DataType objects, as per your previous Project) :

```
class NodeQueue{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const NodeQueue & nodeQueue);
  public:
    NodeQueue();                                                         //(1)
    NodeQueue(size_t size, const DataType & value);                      //(2)
    NodeQueue(const NodeQueue & other);                                  //(3)
    ~NodeQueue();                                                        //(4)
    NodeQueue& operator= (const NodeQueue & rhs);                        //(5)
    DataType & front();                                                 //(6a)
    const DataType & front() const;                                     //(6b)
    DataType & back();                                                  //(7a)
    const DataType & back() const;                                      //(7b)
    void push(const DataType & value);                                   //(8)
    void pop();                                                          //(9)
    size_t size() const;                                                //(10)
    bool empty() const;                                                 //(11)
    bool full() const;                                                  //(12)
    void clear();                                                       //(13)
    void serialize(std::ostream & os) const;                            //(14)
  private:
    Node * m_front;
    Node * m_back;
};
```

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

- **m_front,** a Node Pointer type, pointing to the front element of the Queue.
- **m_back**, a Node Pointer type, pointing to the back element of the Queue.

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

- **(1) Default Constructor** – will instantiate a new Queue object with no elements (Nodes).
- **(2) Parametrized Constructor** – will instantiate a new Queue object, which will dynamically allocate 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 Queue object which will be a separate copy of the data of the **other** Queue object which is getting copied. *Note*: Consider why now you do need to implement this.
- **(4) Destructor** – will destroy the instance of the Queue object. *Note*: Consider why now you do need to implement this.
- **(5) operator=** will assign a new value to the calling Queue 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) front** returns a Reference to the front element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.

- **(7a,7b) back** returns a Reference to the back element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(8) push** inserts at the back of the Queue an element of the given value. *Note*: No imposed maximum size limitations exist for the Node-based Queue variant.
- **(9) pop** removes from the front element of the Queue. *Note*: Checking if the Queue is empty prior to popping an element makes sense.
- **(10) size** will return the size of the current Queue.
- **(11) empty** will return a bool, true if the Queue is empty.
- **(12) full** will return a bool, true if the Queue is full. *Note*: Kept for compatibility, should always return false.
- **(13) clear** performs the necessary actions, so that after its call the Queue will be empty. *Note*: Consider now how you will implement this in contrast to the other queue variant.
- **(14) serialize** outputs to the parameter ostream os the complete content of the calling Queue object.

as well as a non-member overload for:

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

You will create all necessary files according to the above requirements, apart from the DataType.h and DataType.cpp files which are provided fully implemented. You should also create a source file proj9.cpp which will be a test driver for your classes.

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 .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: PA9_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
```