CS 202 - Computer Science II

Project 7

Due date (FIXED): Monday, 4/24/2017, 11:59 pm

Objectives: The main objectives of this project are to test your ability to create and use queue-based dynamic data structures, and generalize your code using templates. A review of your knowledge on working with lists, as well as manipulating dynamic memory, classes, pointers and iostream to all extents, is also included.

Description:
For this project you will create a templated Queue class, with 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).

As per the previous Project, **Pointer-based** and **Bracket-notation** ([ ]) in order to perform **array manipulation** are both **allowed**. The following provided specifications refer to Queues 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 ArrayQueue and NodeQueue and all their functionalities are generalized templated classes.

Array-based Queue:
The following header file extract is used to explain the required specifications for the class. This only refers an Array-based Queue 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:

```
…
const int ARRAY_MAX = 1000;
class ArrayQueue{
  public:
    ArrayQueue();                                                       //(1)
    ArrayQueue(int size, const DataType& value);                        //(2)
    ArrayQueue(const ArrayQueue& other);                                //(3)
    ~ArrayQueue();                                                      //(4)
    ArrayQueue& operator= (const ArrayQueue& other_arrayQueue);        //(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)
    int size() const;                                                  //(10)
    bool empty() const;                                                //(11)
    bool full() const;                                                 //(12)
    void clear();                                                      //(13)
    friend std::ostream& operator<<(std::ostream& os,                  //(xx)
                                    const ArrayQueue& arrayQueue);
  private:
    DataType m_array[ARRAY_MAX];
    int m_front;
```

```
     int m_back;
     int m_size;
};
…
```

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

- **m_array,** the array that holds the data. Note: Here it is given to hold DataType class objects and have a maximum size of ARRAY_MAX. For this Project's requirements, both of these parameters will have to be determined via Template Parameters.
- **m_size**, an int, keeps track of how many elements are currently stored in the Queue (& considered valid). Note: This should not exceed ARRAY_MAX.
- **m_front,** an int, with the respective m_array index of the front (first) element of the Queue.
- **m_back**, an int, with the respective m_array index of the back (last) 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 int size 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 other_arrayQueue 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 (therefore, it also has to be templated according to the specifications). Note: Since m_size can never exceed ARRAY_MAXSIZE, 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 can be less than 0, 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), and false otherwise.
- **(12) full** will return a bool, true if the Queue is full (m_size==ARRAY_MAXSIZE), and false otherwise.
- **(13) clear** perform the necessary actions, so that after its call the Queue will be considered empty.

as well as a friend function:

- **(xx) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the complete content of the calling Queue object. Note: In order to do that, and since accessing is only on the front element-basis, it will need to sequentially pop elements until the Queue's final element (the original back) is reached. **(A templated implementation of this is not required for 100pt grade).**

Final Note about Array-based Queue:

The required implementation will be a wrap-around Queue. This means that:

a) Pushing an element will move the back by one (m_back = (m_back+1) % ARRAY_MAXSIZE) as seen in Lecture 21 and increase the size by 1.
b) Popping an element will move the front by one (m_front = (m_front+1) % ARRAY_MAXSIZE) as seen in Lecture 21 and decrease the size by 1.
c) Keeping track of the count of elements in the Queue (through m_size) as seen in Lecture 21 is necessary.

Node-based Queue:

The following header file extract is used to explain the required specifications for the class. This only refers a Node-based Queue 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 NodeQueue{
  public:
    NodeQueue();                                                   //(1)
    NodeQueue(int size, const DataType& value);                    //(2)
    NodeQueue(const NodeQueue& other);                             //(3)
    ~NodeQueue();                                                  //(4)
    NodeQueue& operator= (const NodeQueue& other_nodeQueue);       //(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)
    int size() const;                                              //(10)
    bool empty() const;                                            //(11)
    bool full() const;                                             //(12)
    void clear();                                                  //(13)
    friend std::ostream& operator<<(std::ostream& os,              //(xx)
                                    const NodeQueue& nodeQueue);
  private:
    Node *m_front;
    Node *m_back;
};
…
```

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 NodeQueue templated class:

```
…
class Node{
  public:
    Node();
    Node(const DataType& data, Node* next = NULL);
    DataType& getData;
    const DataType& getData() const;
  friend class NodeQueue;
  private:
    Node* m_next
    DataType m_data;
};
```

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

- **m_front,** a templated (you need to template this) Node Pointer type, pointing to the front (first) element of the Queue.
- **m_back**, a templated (you need to template this) Node Pointer type, pointing to the front (first) 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 int size 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 again why you need to implement this.
- **(4) Destructor** – will destroy the instance of the Queue object. Note: Consider again why you need to implement this.
- **(5) operator=** will assign a new value to the calling Queue object, which will be an exact copy of the other_nodeQueue object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. Note: Consider again how you 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 (therefore, it also has to be templated according to the specifications). 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, and false otherwise.
- **(12) full** will return a bool, true if the Queue is full, and false otherwise. Note: Kept for compatilbility, should always return false.
- **(13) clear** perform the necessary actions, so that after its call the Queue will be considered empty. Note: Consider again how you need to implement this.

as well as a friend function:

- **(xx) operator<<** will output (to terminal or file depending on the type of ostream& os object passed as a parameter to it) the complete content of the calling Queue object. Note: In order to do that, and since accessing is only on the front element-basis, it will need to sequentially pop elements until the Queue's final element (the original back) is reached. **(A templated implementation of this is not required for 100pt grade).**

Final Note about Node-based Queue:

The required implementation will be a linear linked-list. This means that:

a) It will have a front pointer to the first element of the Queue, as seen in Lecture 21.
b) It will have a back pointer to the last element of the Queue, as seen in Lecture 21.
c) You may modify the specifications to add an auxiliary variable to keep track of the count of elements in the Queue (e.g. an int m_size) is you wish.

You will create the necessary ArrayQueue.h and NodeQueue.h files that contain the **templated class declarations and implementations**. You should also create a source file proj7.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 ArrayQueue.h and a single NodeQueue.h, each holding both declarations and respective implementations.
Suggestion(s): First try to implement the two Queue variants as non-templated classes (will closely resemble a simplified version of Project_6). Create a test driver for your code and verify that it works. Only then move on to write template-based generic versions of your two 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 proj7.cpp
- If you have header file, name it proj7.h
- If you have class header and source files, name them as the respective class (ArrayQueue.h NodeQueue.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: PA7_Smith_John.zip

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.