CS 302
Homework, Asst. #04

Purpose:	Learn concepts regarding linked lists, stacks, and queues.
Due:	Tuesday (9/23) → Must be submitted on-line before class.
Points:	Part A → 75 pts, Part B → 50 pts

<u>**Assignment:**</u>

<u>**Part A:**</u>

Design and implement a series of C++ template classes to implement a stack and queue as follows:

- ***linkedStack*** to implement the stack using a linked list
- ***linkedQueue*** to implement the queue with a linked list

A main will be provided that can be used to test the ***linkedStack*** and ***linkedQueue*** classes.

Each link in the ***linkedStack*** and ***linkedQueue*** should hold one second of audio at 32*kh*. As such, we will be us using a custom data structure for this assignment. Each link will have an array, and when full, a new link will be added.

When the ***linkedStack*** and ***linkedQueue*** classes are fully functional, create a main program using the a stack and queue to reverse a sound recording. This will be useful for detecting satanic messages in music[1]. Specifically, your program should read the data file, reverse the recording, and place the output into a new data file.

The input and output file names should be read from the command line. If neither file name is not provided on the command line, the program should display a usage message detailed what should be entered (i.e., **`Usage: ./reverse <inputFile> <outputFile>`**). If either file name is provided, but can not be opened, an error message should be displayed (i.e, **`Error, unable to open input file: soundClip.ogg`**).

<u>**Part B:**</u>

When completed, execute the program on a series of different sounds clips.

Create and submit a write-up (open document or word format) with a write-up not too exceed 200 words including the following:

- Name, Assignment, Section
- Compare using an array only and linked list only for this assignment.
    - Include advantages and disadvantages of each implementation approach.
    - Comparison of memory usage between array and linked implementations.
- State advantages of using the custom data structure for this assignment.
- Big-O for basic stack and queue operations (i.e., each key function).
    - Key functions include *push()*, *pop()*, *top()*, and *addItem()*, *deleteItem()*.

1   For more information, refer to: http://en.wikipedia.org/wiki/Backmasking

Digital Sound Information

Computer audio files are digital versions of the original analog sound. The analog sound is sampled several thousand of times a second and that information is stored as numbers. This process is referred to as Analog to Digital Conversion (i.e., 'A-to-D').

The Unix utility program, **sox**[2] (Sound eXchange) allows us to convert from standard audio formats (such as .wav and .ogg) to data files (human readable, text format). We will use the **sox** to convert a standard sound file into a data file. The program will read the data file and create a new data file. We will then use **sox** to convert the new data file back into a standard audio file. The standard audio file can be played with most music players.

The .dat data file is a text file containing the digital version of the sound information in human readable format. The file starts with a a couple of lines describing the sample rate and the number of channels encoded. The remaining lines contain a time reference and sample value (between -1.0 and +1.0). Below is the beginning of a sample file:

```
; Sample Rate 32000
; Channels 1
              0                0
      3.125e-05                0
       6.25e-05                0
      9.375e-05                0
       0.000125                0
     0.00015625                0
```

Notice that for this example, the numbers in the first column increase by 1/32000 each step. This is because the sample rate is 32kHz. *Note*, CDs are typically 44.1kHZ and multiple channels.

Reversing an Sound File

Below is a small excerpt of a single channel sound file. The numbers in the first column are the time reference (in seconds). The numbers in the second column are the sample value.

```
     0.053    0.00054931641
0.05303125   -0.00021362305
 0.0530625   -0.00091552734
0.05309375    -0.0012512207
  0.053125    -0.0011291504
0.05315625   -0.00091552734
 0.0531875   -0.00082397461
```

To “reverse” the audio, we will reverse the numbers in the second column (and only the numbers in the second column). We will use a stack and queue to accomplish this function. *Note*, the original two header lines (see example in digital sound information section) must be maintained.

Sound Exchange:

To install the **sox** utility, type:

```
ed-vm% sudo apt-get install sox
```

You will be prompted for the administrator password. *Note,* ed-vm% is the prompt for my computer. Your's will be different. Once the **sox** utility is installed, to convert the audio file *vogonMessage.ogg* into a data file;

```
ed-vm% sox vogonMessage.ogg vogonMessage.dat
```

2 For more information, refer to: http://en.wikipedia.org/wiki/SoX

The first file is the input and the second file is the output. Thus, to convert a data file into an audio file;

```
ed-vm% sox revVogonMessage.dat revVogonMessage.ogg
```

The file formats are determined based on the extensions.

Processing Overview

The general process is outlined as follows:

1. Using **sox**, convert the standard audio file (.wav, .ogg, etc.) into a .dat data file
2. Using your program
    1. read the input file (.dat format)
    2. reverse the recording data
    3. create the output file (.dat format)
3. Using **sox**, convert the .dat data file into a standard audio file (.wav, .ogg, etc.).

Once the final audio file is created, you can listen to it using an audio player of your choice (and then check for satanic messages). For simplicity we will use single channel recordings and the Ogg[3] format (.ogg) which an open format.

Script File

You will need to create a bash script[4] file to perform the conversion from the audio file to data format, execute your program, convert the data file back to an audio file. The script file should accept the input file name, use **sox** to convert to a data file, use your program to create a new data file (with the sound reversed) and again use **sox** to convert the data file into an audio file.

The script file should require the input file (1st) and output file (2nd) on the command line. This includes verifying the file names were provided and that the input file exists. Using **sox**, the script should create the data file, use your program to create the new data file, and use **sox** to create the final sound file. The data files should be removed when done. Refer to the example output for use of the script file.

Make File:

You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executables. The makefile should be able to build both executables; one for the testing (i.e., testDataStructures) and the other for the audio reverse program (i.e., reverse).

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

3 For more information, refer to: http://en.wikipedia.org/wiki/Ogg
4 For more information, refer to: http://www.gnu.org/software/bash/manual/

Class Descriptions

- Linked List Stack Class
  The linked stack class will implement a stack with a linked list including the specified functions. We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType dataSet[SIZE];
      int   top;
      nodeType<myType> *link;
};
```

  In addition, include the following: `#define    SIZE   32000`

  The *dataSet[]* array should hold SIZE entries. Only when the array is filled a new element linked list should be created. The ***top*** variable should be used to track the stack top for the current node.

linkedStack<myType>
-nodeType<Type> *stackTop
-count: int
+linkedStack()
+~linkedStack()
+isEmptyStack() const: bool
+initializeStack(): void
+stackCount(): int
+push(const myType& newItem): void
+top() const: myType
+pop(): void

Function Descriptions

- The *linkedStack()* constructor should initialize the stack to an empty state (*stackTop*=NULL).
- The *~linkedStack()* destructor should delete the stack (including releasing all the allocated memory).
- The *initializeStack()* function will create and initialize a new, empty stack.
- The *isEmptyStack()* function should determine whether the stack is empty, returning *true* if the stack is empty and *false* if not.
- The *stackCount()* function should return the count of elements on the stack.
- The *push(const Type& newItem)* function will add the passed item to the top of the stack.
- The *top()* function will return the current top of the stack.
- The *pop()* function will remove the top item from the stack (and return nothing). If the stack is empty, nothing should happen.

- Linked List Queue Class
  The linked queue class will implement a queue with a linked list including the specified functions. We will use the following node structure definition.

```
template <class myType>
struct queueNode
{
      myType dataSet[SIZE];
      int   front, back;
      queueNode<myType> *link;
};
```

  In addition, include the following: `#define    SIZE   32000`

  The dataSet[] array should hold SIZE entries. Only when the array is filled a new element linked list should be created. The ***front*** and ***back*** variables should be used to track the current front and back for the current node.

linkedQueue<myType>
-queueNode<myType> *queueFront
-queueNode<myType> *queueRear
-count: int
+linkedQueue()
+~linkedQueue()
+isEmptyQueue() const: bool
+initializeQueue(): void
+addItem(const myType& newItem): void
+front() const: myType
+back() const: myType
+deleteItem(): void
+queueCount(): int
+printQueue(): void

Function Descriptions - Queue

- The *initializeQueue()* function will create and initialize a new, empty queue.
- The *isEmptyQueue()* function should determine whether the queue is empty, returning *true* if the queue is empty and *false* if not.
- The *addItem(const Type& newItem)* function will add the passed item to the back of the queue.
- The *front()* function will return the current front of the queue.
- The *back()* function return the current back of the queue.
- The *deleteItem()* function will remove the front item from the queue (and return nothing). If the queue is empty, nothing should happen.
- The *linkedQueue()* constructor should initialize the queue to an empty state (*queueFront* = NULL and *queueRear* = NULL).
- The *~linkedQueue()* destructor should delete the queue (including releasing all the allocated memory).
- The *printQueue()* function should print the current elements of queue.
- The *queueCount()* function should return the current count of elements in the queue.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

<u>Example Execution:</u>

Below is an example program execution for the main.

```
ed-vm% ./testDataStructures
-----------------------------------------------------------------
CS 302 - Assignment #4
Basic Testing for Linked Stacks and Queues Data Structures

*****************************************************************
Test Queue Operations - Integers:

Queue 0 (original): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Queue 0 (count): 20

Queue 1 (odds):     1 3 5 7 9 11 13 15 17 19
Queue 1 (count): 10

Queue 2 (evens):    2 4 6 8 10 12 14 16 18 20
Queue 2 (count): 10

-----------------------------------------------------------------
Test Queue Operations - Floats:

Queue 3 (floats, original): 1.5 2.5 3.5 4.5 5.5 6.5 7.5
Queue 3 (floats, modified): 1.5 2.5 3.5 4.5 5.5 6.5 7.5

Queue 3 (count): 7
Queue 3 (first item): 1.5
Queue 3 (last item): 7.5

-----------------------------------------------------------------
Test Queue Operations - Many Items:

Many items, test passed.


*****************************************************************
Test Stack Operations - Reversing:

Original List:     2 4 6 8 10 12 14 16 18 20
Reverse List:      20 18 16 14 12 10 8 6 4 2
Copy A (original): 2 4 6 8 10 12 14 16 18 20

-----------------------------------------------------------------
Test Stack Operations - Doubles:

Original List:     1.1 3.3 5.5 7.7 9.9 11.11 13.13 15.15
Reverse List:      15.15 13.13 11.11 9.9 7.7 5.5 3.3 1.1
Copy A (original): 1.1 3.3 5.5 7.7 9.9 11.11 13.13 15.15

-----------------------------------------------------------------
Test Stack Operations - Many Items:

Many items, test passed.

-----------------------------------------------------------------
Game Over, thank you for playing. ed-vm%
ed-vm%
ed-vm% ./reverse
```

```
Usage: ./reverse <inputFileName> <outputFileName>
ed-vm%
ed-vm% ./reverse none none none
Error, must provide input and output file names.
ed-vm%
ed-vm% ./reverse null.txt tmp.dat
Error, unable to open input file: null.txt
ed-vm%
ed-vm%
ed-vm% sox goodnews.ogg goodnews.dat
ed-vm% ./reverse goodnews.dat good.dat
ed-vm% sox good.dat good.ogg
ed-vm% play good.ogg
ed-vm%
ed-vm%
ed-vm% ./rev goodnews.ogg
Error, must provide output file name.
ed-vm%
ed-vm%
ed-vm% ./rev goodnews.ogg good.ogg
ed-vm% play good.ogg
ed-vm%
```