

CS 202 – Assignment #10

Purpose: Learn concepts regarding linked lists, stacks, and queues.
Points: 200 pts

Assignment:

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.



Paul: "If you play *Maybe I'm Amazed* backwards, you'll hear a recipe for a really ripping lentil soup."

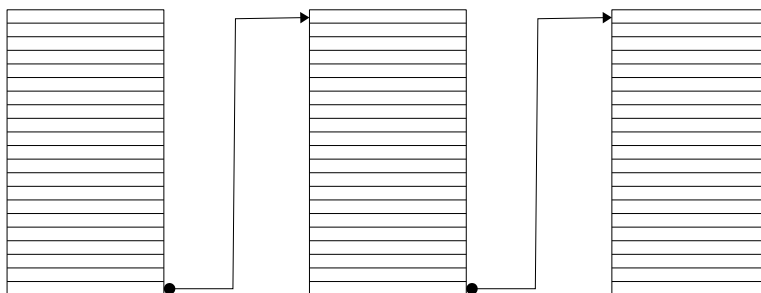
Each link in the ***linkedStack*** and ***linkedQueue*** should hold one second of audio at 32kh. As such, we will be using a custom data structure for this assignment. Each link will have an array, and when full, a new link will be added.

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 music¹. 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**).

Linked List Implementation

Each node in the linked list should hold **size** number of elements. For example, if size=100, each link should hold 100 elements. Only when a 101st element is added, should a new link should be created. And, the next link would not be created until the 201st element is added.



When all items are removed from a specific link, only then should the link be deleted.

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

Submission:

- All files must compile and execute on Ubuntu and compiler with C++11.
 - Submit: **reverse.cpp**, **linkedQueue.h**, **linkedStack.h**, **makefile**
- Submit source files list on assignment page.
- Once you submit, the system will score the project and provide feedback.
 - If you do not get full score, you can (and should) correct and resubmit.
 - You can re-submit an unlimited number of times before the due date/time.
- Late submissions will be accepted for a period of 24 hours after the due date.

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 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 install sox
```

You will be prompted for the administrator password. *Note*, the **ed-vm%** is the prompt for my computer. Yours 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
```

The first file (**vogonMessage.ogg**) is the input and the second file (**vogonMessage.dat**) 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.).
4. Optionally, play one or both files (i.e., **play file.ogg**).

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). The command line “play” utility is installed with **sox**. For simplicity we will use single channel recordings and the Ogg³ format (.ogg) which an open (non-proprietary) format.

A script file is provided to perform the outlined processing.

Main File

A template for the main file, **reverse.cpp**, is provided. You will need to complete the noted sections (command line argument handling and audio data file read and write operations). The error message strings and file output statements are included in the template. You will need to ensure the appropriate message are printed at the appropriate times. *Note*, regardless of the condition, always use “**return EXIT_SUCCESS**” even if an error occurred.

Make File:

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

```
make
```

Which should create the executable file, *reverse*. You may refer to previous assignments for examples of make files.

3 For more information, refer to: <http://en.wikipedia.org/wiki/Ogg>

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;
    unsigned int top;
    nodeType<myType> *link;
};
```

The *dataSet[]* array should hold **size** entries. Only when the array is filled a new element linked list should be created.

linkedStack<myType>
-nodeType<Type> *stackTop
-size: unsigned int
-count: unsigned int
-ARR_MIN = 5: static constexpr unsigned int
-ARR_MAX = 96000: static constexpr unsigned int
-ARR_DEFAULT = 22050: static constexpr unsigned int
+linkedStack(unsigned int=ARR_DEFAULT)
+~linkedStack()
+isEmptyStack() const: bool
+initializeStack(): void
+stackCount(): unsigned 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=nullptr*). If the passed link array size is not valid, set the size to the default (ARR_DEFAULT).
- 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 (across all links).
- The *push(const Type& newItem)* function will add the passed item to the top of the stack. The **top** variable should be used to track the stack top for the current node. If current node array is full, a new node with a new node array must be created to hold the new item.
- The *pop()* function will remove the top item from the stack (and return nothing). If the stack is empty, nothing should happen and no error message. If the item removed is the last item from the node array, the node link must be deleted along with the node array.
- The *top()* function will return the current top of the stack **without** removing it.

- 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;
    unsigned int    front, back;
    queueNode<myType> *link;
};
```

The *dataSet[]* array should hold **size** entries. Only when the array is filled a new element linked list should be created.

linkedQueue<myType>
-queueNode<myType> *queueFront
-queueNode<myType> *queueRear
-size: unsigned int
-count: unsigned int
-ARR_MIN = 5: static constexpr unsigned int
-ARR_MAX = 96000: static constexpr unsigned int
-ARR_DEFAULT = 22050: static constexpr unsigned int
+linkedQueue(unsigned int=ARR_DEFAULT)
+~linkedQueue()
+isEmptyQueue() const: bool
+initializeQueue(): void
+addItem(const myType& newItem): void
+front() const: myType
+back() const: myType
+deleteItem(): void
+queueCount(): unsigned int
+printQueue(): void

Function Descriptions - Queue

- The *linkedQueue()* constructor should initialize the queue to an empty state (*queueFront* = NULL and *queueRear* = NULL).
- The *initializeQueue()* function will create and initialize a new, empty queue. If the passed link array size is not valid, set the size to the default (ARR_DEFAULT).
- The *~linkedQueue()* destructor should delete the queue (including releasing all the allocated memory). This includes each node and the node array.
- The *isEmptyQueue()* function should determine whether the queue is empty, returning *true* if the queue is empty and *false* if not.
- The *front()* function will return the current front of the queue **without** removing it.
- The *back()* function return the current back of the queue **without** removing it.
- The *addItem(const Type& newItem)* function will add the passed item to the back of the queue. The **front** and **back** variables should be used to track the **front** and **back** for the current node. If current node array is full, a new node with a new node array must be created to hold the new item.

- The `deleteItem()` function will remove the front item from the queue (and return nothing). If the queue is empty, nothing should happen and no error message provided. If the item removed is the last item from the node array, the node link must be deleted along with the node array.
- The `printQueue()` function should print the current elements of queue.
- The `queueCount()` function should return the current count of elements in the queue across all links.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. *Note, points will be deducted for especially poor style or inefficient coding.*

Example Execution:

Below is an example program execution for the main.

```
ed-vm% ./testStack
-----
CS 202 - Assignment #10
Basic Testing for Linked Stack

*****
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

Original Shorts List:
383 886 777 915 793 335 386 492 649 421 362 27 690 59 763 926 540 426 172 736
211 368 567 429 782 530 862 123 67 135 929 802 22 58 69 167 393 456 11 42
229 373 421 919 784 537 198 324 315 370

Reverse Shorts List:
370 315 324 198 537 784 919 421 373 229 42 11 456 393 167 69 58 22 802 929
135 67 123 862 530 782 429 567 368 211 736 172 426 540 926 763 59 690 27 362
421 649 492 386 335 793 915 777 886 383

Copy A (original):
383 886 777 915 793 335 386 492 649 421 362 27 690 59 763 926 540 426 172 736
211 368 567 429 782 530 862 123 67 135 929 802 22 58 69 167 393 456 11 42
229 373 421 919 784 537 198 324 315 370

-----
Test Stack Operations - Doubles:

Original List:      1.1 3.3 5.5 7.7 9.9 11.11 13.13 15.5 17.1 19.2 21.3 23.4 25 5
27.6 29.7 31.8 33.9 35.1 37.2
Reverse List:      37.2 35.1 33.9 31.8 29.7 27.6 5 25 23.4 21.3 19.2 17.1 15.5
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 15.5 17.1 19.2 21.3 23.4 25 5
27.6 29.7 31.8 33.9 35.1 37.2

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

Many items, test passed.

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

```
ed-vm%
ed-vm%
ed-vm% ./testQueue
```

```
-----
CS 202 - Assignment #10
Basic Testing for Linked Queue
```

```
*****
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 - Shorts:
```

```
Shorts Queue 0 (original):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Shorts Queue 0 (count): 50
```

```
Shorts Queue 1 (odds):
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
Shorts Queue 1 (count): 25
```

```
Shorts Queue 2 (evens):
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
Shorts Queue 2 (count): 25
```

```
-----
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.
```

```
-----
Game Over, thank you for playing.
```

```
ed-vm%
ed-vm%
ed-vm%
ed-vm%
ed-vm%
```

```
ed-vm%
ed-vm%
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%
```