CS 133, Spring 2022
Programming Project #1: Melody (20 points)
Due Thursday, April 14, 2022, 11:59 PM

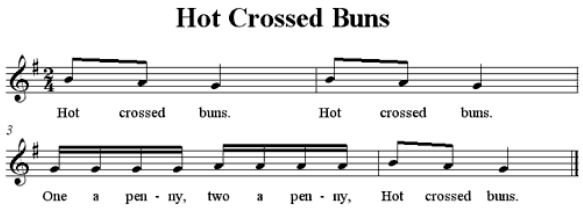
This program focuses on using `stack` and `queue` collections. Turn in files named `Melody.h`, `Melody.cpp` and `song.txt` on the Project section of the course website. You will need the starter code zip from the course website that contains `Note.h`, `Note.cpp`, `melodymain.cpp` and several input files (files in the `res` folder with `.txt` extensions) and audio files (`.mp3`s). This program will require you have a working **Qt** installation.

Background Information about Music:

Music consists of notes which have lengths and pitches. The pitch of a note is described with a letter ranging from A to G. As 7 notes would not be enough to play very interesting music, there are multiple octaves; after we reach note G we start over at A. Each set of 7 notes is considered an octave. Notes may also be accidentals. This means that they are not in the same key as the music is written in. We normally notate this by calling them sharp, flat or natural. Music also has silences which are called rests.

For this assignment we will be representing notes using scientific pitch notation. This style of notation represents each note as a letter and a number specifying the octave it belongs to. For example, middle C is represented as `C4`. You do not need to understand any more than this about scientific pitch notation but you can read more about it here if you are interested: http://en.wikipedia.org/wiki/Scientific_pitch_notation.

What you will do:



You will write a class to represent a song. A song is comprised of a series of notes. It may have repeated sections. As we don't like to have any redundancy, **you may only store one copy of a repeated chunk of notes**.

Music is usually printed like the example on the right. The notes are a series of dots. Their position in relation to the lines determines their pitch and their tops and color, among other things, determine their length. Since it would be difficult for us to read input in this style, we will instead read input from a text file.

```
0.2 C 4 NATURAL false
0.4 F 4 NATURAL true
0.2 F 4 NATURAL false
0.4 G 4 NATURAL false
0.2 G 4 NATURAL true
0.2 A 4 NATURAL false
0.4 R false
0.2 C 5 NATURAL false
0.2 A 4 NATURAL false
…
```

An example input file is shown at the left. Each line in it represents a single note. The first number describes the length of the note in seconds. The letter that follows describes the pitch of the note, using the standard set of letters (A – G) or R if the note is a rest. For notes other than rests, the third item on the line is the octave that the note is in and the following is the note's accidental value. The final piece of information for all notes is `true` if the note is the start or stop of a repeated section and `false` otherwise.

You will implement several member functions in the `Melody` class which will allow you to use `melodymain.cpp` to play your song with mp3 player like functionality. Your melody will be able to play as well as append another melody to itself, reverse and have its tempo changed.

The most challenging part of this assignment is getting melodies to play with repeats correctly. The file above, which contains 4 repeated notes, is equivalent to the repetitive file displayed to the right. When you play the above file you should play it the same as you would play the file to the right.

```
0.2 C 4 NATURAL false
0.4 F 4 NATURAL false
0.2 F 4 NATURAL false
0.4 G 4 NATURAL false
0.2 G 4 NATURAL false
0.4 F 4 NATURAL false
0.2 F 4 NATURAL false
0.4 G 4 NATURAL false
0.2 G 4 NATURAL false
0.2 A 4 NATURAL false
0.4 R false
0.2 C 5 NATURAL false
0.2 A 4 NATURAL false
…
```

Implementation Details:

You will write one class; `Melody`. You must use the C++ standard library's `stack` and `queue`. You must use them as stacks and queues; you may **NOT** use (or construct for yourself) any index based functions, iterators or for-each loops. Your class must have the constructors/member functions below. It must be possible to call the member functions **multiple times in any order** and get the correct results each time. Your `Melody` class should use a `queue` to store the notes in the song. But unless otherwise specified, you may not create any other auxiliary data structures (such as arrays, `list`s, `stack`s, `queue`s, etc) to help you solve any member function below. This includes creating copies of your `queue` member variable.

`Note` Class *(provided by the instructor):*

We have provided you with a class named `Note` that your `Melody` class will use. A `Note` object represents a single musical note that will form part of a melody. It keeps track of the length (duration) of the note in seconds, the note's pitch (A-G, or R if the note is a rest), the octave, and the accidental (sharp, natural or flat). Each `Note` object also keeps track of whether it is the first or last note of a repeated section of the melody.

The `Note` class provides the following functionality that you should use in your program.

Function / Operator	Description
`Note(`**`duration`**`,` **`pitch`**`,` **`octave`**`,` **`accidental`**`,` **`repeat`**`)`	Constructs a `Note` object.
`Note(`**`duration, pitch,repeat`**`)`	Constructs a `Note` object, omitting the octave and accidental values. Used to construct a rest (`R`).
`getAccidental(), getDuration(), getOctave(), getPitch(), isRepeat()`	Returns the state of the note as passed to the constructor.
`play()`	Plays the note so that it can be heard from the computer speakers.
`setAccidental(`**`accidental`**`), setDuration(`**`duration`**`), setOctave(`**`octave`**`), setPitch(`**`pitch`**`), setRepeat(`**`repeat`**`)`	Sets aspects of the state of the note based on the given value.
`<<`	Outputs a text representation of the note.

You can look at the contents of the provided `Note.h` and `Note.cpp` to answer any further questions about how it works.

`Melody` Class *(for you to implement)*:

`Melody``(queue<Note>* song)`
Initializes your melody to store the passed in pointer to a `queue` of `Note`s. You should throw a string exception if the `queue` is `null`.

`double` **`getTotalDuration`**`()`
Returns the total length of the song in seconds. If the song includes a repeated section the length should include that repeated section twice. For example, both sample files shown on page one have length 3.6. You should not loop through the notes every time this member function is called.

`void` **`changeTempo`**`(double tempo)`
Changes the tempo of each note to be `tempo` percent of what it formerly was. Passing a `tempo` of `1.0` will make the tempo stay the same. `tempo` of `2.0` will make each note twice as long. `tempo` of `0.5` will make each note half as long. Keep in mind that when the tempo changes the length of the song also changes. You should throw a `string` exception if the passed in `tempo` is negative.

`void` **`reverse`**`()`

Reverses the order of notes in the song, so that future calls to the `play` member function will play the notes in the opposite of the order they were in before `reverse` was called. For example, a song containing notes `A, F, G,` then `B` would become `B, G, F, A`. You may use one temporary `stack` **or** one temporary `queue` to help you solve this problem.

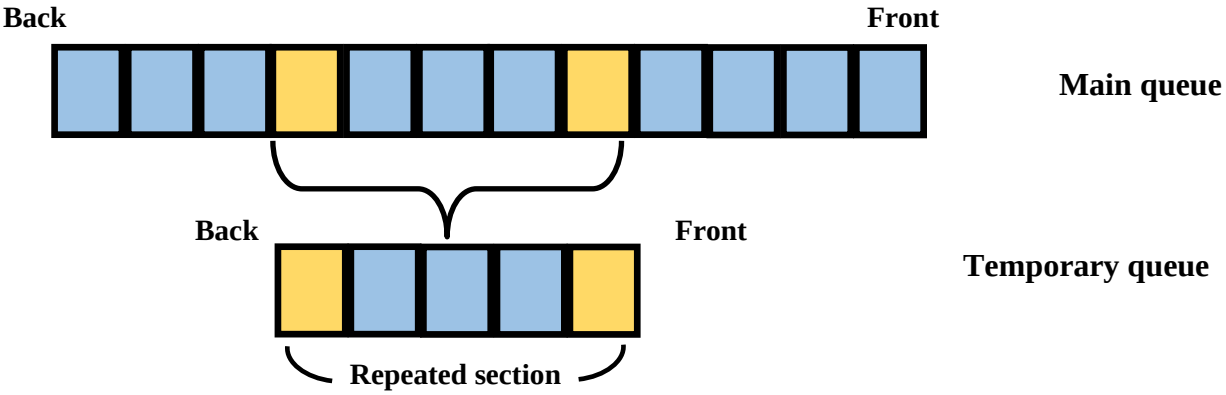
`void` **`append`**`(Melody& other)`
Adds all notes from the given other song to the end of this song. For example, if this song is `A,F,G,B` and the other song is `F, C, D`, your method should change this song to be `A, F, G, B, F, C, D`. The other song should be unchanged after the call. Remember that objects can access the private members of other objects of the same type.

`ostream& operator<<(ostream& out, Melody& song)`

Outputs information about each note to the passed in stream. Each note should be on its own line and output using its << operator. An example file is displayed on page 1. Your << should reflect the changes that have been made to the song by calling other member functions.

`void play()`

Plays the song by calling each note's `play` member function. The notes should be played from the beginning of the `queue` to the end unless there are notes that are marked as being the beginning or end of a repeated section. When the first note that is a beginning or end of a repeated section is found you should create a second `queue`. You should then get notes from the original `queue` until you see another marked as being the beginning or end of a repeat. As you get these notes you should play them and then place them back **in both queues**. Once you hit a second marked as beginning or end of a repeat you should play everything in your secondary queue and then return to playing from the main queue. It should be possible to call this member function multiple times and get the same result.



The yellow blocks represent notes with start or end of a repeat set to `true`. They and the other notes in between them should be moved to a separate queue when played so that they can be repeated.

Creative Aspect (`song.txt`):

Along with your program, submit a file called `song.txt` that contains a song that can be used as input. For full credit, the file should be in the format described above and contain at least 10 notes. It should also be your own work (you may not just turn in one of our sample songs) but you do not have to compose a song yourself. You are welcome to make `song.txt` be a song written by somebody else, such as a lullaby or nursery rhyme or song by your favorite band. This will be worth a small portion of your grade.

Development Strategy and Hints:

We suggest the following development strategy for solving this program:

1. Create the `Melody` class and declare every member function and operator. Leave every function's body blank; if necessary, return a "dummy" value like `nullptr` or `0`. This will allow you to be able to compile the provided code. Remember that the code will initially appear to have a ton of errors. This is because it requires a `Melody` class and that class doesn't exist yet. Once you create the "dummy" version you will be able to compile and run without errors although, of course, the output will be incorrect.
2. Implement the constructor, and the << operator.
3. Implement the `getTotalDuration` and `changeTempo` member functions. You can check the results of the `changeTempo` function by reading in one of the sample files, calling `changeTempo` and then printing it out and checking your output matches what you expected.
4. Write the `reverse` and `append` member functions.
5. Write an initial version of `play()` that assumes there are no repeating sections.
6. Add the `play()` code that looks for repeated sections and plays them twice, as described previously.

You can test the output of your program by running it on various inputs in the `melodymain.cpp` client. You can also use our Output Comparison Tool to see that your outputs match what is expected.

Tips:

There are two provided defines at the top of `Note.h` that you may find useful to change:

- `DEFAULT_WHOLE_NOTE` holds the length your program will play a whole note (note with length 1). Some computers seem to be a lot slower than others at starting and stopping sound. If you find that you are only hearing some of the notes try making this value bigger. Your computer just may not be able to keep up with the default speed.
- `SILENT_MODE` this is commented out in the provided file but you may find it helpful to uncomment it. By default, with it commented out, the `Note play` member function will play a sound on your speakers. It can be hard to debug when you are just hearing your output. There are also likely times and places you may want to work where you either don't have access to speakers or don't want to annoy others around you by making noise. If you uncomment this line the `Note` class will print out a note when it is played instead of playing a sound.

Style Guidelines and Grading:

Part of your grade will come from appropriately using `stack`s and `queue`s. You may only use the `size`, `empty`, `push`, `pop` and `top` / `front` members for both `stack`s and `queue`s. Do not make unnecessary or redundant extra passes over a `queue` when the answer could be computed with fewer passes.

Redundancy is always a major grading focus; avoid redundancy and repeated logic as much as possible in your code.

Properly encapsulate your objects by making member variables `private`. Avoid unnecessary member variables; use member variables to store important data of your objects but not to store temporary values only used within a single function. Initialize member variables in constructors only.

Follow good general style guidelines such as: appropriately using control structures like loops and `if`/`else` statements; avoiding redundancy using techniques such as private helper functions, loops, and `if`/`else` factoring; properly using indentation, good variable names, and proper types; and not having any lines of code longer than 100 characters.

Comment descriptively at the top of your class, each function, and on complex sections of your code. Comments should explain each function's behavior, parameters, return, pre/post-conditions, and any exceptions thrown. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client. Your comments should be written in your own words and not taken verbatim from this document.