CS 3005: Programming in C++

Notes, Frequencies, Time Signatures, and Tempos

Introduction

In this assignment you will add a time signature and a tempo to the musical score, including the ability to configure them from the score editor, and read/write them in score files.

You will also add a note class and a frequency class to support music notation. These classes will be used more in future assignments.

A note represents a musical note, including its frequency and duration. We will specify the frequency by specifying the note from a keyboard, and the octave it belongs to. We will specify the duration by describing whole, half, quarter, etc. notes.

Refer to the class notes for more discussion on octaves, keys, frequency calculations, duration, etc.

Syntax in the `.score` file for time signature and tempo

This format was designed to be read using the C++ standard library's `>>` operator. All values are whitespace delimited. By context, your code should be able to determine whether the next value is a `std::string` or a `double`.

The TIME-SIGNATURE keyword must always be followed by two integers.

The TEMPO keyword must always be followed by one floating point number.

Both of these keywords will be encountered directly inside a SCORE/SCORE-END block.

Sample SCORE file

```
SCORE
  TIME-SIGNATURE 3 4
  TEMPO 120

SCORE-END
```

Notes on the `ScoreReader::readScore` method

- If the method encounters the TIME-SIGNATURE keyword, it should immediately read two integers, and use them to configure the `TimeSignature` field in the `MusicalScore` object.
- If the method encounters the TEMPO keyword, it should immediately a floating point number, and use it to configure the `Tempo` field in the `MusicalScore` object.

Notes on the `ScoreWriter::writeScore` method

- The TIME-SIGNATURE and TEMPO keywords are indented two spaces.
- Their values are on the same line, separated by spaces.
- The << operator should be used to send their values to the score stream.
- TIME-SIGNATURE comes first and TEMPO comes second.
- There is a blank line after the TEMPO line.
- There is not a blank line after SCORE.

Assignment

Here are the *new* commands that are required in the score editor program for this assignment. Previous commands are still required.

Command	Prefixable?	Function	Description
`score-set-time-signature`	no	`setScoreTimeSignatureUI`	Edit the time signature of a score.
`score-set-tempo`	no	`setScoreTempoUI`	Edit the tempo of a score.

Example Session

```
$ ./program-score-editor/score_editor
Choice? menu
Options are:
  # - Skip to end of line (comment).
  comment - Skip to end of line (comment).
  echo - Echo back the arguments given.
  help - Display help message.
  menu - Display help message.
  quit - Terminate the program.
  score-add-envelope - Add envelope to the score.
  score-add-instrument - Add instrument to the score.
  score-add-waveform - Add waveform to the score.
  score-edit-envelope - Edit envelope in the score.
  score-edit-instrument - Edit instrument in the score.
  score-edit-waveform - Edit waveform in the score.
  score-list-envelopes - List envelopes in the score.
  score-list-instruments - List instruments in the score.
  score-list-waveforms - List waveforms in the score.
  score-read - Read score from file.
  score-set-tempo - Edit the tempo of a score.
  score-set-time-signature - Edit the time signature of a score.
  score-write - Write score to score file.

Choice? score-set-time-signature
Beats per bar: 6
Beat value: 8
Choice? score-set-tempo
Beats per minute: 123.4
Choice? score-write
Filename: demo.score
Choice? quit
```

The output file: demo.score.

Programming Requirements

Create `library-score/TimeSignature.{h,cpp}`

`TimeSignature` Class

This class will represent a time signature. That is the number of beats in a bar and the duration of a beat.

`protected` Data Members:

- An `int` the beats per bar.
- An `int` the value of a beat.

`public` Methods:

- `TimeSignature();` Configures the default time signature to $^4/_4$.
- `TimeSignature(const int beats_per_bar, const int beat_value);` Configures the time signature to `beats_per_bar` beats in a bar and `beat_value` beats in a beat.
- `int getBeatsPerBar() const;` Returns the number of beats in a bar.
- `int getBeatValue() const;` Returns the beat value.
- `void setBeatsPerBar(const int beats_per_bar);` Sets the number of beats in a bar, but only if `beats_per_bar` is at least 1.
- `void setBeatValue(const int beat_value);` Sets the beat value, but only if `beat_value` is at least 1.

Update `library-score/MusicalScore.{h,cpp}`

We will add to the `MusicalScore` class by adding the `Instrumentarium` collection. Future assignments will add more to the class.

`MusicalScore` Class

This class will store all of the information for a piece of music.

Data Members:

Add these data member:

- A `TimeSignature` object for storing the music's time signature.
- A `double` object for storing the music's tempo (beats per minute).

`public` Methods:

- `MusicalScore();` Allow the `TimeSignature` object to default construct. Initialize the tempo to 100.
- `MusicalScore(const TimeSignature& time_signature, const double tempo);` Initialize these data members from the parameters.
- `TimeSignature& getTimeSignature();` Return the data member.
- `const TimeSignature& getTimeSignature() const;` Return the data member.
- `double getTempo() const;` Return the data member.
- `void setTempo(const double tempo);` Change the data member, but only if the new tempo is > 0.

Update `library-score-io/ScoreReader.{h,cpp}`

We will update the `ScoreReader` class by adding the ability to read time signatures and tempos.

`ScoreReader` Class

This class will eventually read all of the information for a piece of music from the `.score` file format.

Data Members:

No data members are required.

`public` Methods:

- `void readScore(std::istream& input_stream, MusicalScore& score) const;` Add the ability to recognize the TIME-SIGNATURE and TEMPO keywords. See the description above.

Update `library-score-io/ScoreWriter.{h,cpp}`

We will update the `ScoreWrite` class by adding the ability to write time signatures and tempos.

`ScoreWriter` Class

This class will eventually write all of the information for a piece of music from the `.score` file format.

Data Members:

No data members are required.

`public` Methods:

- `void writeScore(std::ostream& output_stream, const MusicalScore& score) const;` Add the TIME-SIGNATURE and TEMPO keywords. See the description above.

Update `library-commands/score_editor_aux.{h,cpp}`

Commands created for the score editor program will go here. We're adding more this assignment.

Functions:

- `int register_score_editor_commands(ApplicationData& app_data);` Update to add the new commands specified for this assignment.
- `void setScoreTimeSignatureUI(ApplicationData& app);` Asks the user for the values and sets them in the

`ApplicationData` object's `MusicalScore` object. See the description above.
- `void setScoreTempoUI(ApplicationData& app);` Asks the user for the value and set it in the `ApplicationData` object's `MusicalScore` object. See the description above.

Create `library-score/Frequency.{h,cpp}`

`Frequency` Class

This class only has static methods and data members. Note that there is only 1 public method of the class. The rest of the methods are used to pre-calculate the frequencies associated with note names. The public method looks up the frequency using the name.

`protected` Data Members:

- `static double trt;` The twelfth root of two. The data member must be called exactly `trt`.
- `static std::map<std::string, double> smFrequencies;` A map from note names (like "C4") to frequencies. Initialized in the implementation file by the return value of `generateFrequencies()`.

`public` Methods:

- `static double getFrequency(const std::string& note);` If `note` is a key in the map, return the associated frequency. If not, then return 0.0.

`protected` Methods:

- `static double computeFactor(const unsigned int& octaves, const unsigned int& keys);` Returns the scaling factor to move a frequency by `octave` octaves and `keys` positions within an octave. Under normal operation `octave` should be at least 0 and no more than 9 and `keys` should be at least 0 and no more than 11. Your code does not need to check this. The scaling factor is computed as 2 raised to the power of `octave` times the twelfth root of two to the power of `keys`. You should use the `std::pow` function from `<cmath>` to accomplish this.
- `static double moveLeft(const double& frequency, const unsigned int& octaves, const unsigned int& keys);` To move left, we divide a frequency by the factor computed by `computeFactor()`.
- `static double moveRight(const double& frequency, const unsigned int& octaves, const unsigned int& keys);` To move right, we multiply a frequency by the factor computed by `computeFactor()`.
- `static std::map<std::string, double> generateFrequencies();` Generates all of the frequencies from C0 to B9, including all sharps and flats. For each, stores the note in the map with its associated frequency. Each octave has the note names: "C", "C#", "Db", "D", "D#", "Eb", "E", "F", "F#", "Gb", "G", "G#", "Ab", "A", "A#", "Bb", "B". Note that neighboring sharp and flat notes are different names for the same frequency. Octaves 0 through 9 must be computed and stored. Use this process: A4 has frequency 440.0. C0 is computed from moving A4 left the correct number of octaves and positions. All other notes are computed by moving C0 right the correct number of octaves and positions.

Create `library-score/Note.{h,cpp}`

`global` Constants

- `constexpr double SIXTEENTH_NOTE = 0.125/2.0;`
- `constexpr double EIGHTH_NOTE = 0.125;`
- `constexpr double QUARTER_NOTE = 0.25;`
- `constexpr double HALF_NOTE = 0.50;`
- `constexpr double WHOLE_NOTE = 1.00;`

`Note` Class

Stores and manages a note's name and duration. Names are in the format "C#5", a key followed by the octave. Durations are stored in fractions of a whole note.

Data Members:

Data members must be `protected` or `private`. `public` data members are not allowed.

- A `std::string` for the name of the note.

- A `double` for the duration of the note in whole notes.

`public` Methods:

- `Note();` Initializes the name to "" and the duration to 0.
- `Note(const std::string& full_note);` Initializes the name to "" and the duration to 0., then uses the `set` method to configure the name and duration from the input string.
- `Note(const std::string& name, const double& duration);` Initializes the name and duration from the parameters.
- `Note(const std::string& name, const std::string& duration_str);` Initializes the name from the parameter, and the duration to 0. Then uses the `setDuration` method to configure the duration from the parameter.
- `const std::string& getName() const;` Returns the data member.
- `const double& getDuration() const;` Returns the data member.
- `double getFrequency() const;` Uses the `Frequency` class to lookup the numeric frequency from the name and returns it. If the name is not found by the `Frequency` class, returns 0.
- `void setName(const std::string& name);` If the `Frequency` class recognizes the `name`, then sets the name. Otherwise does nothing.
- `void setDuration(const double duration);` If the duration is positive, sets the duration. Otherwise does nothing.
- `void setDuration(const std::string& duration_str);` If the duration is and of the strings `w,h,q,e,s`, then sets the duration from the respective constant. Otherwise does nothing. If the duration string has length 2, and the second character is `.`, then sets the duration to be 1.5 time the duration specified by the letter. If the second character is `t`, then sets the duration to $1/3$ of the duration specified by the letter. If the second character is anything else, then does nothing. If the duration string is longer than 2, does nothing.
- `void set(const std::string& full_note);` Uses `splitString` to split the full note string into parts, then uses `setName` and `setDuration` to configure the data members.

`protected` Methods:

- `void splitString(const std::string& full_note, std::string& name, std::string& duration);` Splits the `full_note` string into `name` and `duration` string. The duration will be the first 1 or 2 characters, and the name will be the rest of the string. The duration is length 2 of the second character is `t` or `.`, otherwise its length is 1.

Free Functions

- `std::ostream& operator<<(std::ostream& output_stream, const Note& note);` This function displays a note to the output stream in the format: "duration-as-a-double name-as-a-string(frequency-as-a-double)". For example "0.25 A4(440.0)" is a quarter note in the A4 key.

Additional Documentation

-

Grading Instructions

To receive credit for this assignment:

- your code must be pushed to your repository for this class on GitHub
- all unit tests must pass
- all acceptance tests must pass
- all programs must build, run, and execute as described in the assignment descriptions.

Extra Challenges (Not Required)

TBA