CS 3005: Programming in C++

Musical Staff

Introduction

In this assignment you will add a musical staff to the project. The musical staff will consist of a set of notes, and the instrument that will be used to render the staff into an audio track.

There will also be some updates to existing classes, and the addition of a staff note, that contains the note information we have already created, an the position in the staff where the note begins.

Syntax in the `.score` file for staff

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`.

TBA

Sample STAFF record

```
STAFF staff-name instrument-name
  0 qC2
  0.25 wBb9
END-STAFF
```

Notes on the `ScoreReader::readStaff` method

- Expects an empty MusicalStaff object as parameter.
- `STAFF` keyword has already been read when called.
- If the instrument name does not identify a valid instrument in the score, immediately return.
- If the instrument exists in the score, set the staff's name and instrument.
- Until `STAFF-END` is found, or end of file is found, read a staff note and add it to the staff.
- A staff note begins with a number. This method has to first read the number as text, and if it is not equal to "END-STAFF" it will need to convert it to a number.
- The note duration and pitch is a single "word" of text that should be parsed by the Note class' set method.

Notes on the `ScoreReader::readScore` method

- The score doesn't have a way to store a staff yet, but the `readScore` method should still recognize the `STAFF` keyword, and call `readStaff` to do the rest of the work.

Notes on the `ScoreWriter::getDurationLetter` method

- This method translates a number into a string that represents the duration of a note in the score.
- The table below specifies the numbers and strings they correspond to.
- Since floating point numbers are difficult to be exact, the number that most closely matches is the one that should be select.

Number	String
1.0/16.0	"s"
1.0/8.0	"e"
1.0/4.0	"q"
1.0/2.0	"h"
1.0/1.0	"w"

Notes on the `ScoreWriter::formatNote` method

- Converts the note's duration to a string and concatenates that with the note's name.

Notes on the `ScoreWriter::writeStaffNote` method

- Each note is on its own line, with 4 leading spaces.
- The note duration and pitch string is generated by the `formatNote` method.

Notes on the `ScoreWriter::writeStaff` method

- `STAFF` is indented by 2 spaces.
- The staff name and instrument name are on the same line as `STAFF`, with single spaces between them.
- Each note is written by calling the `writeStaffNote` method.
- `STAFF-END` is indented by 2 spaces, and followed by a blank line.

Notes on MusicalStaff::getDurationInWholeNotes

- The end of a note is computed as the beginning of the note, plus the duration of the note.
- Finds the largest end of any note in the staff.
- Returns the largest end.

Notes on MusicalStaff::render

- Converting times that are in whole notes to seconds requires multiplying by the seconds per whole note.
- Seconds per whole note is computed from the time signature's beat value * 60.0 / the tempo.
- Configures a track to store the whole staff's notes.
- The track's samples_per_seconds is taken from the parameter.
- The track's duration is computed using the duration in whole notes, multiplied by the seconds per whole note.
- For each note in the staff, uses the instrument's `generateSamples` to fill a temporary audio track.
- The frequency is found from the note.
- The duration in whole notes is found from the note.
- The start of the note in whole notes is found from the staff note.
- Adds the temporary audio track to the main audio track with the `addAt` method. The second parameter to this method is the start time of the note, in seconds. Since `StaffNote` stores the start time in whole notes, this needs to be multiplied by seconds per whole note to get the start time in seconds.

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

TBA

Example Session

```
$ ./program-score-editor/score_editor
TBA
Choice? quit
```

The output file: demo.score.

Programming Requirements

Update `library-audiofiles/AudioTrack.{h,cpp}`

We will add to the `AudioTrack` class by allowing it to add values from another `AudioTrack` object, at a given position in the track. This is the basic operation that will allow us to make an audio track from multiple notes.

`AudioTrack` Class

This class stores digital audio information for one track of sound.

Data Members:

No changes.

`public` Methods:

- `void addAt(const AudioTrack& other_track, double offset_seconds);` Adds the audio data from `other_track` starting at position `offset_seconds` seconds in the track. This is additive. Whatever value is already at these positions, is added to.

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

`StaffNote` Class

This class will represent a note in a staff, including the position in the staff when the note begins.

`protected` Data Members:

- A `Note` object.
- A `double` the number of whole notes since the beginning of the staff that this note begins.

`public` Methods:

- `StaffNote();` defaults to default Note and 0.0 start
- `StaffNote(const Note& note, const double start);` Initializes data members from parameters
- `Note& getNote();` Return the data member.
- `const Note& getNote() const;` Return the data member.
- `double getStart() const;` Return the data member.
- `void setStart(const double start);` Update the data member, but only if the new values is non-negative.

Free Functions:

- `std::ostream& operator<<(std::ostream& os, const StaffNote& staff_note)` Sends the note information to the stream. Format is “start note_name note_duration”.

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

`MusicalStaff` Class

This class will represent a staff, with an instrument and a list of staff notes.

`protected` Data Members:

- `std::string` The name of the staff.
- `std::shared_ptr<Instrument>` The instrument used to render the notes to an audio track.
- `std::vector<StaffNote>` The notes of the staff.

`public` Methods:

- `MusicalStaff();` defaults to empty name, nullptr for Instrument, and no notes
- `MusicalStaff(std::shared_ptr<Instrument> instrument);` empty name and no notes, parameter used to initialize Instrument
- `MusicalStaff(const std::string& name, std::shared_ptr<Instrument> instrument);` no notes, parameters initialize name and Instrument
- `MusicalStaff(const MusicalStaff& src);` copy all data members from src.
- `virtual ~MusicalStaff();` Required, needs empty block of code.
- `MusicalStaff& operator=(const MusicalStaff& rhs) = default;` Required. Note the implementation is

supplied by the compiler.
- `virtual std::string toString() const;` Formats for printing. Format is “name_of_staff name_of_instrument”.
- `const std::string& getName() const;` Return the data member.
- `void setName(const std::string& name);` Update the data member.
- `std::shared_ptr<Instrument> getInstrument();` Return the data member.
- `std::shared_ptr<const Instrument> getInstrument() const;` Return the data member.
- `void setInstrument(std::shared_ptr<Instrument> instrument);` Update the data member.
- `void addNote(const StaffNote& note);` Add the note to the end of the notes.
- `const std::vector<StaffNote>& getNotes() const;` Return the data member.
- `double getDurationInWholeNotes() const;` Computes the end of the last note to finish, in whole notes. This is not necessarily the last note in the list. Finish time is the start time plus duration.
- `void render(const TimeSignature& time_signature, const double tempo, const int samples_per_second, AudioTrack& track) const;` Renders the staff into digital sound, in the audio track. Basic idea: create an audio track that is long enough to hold the whole staff. For every note, use Instrument::generateSamples to fill a temporary audio track with samples. Then, add the temporary audio track to the aggregator audio track, at the correct position.

Free Functions:

- `std::ostream& operator<<(std::ostream& output_stream, const MusicalStaff& staff);` Uses `MusicalStaff::toString()` to send the string representation to the output stream.

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

We will update the `ScoreReader` class by adding the ability to read a staff.

`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 STAFF keyword and call `readStaff()`. Does not yet keep the staff.
- `void readStaff(std::istream& is, MusicalScore& score, MusicalStaff& staff) const;` Reads the entire staff from the input stream. See the description above.

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

We will update the `ScoreWrite` class by adding the ability to write a staff.

`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 writeStaff(std::ostream& os, const MusicalScore& score, const MusicalStaff& staff) const;` Write a staff object to the output stream in the format specified in the `.score` file format.
- `void writeStaffNote(std::ostream& os, const MusicalScore& score, const StaffNote& staff_note) const;` Write individual note information to the output stream in the format specified in the `.score` file format.
- `std::string getDurationLetter(const double duration) const;` Given the floating point duration of a note in fractions of a whole note, return a string representing that duration. See the table of notes above.
- `std::string formatNote(const Note& note) const;` Given the note object, format the note for writing to the

output stream in the format specified in the `.score` file format.

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