CS 3005: Programming in C++

Score Editor (Writer)

Introduction

In this assignment you will start the ability to write score files from the score editor program. This will be apparent to users through a new command.

We will also add support for instruments to the score editor program. This includes user commands to add, edit, and list instruments. It also includes additions to reading and writing score files that that have instruments.

Future assignments will continue to add functionality to the program.

Syntax in the `.score` file for an instrument

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 INSTRUMENT keyword must always be followed by the instrument name.

The INSTRUMENT description must always contain a WAVEFORM and an ENVELOPE. These may be complete descriptions of new waveforms and envelopes, or they may be a identifiers for a waveform and an envelope that have previously been described. "Previously described" means it has been already added to the score. This can happen because the waveform or envelope was described in an earlier INSTRUMENT description, or it was described earlier in the file, outside of an instrument, or it was added to the `MusicalScore` object before the score file was read.

INSTRUMENT with full Waveform and Envelope descriptions

```
INSTRUMENT piano-left
  WAVEFORM piano-sawtooth sawtooth
    AMPLITUDE 0.87
  WAVEFORM-END

  ENVELOPE piano-adsr ADSR
    MAXIMUM-AMPLITUDE 0.25
    ATTACK-SECONDS 0.01
    DECAY-SECONDS 0.02
    SUSTAIN-AMPLITUDE 0.5
    RELEASE-SECONDS 0.03
  ENVELOPE-END

INSTRUMENT-END
```

INSTRUMENT with minimal Waveform and Envelope descriptions

```
INSTRUMENT piano-right
  WAVEFORM piano-sine sine
  WAVEFORM-END

  ENVELOPE piano-ad AD
  ENVELOPE-END

INSTRUMENT-END
```

Sample SCORE file

```
SCORE

    WAVEFORM piano-square square
      AMPLITUDE 0.77
```

```
    WAVEFORM-END

    WAVEFORM piano-sawtooth sawtooth
      AMPLITUDE 0.87
    WAVEFORM-END

    ENVELOPE piano-ar AR
      ATTACK-SECONDS 0.02
      SUSTAIN-AMPLITUDE 0.75
      RELEASE-SECONDS 0.03
    ENVELOPE-END

    ENVELOPE piano-adsr ADSR
      ATTACK-SECONDS 0.01
      DECAY-SECONDS 0.02
      SUSTAIN-AMPLITUDE 0.5
      RELEASE-SECONDS 0.03
    ENVELOPE-END

  INSTRUMENT piano-right
    WAVEFORM piano-square square
      AMPLITUDE 0.77
    WAVEFORM-END

    ENVELOPE piano-ar AR
      ATTACK-SECONDS 0.02
      SUSTAIN-AMPLITUDE 0.75
      RELEASE-SECONDS 0.03
    ENVELOPE-END

  INSTRUMENT-END

  INSTRUMENT piano-left
    WAVEFORM piano-sawtooth sawtooth
      AMPLITUDE 0.87
    WAVEFORM-END

    ENVELOPE piano-adsr ADSR
      ATTACK-SECONDS 0.01
      DECAY-SECONDS 0.02
      SUSTAIN-AMPLITUDE 0.5
      RELEASE-SECONDS 0.03
    ENVELOPE-END

  INSTRUMENT-END

SCORE-END
```

Notes on the `ScoreReader::readInstrument` method

- The method is called *after* the `INSTRUMENT` keyword has been read, so it will start reading the name.
- If the instrument name refers to an instrument already existing in the score, the method still needs to read until the `INSTRUMENT-END` keyword, but it should not make any changes to the existing instrument.
- If the instrument name does not exist in the score, and valid wavform and envelope descriptions are encountered, the method should use `make_shared` to create a new one, and add it to the score.
- When `ENVELOPE` or `WAVEFORM` keywords are encountered, the method should call the appropriate method in the `ScoreReader` object, receiving the return value to use in the creation of the instrument.
- If input processing ends without a `INSTRUMENT-END` keyword, the method should return a null pointer, not adding an instrument to the score.
- If input processing ends without a valid waveform or without a valid envelope, the method should return a null pointer, not adding an instrument to the score.
- Any word that is read in a position that should be a keyword, but is not a recognized keyword, should cause the instrument to not be added to the score. The method should return a null pointer immediately if this happens.
- If the ending keyword is not found before the end of the input stream, the instrument should not be added to the score.
- Under normal operation, the method returns the instrument pointer, whether it is a new pointer that has been added to the score, or it is an already existing instrument in the score.
- If an error occurs, such as reading non-keyword or end of input stream without ending keyword, the method should return a null pointer.

Notes on the `ScoreWriter::writeScore` method

- The SCORE and SCORE-END keywords are at the beginning of their lines.
- All waveforms, envelopes, and instruments should be written in their container order, as defined by the iterator.
- Writing the items should be accomplished by calling the specific method.

Notes on the `ScoreWriter::writeInstrument` method

- The INSTRUMENT and INSTRUMENT-END keywords have 2 leading spaces.
- Writing the waveform and envelope should be accomplished by calling the specific methods for each.

Notes on the `ScoreWriter::writeWaveform` method

- The WAVEFORM and WAVEFORM-END keywords have 4 leading spaces.
- The attribute keywords inside the waveform have 6 leading spaces.
- All relevant attributes should be written in the order they are configured in the UI functions.

Notes on the `ScoreWriter::writeEnvelope` method

- The ENVELOPE and ENVELOPE-END keywords have 4 leading spaces.
- The attribute keywords inside the envelope have 6 leading spaces.
- All relevant attributes should be written in the order they are configured in the UI functions.

Assignment

Here are the *new* commands that are required for this assignment. Previous commands are still required.

Command	Prefixable?	Function	Description
`score-write`	no	`writeScoreUI`	Write score to score file.
`score-list-instruments`	no	`listScoreInstrumentsUI`	List instruments in the score.
`score-add-instrument`	no	`addScoreInstrumentUI`	Add instrument to the score.
`score-edit-instrument`	no	`editScoreInstrumentUI`	Edit instrument in the 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-write - Write score to score file.

Choice? score-add-waveform
Waveform name: sin1
Waveform type: sine
Amplitude: 0.80
Choice? score-add-waveform
Waveform name: squ2
Waveform type: square
Amplitude: 0.82
Choice? score-add-envelope
Envelope name: ad1
```

```
Envelope type: AD
Maximum amplitude: 0.75
Attack seconds: 0.011
Choice? score-add-envelope
Envelope name: adsr4
Envelope type: ADSR
Maximum amplitude: 0.78
Attack seconds: 0.016
Decay seconds: 0.017
Sustain amplitude: 0.57
Release seconds: 0.018
Choice? score-add-instrument
Instrument name: ins1
Waveform name: sin1
Envelope name: ad1
Choice? score-add-instrument
Instrument name: ins2
Waveform name: squ2
Envelope name: adsr4
Choice? score-list-instruments
ins1 : sin1 ad1
ins2 : squ2 adsr4
Choice? score-edit-instrument
Instrument name: ins1
Waveform name: squ2
Envelope name: ad1
Choice? score-edit-instrument
Instrument name: ins2
Waveform name: squ2
Envelope name: ad1
Choice? score-list-instruments
ins1 : squ2 ad1
ins2 : squ2 ad1
Choice? score-write
Filename: demo.score
Choice? quit
```

The output file: demo.score.

Programming Requirements

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:

- An `Instrumentarium` collection object for storing all instruments that may be used in the music.

`public` Methods:

- `MusicalScore();` Allow the `Instrumentarium` object to be default constructed. Should not require any changes.
- `Instrumentarium& getInstrumentarium();` Return the data member.
- `const Instrumentarium& getInstrumentarium() const;` Return the data member.

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

We will update the `ScoreReader` class by adding the ability to read `Instrument` objects. Future assignments will add more to the class.

`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 INSTRUMENT keyword, and call `readInstrument()` to read the instrument.
- `std::shared_ptr<Instrument> readInstrument(std::istream& is, MusicalScore& score) const;` This method reads an instrument formatted with the format described above.

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

We will create the `ScoreWriter` class with the ability to write the currently existing score elements. Future assignments will add more to the class.

`ScoreWrite` 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:

- `ScoreWriter();` Empty, nothing to initialize.
- `virtual ~ScoreWriter();` Required but empty body.
- `void writeScore(std::ostream& output_stream, const MusicalScore& score) const;` Write the SCORE and SCORE-END keywords to the output stream, along with the waveforms, envelopes and instruments. Uses specific methods to write individual items.
- `void writeInstrument(std::ostream& output_stream, const MusicalScore& score, const Instrument& instrument) const;` Writes an instrument to the output stream. Note the file format examples above. Uses other methods to write individual items.
- `void writeWaveform(std::ostream& output_stream, const MusicalScore& score, const Waveform& waveform) const;` Writes a waveform to the output stream. Note the file format examples above.
- `void writeEnvelope(std::ostream& output_stream, const MusicalScore& score, const Envelope& envelope) const;` Writes an envelope to the output stream. Note the file format examples above.

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

Commands created for the score editor program will go here. We'll make a few in this assignment, and add more in future assignments.

Functions:

- `int register_score_editor_commands(ApplicationData& app_data);` Update to add the new commands specified for this assignment.
- `void writeScoreUI(ApplicationData& app);` Asks the user for a filename, opens it as an output file stream, and uses a temporary `ScoreWriter` object to write the score to the file. If the file fails to open, an error message is displayed. See formatting examples above.
- `void listScoreInstrumentsUI(ApplicationData& app);` Display the list of instruments in the score. See formatting examples above.
- `void addScoreInstrumentUI(ApplicationData& app);` Add an instrument to the score. See formatting examples above.
- `void editScoreInstrumentUI(ApplicationData& app);` Edit an instrument in the score. See formatting examples above.

Additional Documentation

- None

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