CS 330: Homework 3

Nullaby

Overview

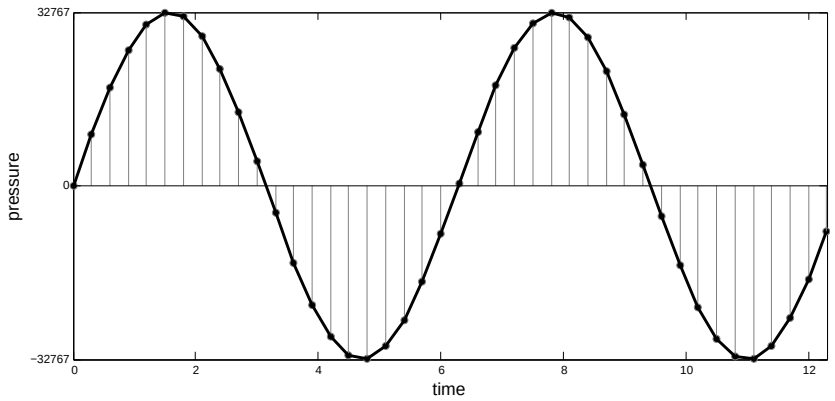
Your task in this homework is write a tool for converting an XML representation of a melody written in the standard Western musical scale to a WAV file. You will digitally synthesize each note. Because there is no way to predict how much memory will be needed ahead of time, you will dynamically allocate large chunks of memory. When all the samples of all the notes have been stitched together, the melodious stream can be written to disk.

As you complete this homework, you will gain more experience with pointers, dynamic memory allocation, binary files, protocols, higher-order functions, push parsing, callback-driven systems, and the C++ programming language.

Digital Sound

Sound is a wave of pressure that propogates through the physical world. How fast a sound wave moves determines its pitch. This rate of movement, or *frequency*, is measured in cycles or oscillations per second.

We record sound using some physical device that is sensitive to a wave's changing force. Early *analog* recorders used a vibrating stylus to etch a continuous alternate representation of the sound into foil, wax, or paper. Today, recording is done with microphones that produce an electrical signal as they vibrate. Modern recording equipment is *digital*—capturing only a countable sampling of the wave's force. Pressure readings are relative to the microphone's resting position. When the pressure is greater than equilibrium, we see positive values. When the pressure is lower, we see negative values. If we plot these readings over time t, we see a displacement graph like the following:



One of the beautiful things about this digital model is that we can synthesize sound artificially—we don't need to record something first. In fact, to create a sound clip for a given frequency and volume, we simply walk along the wave represented by $\text{amplitude} \cdot \sin(2\pi t \cdot \text{frequency})$. We capture the magnitude of the wave at all sample times t and write them out to disk using some digital sound protocol like the WAV format. Our sound card and speakers will convert these samples back into a continuous signal that is blasted toward our eardrums.

XML Music

The musical notation for this homework is a simple XML format. The following example demonstrates a possible (i.e., random and terrible) melody containing notes of several durations and some repetition of note sequences:

```
<?xml version="1.0" encoding="UTF-8"?>
<song octave="4" duration="2" bpm="120">
  <note name="C"/>
  <note name="D+" duration="1"/>
  <rest duration="2"/>
  <repeat times="3">
    <note name="B-" octave="5"/>
    <note name="A" octave="5"/>
  </repeat>
</song>
```

Believe it or not, you don't need to have any background in music to be able to complete this assignment.

The following rules describe the XML file structure in more detail. You may assume the XML file is well-formed, but you cannot assume the attributes to be in any particular order.

1. The root entity is `song`, which you may assume has a default octave attribute, a default note/rest duration attribute, and a beats-per-minute attribute. Durations are of the form 1 (a whole note), 2 (half), 4 (quarter), 8 (eighth), 16 (sixteenth), or 32 (thirty-second).

2. Nested in `song` is a sequence of `note`, `rest`, and `repeat` entities.

3. All `note`s have a `name` attribute, one of A, B, C, D, E, F, G, optionally followed by a + (sharp, raise the note a half step) or − (flat, lower the note a half step). Notes may or may not have `octave` and `duration` attributes. If they do not, the containing `song`'s attributes are used.

4. `Rest`s may or may not have a `duration` attribute. If they do not, the containing `song`'s `duration` attribute is used. Rests are effectively soundwaves with frequency 0.

5. Repeated sequences are grouped in a `repeat` entity. You may assume such entities have a `times` attribute indicating how many times the sequence is to be inserted in the song. If `times` is 2 and the contained sequence is ABC, the repeated sequence effectively expands to ABCABC. You may assume the `times` attribute is greater than 0. Repeated sequences may nest.

Requirements

Specifications do not tell you how to solve a problem—just what pieces may be used. The requirements below and the descriptions above will need to be thought about and pieced together using your own good mind. You will likely need to read this document many times, ask questions, and afford yourself plenty of time to finish this homework.

Your solution is to meet the following specification:

- Place all files in directory `<YOUR-REPOSITORY>/nullaby`.
- All code must run on `dplsubmit`.
- Write code in C++. You may use functions from either the standard C or C++ libraries.
- Keep functions and methods short, document any code chunk whose purpose isn't immediately obvious, and use meaningful names.
- Declare each class in its own header file and implement it in its own implementation file. Name the files using the class name, e.g., `Note.h` and `Note.cpp`.
- Write a class `Pitch` with the following specification:
  - Has a constructor that accepts a duration parameter as an `int`.
  - Has a `virtual` destructor. (See `http://www.parashift.com/c++-faq/virtual-dtors.html`.) Likely this method need do nothing but exist.
  - Has a pure `virtual const` method named `GetFrequency` that returns the pitch's frequency as a `float`.
  - Has a non-virtual `const` method `GenerateSamples` that accepts as parameters a beats-per-minute as an `int` and a number of samples as an `int` reference. It returns a pointer to a `short` buffer containing samples for the pitch. The number of samples is stored in the reference parameter. A clue as to how you might generate the samples is shared below. What the reference parameter holds when this method is called is undefined. It is an out parameter.
  - Has a pure `virtual const` method named `Clone` that returns a pointer to a dynamically-allocated copy of this pitch.
- Write a class `Note` with the following specification:

- Has a constructor that accepts as parameters the note's name as a `const string` reference (matching `/^[ABCDEFG][+-]?$/`), its octave as an `int`, and its duration as an `int`.
- Has a `const` method `GetHalfStepID` that returns the note's halfstep number as an `int`. Consider the halfstep IDs of all possible notes in the first octave:

C0	C+0/D-0	D0	D+0/E-0	E0	E+0/F0	F+0/G-0	G0	G+0/A-0	A0	A+0/B-0	B0/C-1
0	1	2	3	4	5	6	7	8	9	10	11

The IDs of the second octave are offsets of the first:

C1	C+1/D-1	D1	D+1/E-1	E1	E+1/F1	F+1/G-1	G1	G+1/A-1	A1	A+1/B-1	B1/C-1
12	13	14	15	16	17	18	19	20	21	22	23

And so it goes. Note the octave starts at C, not A. (Hint: you can write this function without any loops or monstrous arrays. Use the letter of the note as an index into a baseline 7-element array.)

- Overrides `GetFrequency` to return the note's frequency. Physicists tell us to calculate the frequency of note X using these equations:

$$\begin{aligned}
\text{frequency of A4} &= 440 \\
\text{magic number} &= 2^{\frac{1}{12}} \\
\text{distance of X from A4} &= \text{halfstep ID of X} - \text{halfstep ID of A4} \\
\text{frequency of X} &= \text{frequency of A4} \cdot \text{magic number}^{\text{distance of X from A4}}
\end{aligned}$$

- Overrides `Clone` to return a pointer to a dynamically-allocated copy of this note. Exploit the fact that we can have covariant return types, i.e., return the pointer as a `Note *`—not a `Pitch *`.

- Write a class `Rest` with the following specification:
  - Has a constructor that accepts as a parameter the rest's duration as an `int`.
  - Overrides `GetFrequency` to return the rest's frequency—which is 0.
  - Overrides `Clone` to return a pointer to a dynamically-allocated copy of this rest. Exploit the fact that we can have covariant return types, i.e., return the pointer as a `Rest *`—not a `Pitch *`.

- Write a class `Nullaby` that encapsulates a melody, which is a sequence of pitches. It has the following specification.
  - Has a `static const` member named `SAMPLES_PER_SECOND` with value 22050.
  - Has a default constructor.
  - Has a deconstructor that releases all resources owned by the melody.

  - Has a `const` method `GenerateSamples` that accepts as parameters a beats-per-minute as an `int` and a number of samples as an `int` reference. It returns a pointer to a `short` buffer containing samples for the melody. The number of samples is stored in the reference parameter. The samples are generated by concatenating all the pitches' samples together into a contiguous, all-encompassing `short` buffer.
  - Has a method `WriteWAV` that accepts as parameters beats-per-minute as an `int` and path to a WAV file as a `const string` reference. It writes the melody's samples out to the specified file, conforming to the WAV specification described at `https://ccrma.stanford.edu/courses/422/projects/WaveFormat`. Use just one channel, select PCM, sample at 22050 samples per second, and write the samples out as `short`s.
  - Has a `const` subscript method that accepts an `int` index parameter and returns the pitch at the specified index as a `const Pitch` reference.
  - Has an `Add` method that accepts a `const Pitch` reference as a parameter. It adds a clone of the pitch onto the end of this melody. Bear in mind that since `Nullaby` triggered the allocation of the clone, it is responsible for releasing the clone's resources when finished with it.
  - Has an `Add` method that accepts a `const Nullaby` reference as a parameter. It adds clones of all the pitches in the specified melody to the end of this melody.
  - Has a `const` method `GetPitchCount` that returns as an `int` the number of pitches that have been added to this melody.

- Has a `main` function and all non-class code in `main.cpp`.
- Expects command-line arguments for the input XML file and a path to the output WAV file.
- Has a `makefile` whose default rule builds an executable named `nullaby`. Add a `clean` rule that deletes your executable. Compile with `-g` to retain more debugging information. Compile with `-std=c++0x` to use newer C++ features.
- Parses the XML file with the `expat` library. Use no global variables. Use `XML_SetUserData` to pass data around to your callbacks.
- Frees all dynamically-allocated memory. Valgrind must report no memory leaks.
- Is accompanied by an XML song that you create. Melodies need not be original. Share it on Piazza in a note tagged `nullaby`.

Generating Samples

Consider the following pseudocode for generating samples for a note of a given frequency and duration (which is 1, 2, 4, 8, 16, or 32):

```
beats_per_second = beats_per_minute / 60
seconds_per_beat = 1 / beats_per_second
seconds_per_whole_note = seconds_per_beat * 4 // assumes 4/4 time

nseconds = seconds_per_whole_note / duration
nsamples = nseconds * samples_per_second

make samples array

// i is an index, which we must convert to time
cycles_per_sample = frequency / SAMPLES_PER_SECOND
for each sample i
  sample i = sin(2 * pi * i * cycles_per_sample) * 32767
```

This only generates samples for an individual note. When you join them with samples from another note, you will hear some popping in the audio. I recommend you gradually increase the magnitude of the first `seconds_per_whole_note / 64` seconds of each pitch from 0 it its full value. Likewise, diminish the last `seconds_per_whole_note / 64` seconds from its full value to 0.

Submission

This homework is graded by hand and with help from the grading script, which can run from your homework directly with `../specs/grade`. Your assignment is expected to fully meet the requirements above, those checked by the grading script, and the following:

1. Variable names should be meaningful and accurate.
2. Non-obvious parts of your code should be commented.
3. Code should be cleanly formatted and indented.

Your work will also be inspected for plagiarism. Please do your own work. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. (If you find violators of this rule, please let me know.) Write your own code.

The grading script sends your instructor an email when it successfully completes. All that remains is for you to push your code to Bitbucket.