CS 330 Homework

Timbre

1 Overview

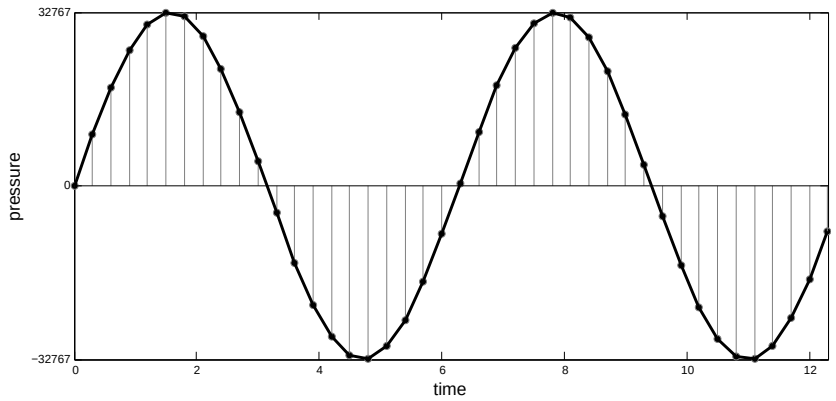
Your responsibility in this homework is to explore abstraction, operator overloading, and memory management in C++ and cross-language execution. You will do so in the context of a writing a program that generates digital sounds with various `timbre`s. Its GUI will be written in Java, but the sound generation code will be written in C++. This is a common split. User interfaces are easier to develop in Java, whereas C++ gives us fast performance.

While the C++ portion of this assignment is checked by unit tests and therefore has a firm specification, the GUI portion of this assignment is left largely unspecified. Creativity is welcomed.

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—which we call *hertz*.

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 representation of the sound into foil, wax, or paper. Today, recording is done with microphones that vibrate under the pressure waves and produce an electrical signal. 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 for a pure frequency, we see a displacement graph like the following:



One of the beautiful things about the displacement model of sound 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 generated by Equation 1:

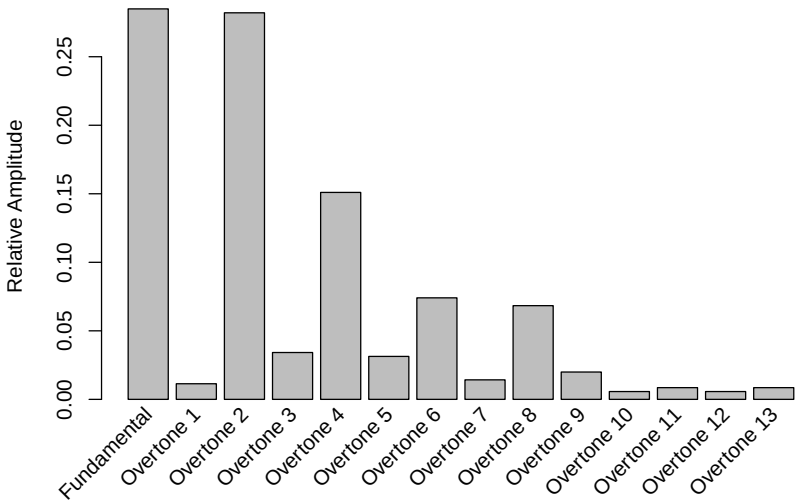
$$\text{pressure at time } t = \text{amplitude} \cdot \sin(2\pi t \cdot \text{frequency}) \tag{1}$$

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.

Overtones

When you push air through a trumpet or hammer down on a piano keyboard, you are actually playing many frequencies. The frequency of the note you think you're playing is the *fundamental* frequency. You are also playing *overtones*, which are integer multiples of the fundamental frequency. The overtones of 440 Hz are $2 \times 440 = 880$ Hz, $3 \times 440 = 1320$ Hz, $4 \times 440 = 1760$ Hz, and so on, with the i^{th} overtone having frequency $i * 440$ Hz. Sounds composed of only one frequency—such as those generated by a machine using Equation 1—sound artificial because they lack overtones. If one wants to synthesize sounds that mimic a real instrument, one must mix the fundamental frequency with its overtones.

Each instrument combines a fundamental frequency and its overtones in different ways. This unique combination of an instrument is called its *timbre*—pronounced *TAM buhr*. A clarinet's timbre, for example, consists of a high-amplitude fundamental frequency, low-amplitude first overtone, high-amplitude second overtone, low-amplitude third overtone, high-amplitude fourth overtone, and so on. The odd overtones are nearly mute, while the even overtones start with high amplitudes but diminish:



The relative amplitudes sum to 1.

2 Requirements

In order to complete this homework, you must satisfy these requirements:

1. Place all files in `<YOUR-REPOSITORY>/timbre`. Case matters.
2. Write `Music.{h,cpp}` to map notes to frequencies, as described below.
3. Write `Samples.{h,cpp}` to generate and mix sound clips, as described below.
4. Read no uninitialized memory, leak no memory, and commit no other such violations of memory, as determined by `valgrind`.
5. Write a GUI for generating sounds and exporting them to WAV files, as described below.
6. Write `makefile`. When run with `make`, it must compile your code to produce an executable. When compiling C code, include `-g` in the options to retain debugging information in the machine code. When run with `make clean`, it must remove all generated files, including class files, object, and library files. When run with `make gui`, it runs your complete sound generating program for interactive use. Do not use any absolute paths—I should be able to checkout your project on my machine without depending on files on your machine.
7. Share a screenshot of your GUI and a sample WAV file generated by your program on Piazza, under folder `timbre`.

To be eligible for later-week submission, you must successfully complete the two methods in `Music.cpp` and a `makefile`. Successful completion is determined by the grading script and the unit tests in `unit_tests1.cpp`.

2.1 Music.{h,cpp}

Complete the following requirements for mapping a note name to its frequency:

1. Write function `GetHalfStepID` to accept a note name as `const string` reference. The name consists of the note's letter (A-G), its optional sharpness of flatness (which raise or lower the frequency), and its octave. For example, an A in the 5th octave has name `A5`, a C♯ in octave 6 has name `C+6`, and an E♭ in octave 4 has name `E-4`. It returns as an `int` the note's half-step ID.

   To map a note to its half-step ID, consider the half-step IDs of all possible notes in the first octave:

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

Note that because of sharps and flats, two different names may often generate the same half-step ID.

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

C1 12	C+1/D-1 13	D1 14	D+1/E-1 15	E1/F-1 16	E+1/F1 17	F+1/G-1 18	G1 19	G+1/A-1 20	A1 21	A+1/B-1 22	B1/C-2 23

And so it goes. Note the octave starts at C, not A.

Implementation 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. Use the octave to determine how far away you are from the baseline array.

2. Write function `GetFrequency` that accepts a note's name as a `const string` reference, just like `GetHalfStepID`, and returns its frequency as a `float`. Calculate the frequency of note X using these equations:

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

3. Declare functions `GetHalfStepID` and `GetFrequency` in `Music.h`. Define them in `Music.cpp`. Place them in namespace `music`.

2.2 Samples.{h,cpp}

Complete the following requirements for generating and mixing sound clips:

1. Write a `Samples` class with the following `public` interface:

   (a) A constructor for sampling a given note. It accepts a frequency as a `float` and a duration as an `int`. It generates and associates with this instance the samples of sine wave defined by the given frequency and duration. How do we do this? First, let's make some assumptions:

   - The duration is one of 1 (whole note), 2 (half note), 4 (quarter note), 8 (eighth note), and 16 (sixteenth note).
   - The sample rate to be 22050, meaning we will generate 22050 samples for each second of sound. You will need this value several times, so defining it as a constant is probably a good idea.
   - A quarter note corresponds to a single beat in the music.
   - The music plays at 120 beats per minute.

- Amplitude is initially 1, meaning the note is at full volume.

With these (arbitrary) conventions in place, let's first figure out how many total samples we'll need for our samples:

$$\text{beats per minute} = 120 \tag{2}$$

$$\text{samples per second} = 22050 \tag{3}$$

$$\text{beats per second} = \frac{\text{beats per minute}}{60} \tag{4}$$

$$\text{seconds per beat} = \frac{1}{\text{beats per second}} \tag{5}$$

$$\text{seconds per whole note} = \text{seconds per beat} \times 4 \tag{6}$$

$$\text{seconds for this note} = \frac{\text{seconds per whole note}}{\text{duration}} \tag{7}$$

$$\text{samples count for this note} = \lceil \text{seconds for this note} \times \text{sample per second} \rceil \tag{8}$$

We now know how many samples we need. We can generate the sample values themselves by iterating through the samples and applying Equation 1. However, this equation expects a time value t. Somehow we'll need to convert a sample index into a time value. If we just had a value that told us how many seconds long each sample was, we could turn index i into time t:

$$i \ \cancel{\text{samples}} \times ? \ \frac{\text{seconds}}{\cancel{\text{sample}}} = t \text{ seconds} \tag{9}$$

We do have such a value—at least, we have the reciprocal samples per second from which we can derive this value. With all unknowns determined, we can generate the samples of the sine wave for this note.

Please use `float`s to store your samples.

(b) Write a constructor that wraps up a pre-made samples array. It accepts an `int` number of samples and a `float` pointer to a dynamically-allocated array of samples. The class considers the pointed-to memory as its own. The constructor does not make its own copy, and you will ultimately need to free the block when you find its no longer needed.

(c) A copy constructor that accepts a `const` reference to another `Samples` instance. This constructor clones this instance using the parameter as a template. The two instances do not share memory—this is a deep copy.

(d) A destructor that frees any dynamically allocated memory persisted by this instance.

(e) A `const GetLength` method that returns the number of samples in this instance.

(f) A `[]` method that accepts an `int` sample index i and returns the i^{th} sample as a `float` reference. This method is needed to outside clients can both read and write to a unprotected `Samples` instance.

(g) A `const []` method that accepts an `int` sample index i and returns the i^{th} sample as a `const float` reference. This method is needed so outside clients can read samples from a read-only `Samples` instance.

(h) An `=` method that accepts another `Samples` instance as a `const` reference and returns a reference to this instance. If this is not a self-assignment, this instance's private members are replaced by copies of the parameter's. Do not leak any memory.

(i) A `*=` method that accepts a `float` scale factor and returns a reference to this instance. It scales each sample by the scale factor. This method is used to raise or lower the amplitude/volume of the sound.

(j) An `|=` method that accepts another `Samples` instance as a `const` reference and returns a reference to this instance. It mixes the parameter's samples with this instance's samples. One mixes sounds by adding pressure values together. You may assume that the two instances have the same length, though this isn't a great assumption. An exception or `assert` statement are safer, but neither is mandated.

(k) A `+=` method that accepts another `Samples` instance as a `const` reference and returns a reference to this instance. It appends all the samples of the parameter after the samples of this instance. You will likely need to create a new array large enough to hold the combined samples arrays. Do not leak memory.

(l) A `*` method that accepts a `float` scale factor and returns a new `Samples` instance based on this instance, but with all its samples scaled. This method is similar to `*=`, but it doesn't change the invoking instance. In fact, you can use your copy constructor and `*=` to implement this method.

(m) A `WriteWAV` method that accepts a path as a `const string` reference. It writes the samples out to a WAV file at the specified path. The WAV protocol is described at `http://www.twodee.org/forothers/wav`[1].

Implementation hint: I prefer using the C API for writing binary files. Functions `fopen`, `fprintf`, `fwrite`, and `fclose` made the task straightforward. I also converted the `float`s to `short`s before writing out the file. This involved scaling by 32767 and casting each sample as I dropped it into a new `short`s array.

2. Declare the `Samples` class in `Samples.h`. Define it in `Samples.cpp`. Place it in the default namespace.

I suggest you test this code before jumping in to the GUI. However, please put your `main` function in a separate file. `Samples.cpp` and `Music.cpp` must not contain their own `main` function, otherwise the unit tests in the grader (which define their own `main`) will fail to compile.

[1]Thanks Craig Stuart Sapp, Scott Wilson, and Stanford University for documenting the protocol and the Internet Archive for preserving their work.

2.3 GUI

Your GUI must meet the following specification:

1. It is written in Java but calls down to your C++ code via the Java Native Interface (JNI) to generate samples and WAV files.
2. It allows the user to configure a note (a frequency and duration) and the relative amplitudes of its fundamental frequency and overtones. What widgets you use to accomplish this are up to you.
3. It allows the user to assemble a series of notes together into a song.
4. It allows the user to export a WAV file for the assembled song to a destination of the user's choosing (see `JFileChooser`).

3 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. 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. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you are resubmitting.