CSE 232 Spring 2016

Programming Project #7

Assignment Overview:

Translate a simple music syntax into sampled sound wav files. Experience with files and vectors, as well as using external (provided) files.

The assignment is worth 40 points (4% of the course grades), and must be completed before 11:59 pm on Monday, March 21st

Background:

Digital sound is based on the sampling of an audio signal at a high rate in such a way that the sampled signal closely mimics the actual analog signal. We are going to write a program that turns a notation of musical notes into sampled sound.

Frequency

One of the oldest problems of music is how to map the notes of a musical piece to a set of audio frequencies. There are various ``tuning approaches" that state slightly different ways of assigning notes to a particular frequency. This project will require that you write a program that does one particular kind of this mapping.

First, we must define some form of musical note notation. The one common representation is the *octpch* note notation. This notation represents each note as a number pair, where the first number indicates which *octave* the note belongs to and the second number the note within the octave, termed the *pitch class*. There are 12 *semitones* within each octave on the keyboard, namely:

Note	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
Pch	0	1	2	3	4	5	6	7	8	9	10	11

For those more musically astute, **we are ignoring** the potential equivalencies of the flat notes. For example, D♭ is equivalent to C♯. Octpch representations are often written in decimal format. For example 5.9 is the 5th octave, 9th semitone. We must map sound frequencies to this representation. We start by choosing a reference note. The reference note frequency mapping is that 4.9 (A in the fourth octave) is 440 Hz. Every tuning mapping must insure that each note in the next higher octave has a frequency that is twice the previous octave, therefore 5.9 ⇒ 880 Hz and 3.9 ⇒ 220 Hz. Our tuning system will assume that each of the semitones within an octave are equally spaced, that is the distance from one semitone to the next is the same within the octave. This is called a *Tempered Scale*. The formula we use is:

$$x * 2^{(o+\frac{m}{12})}$$

where x is a reference Hz value for a known note, o is the *difference* (sign matters) in octave between the reference note and the note in question and m is the *difference* (sign matters) between the reference semitone and the note semitone in question. Consider the value for 0.0, C in the zeroth octave, which is 4 octaves and 9 semitones below 4.9 (A of the 4th octave).

$$C_0 = 440 * 2^{(-4+\frac{-9}{12})} = 440 * 2^{-4.75} = 16.35159 Hz$$

Duration

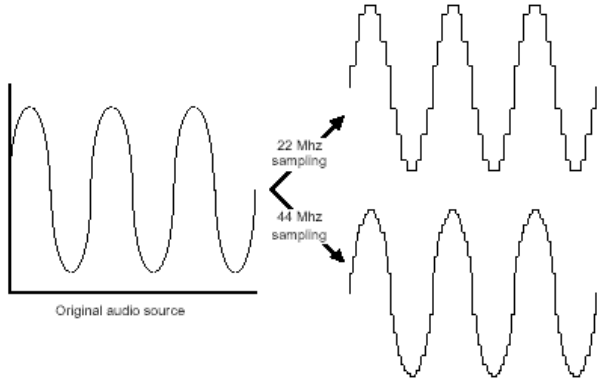
We augment the octpch representation with a count of beats. A ***beat*** is a way to measure the relative duration of a note. For example, a quarter note lasts 1 beat, a half note 2 beats and a whole note 4 beats. The translation of beat to actual time is something we have to account for (later in the specification) but a whole note in our oct-pch-dur representation would be:

4.9.4

which means, 4th octave, 9th pitch class, 4 beats

Sampled Sound

Digital sound is done by sampling an audio wave form with discrete volume measurements many thousands of times a second. CD quality samples audio at 41,000 times per second. The re-creation of that sampled sound, creating a wave form with the same shape as small lines drawn between the various sample points, creates a sound which, if sampled enough, is nearly indistinguishable from the original audio signal.



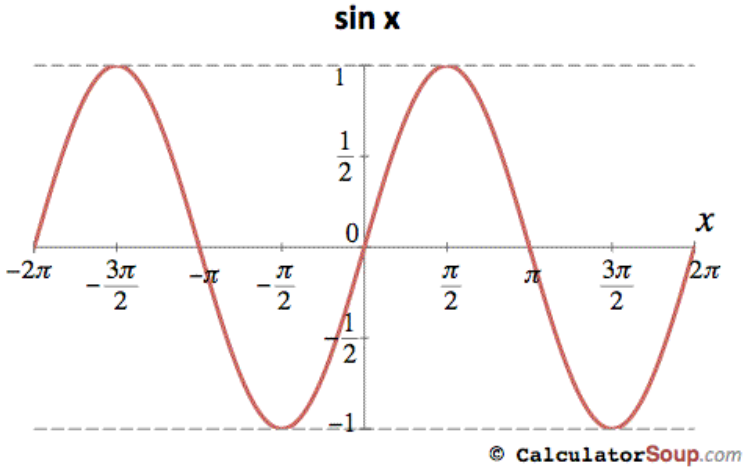
Given a frequency and a duration, we can sample a waveform and save those samples to a vector (in this case a `vector<short>`), and then play that sampled sound (with a little work). We are going to ***generate sine waves*** of particular frequencies for this project.

For example, imagine we are sampling our 440Hz Middle A tone with a sample rate of 8000 samples/second as a sine wave. If we want to generate one second of sound, we need to generate 8000 samples in one second of a sine wave at 440Hz to make a sound, really a tone.

```
double one_increment;
for (int val=0; val<8000; val++){
   one_increment = 440 * (val / 8000.0);
   …
```

The variable `one_increment` is now going to range from 0 – 440, where each increment is going to increase by 1/8000 (0.000125) each time through. We want to turn that into a sine wave, a decent waveform for a tone. Note `sample` is a double and the 8000.0 makes all operations doubles.

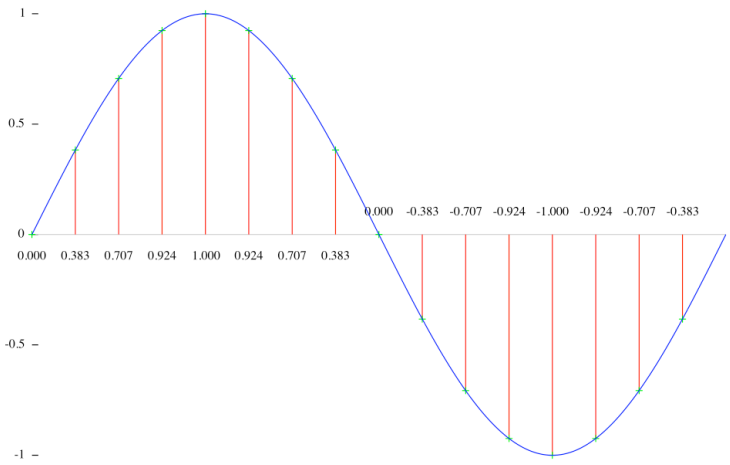
We remember from our trig classes that a sine wave has the following form:



The sin of 0 is 0, the sin of π/2 is 1, sin of π is 0, sin of 3π/2 is -1 and back to 0 at 2π. That is one full cycle of a sine wave. We need to do that 440 times in one second (sampled 8000 times at our sample rate). More specifically, our `for` loop, which goes through 8000 times, should generate 440 full cycles of a sine wave. How to do that? Look at the below:

```
double one_increment, sample;
const double TwoPI = 2.0 * (atan(1.0) * 4.0);
for (int val=0; val<8000; val++){
   one_increment = 440 * (val / 8000.0);
   sample = sin( one_increment * TwoPI );
   …
```

If we think about this, `one_increment` passes through (or near) 1.0, then 2.0, then 3.0 etc. all the way to 440. At 1.0 * 2π, we did one sine wave cycle. At 2.0 * 2π we did another for a total of two, at 3.0 * 2π we did another for a total of 3 etc. That means in the 8000 samples we generate, we go through 440 cycles of a sine wave, which is exactly what we want! We sample each individual sine wave 8000/440 times, or about 18 times. I cheated, the one below is 16 times, but you get the idea. Draw a line between each sample and you get pretty close to the actual sine wave. That is what sampled sound is all about



The last thing we want to do is set the volume associated with our sound. This is really just changing the amplitude of the generated sine waves. The following then generates our samples at a set frequency, set volume and for a set duration. Modify to solve your problem.

```
double one_increment, sample;
const double TwoPI = 2.0 * (atan(1.0) * 4.0);
for (int val=0; val<8000; val++){
   one_increment = 440 * (val / 8000.0);
   sample = sin( one_increment * TwoPI );
   sample = sample * volume;
   …
```

WAV file format

A WAV file is a format that is used to represent a sampled sound that can be played back by nearly any modern computer. You will be provided with a function that can create such a file by providing the samples created from your oct-pch-dur which you can then play. A .wav file is pretty generic. Right click on your newly generated .wav file and see what options you have to play your sound.

Problem Statement

You are going to create `Melody` and `Note` structs and some supporting functions which can read in a music notation file in oct-pch-dur and create a WAV file of sampled sound that "plays" your music.

Program Specifications:

You will be provided with a `main-07.cpp` file and a `class-07.h` file. You must provide the `class-07.cpp` file. You are also provided with `wavefile.cpp` and `wavefile.h` which contains the `write_wav_data` function. It is this function that will create a .wav file from a `vector<short>` of samples.

notes_file specification

Your program is going to read in a notes_file, a sequence of oct-pch-dur notes, with the following format:

- first line is the samples/second rate
- second line is the max volume

- every subsequent line is an oct-pch-dur triple as described above.

Note struct

The `Note` struct in the class-07.h file is fully configured. Each `Note` has a frequency and a duration, and there is a constructor which will make a `Note` given those two values. Nothing else is required here.

Melody struct

- `Melody(string notes_file_name, string wav_file_name, long a_beat)`

constructor. Takes two strings with are the names of the `notes_file` and the `wav_file`, opens them and places them in the Melody data members `ifs_` and `ofs_` respectively. The third parameter is the number of samples in a quarter note (typically ¼ of the sample rate, but you can experiment). It is stored in `one_beat_` of `Melody`

- `void fill_Note_vector()` method of `Melody`.

Parses the info in the notes_file . Using the file format described above:

- stores sample_rate in `samples_per_sec_` of `Melody`
- stores the volume in `volume_` of `Melody`
- for every line after, parses the oct-pch-dur
  - turn oct-pch into a frequency as described and store in `freq_` of `Note`
  - stores number of beats in `dur_` of `Note`
  - constructs a `Note` based on `freq_` and `dur_` and pushes it onto the `vector<Note> v_`

At the end, you have created a `vector<Note>` of all the `Notes` in the notes_file.

- `void generate_wave_data()` method of Melody

For each `Note` in `v_`:

- generate the samples using the description above, pushing each sample on the `vector<short> w_`. That is:
  - get `dur_` and `freq_` from the current `Note`
  - generate in a loop the appropriate number of samples based on beats in `dur_` and the `one_beat_` value which translates beats to samples.
  - use `freq_` to generate the appropriate sample of a sine wave as described above.

At the end you should have a `vector<short> w_` of all the samples of the `Melody`.

The main program then extracts info from the `Melody` and creates a .wav file using the provided `write_wav_data` , creating the requested wav file.

Assignment Deliverables:

`class-07.cpp` is what is required. None of the other provided files should be modified.

Please be sure to use the specified file name, and to preserve the file in your account until your assignment has been graded. You will electronically submit a copy of the file using the "handin" program.

Assignment notes:

1. We have not opened a binary file yet. A binary file is a file where the numbers are interpreted as something other than a character. A wav file is required to be binary. To open a binary file for output, one does:
   ```
   ofstream ofs_;
   ofs_.open("somefile.txt", std::ios::binary);
   ```

2. You could easily generate different waveforms by simply changing `sample = sin` line above. Common waveforms are
   a. square waves
   b. triangle waves
   c. sawtooth waves

3. You probably need a split function to break the oct-pch-dur into component parts. You know how to do that. Just place it in class-07.cpp

4. You will need some math functions from `cmath`. Take what you need (look up what is there).

5. It would be helpful if you could visualize the sound you generate. I'll see if I can get `audacity` (a sound editor) loaded on the lab and x2go machines.