Simple Audio Manipulation Program (SAMP)

CSC3022H Tutoral 5: Templating, specialization and the STL

Department of Computer Science

University of Cape Town

Deadline: 14 May 2018 10:00 AM

Note that students registered for CSC3023F do not have to complete this tutorial.

Brief:

You are required to write a C++ application which can process audio sounds clips. Using your application, it should be possible to perform simple editing operations on sound clips – such as cut and paste – as well as transforming the sound clips. Examples of the latter include fade in/out and normalisation. The sound clips will be 1-channel (mono) or 2-channel (stereo) and will be provided as simple raw byte data which you need to interpret correctly. Programmatically, a raw sound file/clip is a sequence of samples (usually, 8, 16 or 24-bits) of an audio signal that can be sent to a speaker to produce sound. The sound clip also has an associated sample rate – for example, 44.1 kHz (ie. 44100 samples per second). The higher the sample rate, the better, usually, the quality of the sound produced. The number of bits per sample also has a profound effect on audio quality: generally, 8-bits per sample produces really poor sound. Of course, high sampling rates result in very large sound files, which is why compression (such as MP3) is usually used – we will not expect you to manipulate compressed formats. Simple raw (byte stream) audio will be used throughout.

Playing sound files:

You can play sound files on Ubuntu as follow (assuming sox package is installed –you'll need headphones or speakers of course):

```
play -r 44100 -e signed -b 16 -c 2 Run_44100_16bit_stereo.raw
```

Here **-c** specifies the number of channels (1 for mono, 2 for stereo). T he sampling rate (**-r**) is 44.1 kHz, and the sample data is signed (**-e**) 16-bit (**-b**) data. Finally, the raw file has been labelled to make it clear what data it contains.

A worked example loading and manipulating these PCM RAW audio files is given as an ipython notebook. Several audio files will be made available to you (you can use Audacity to convert your favorite songs into RAW files if necessary).

Requirements:

Arguments and program invocation:

samp -r sampleRateInHz -b bitCount -c noChannels [-o outFileName] [<ops>] soundFile1 [soundFile2]

Description

- -r Specifies the number of samples per second of the audio file(s) (usually 44100)
- -b Specifies the size (in bits) of each sample. Only 8bit and 16bit should be supported in your program. More on this later on.
- -c Number of channels in the audio file(s). Your program will only support 1 (mono) or 2 (stereo).
- “outFileName” is the name of the newly created sound clip (should default to “out”).
- <ops> is ONE of the following:
    - -add: add soundFile1 and soundFile2
    - -cut r1 r2: remove samples over range [r1,r2] (inclusive) (assumes one sound file)
    - -radd r1 r2 s1 s2 : add soundFile1 and soundFile2 over sub-ranges indicated (in seconds). The ranges must be equal in length.
    - -cat: concatenate soundFile1 and soundFile2
    - -v r1 r2: volume factor for left/right audio (def=1.0/1.0) (assumes one sound file)
    - -rev: reverse sound file (assumes one sound file only)
    - -rms: Prints out the RMS of the sound file (assumes one sound file only). More details will be given later on.
    - -norm r1 r2: normalize file for left/right audio (assumes one sound file only and that r1 and r2 are floating point RMS values)
    - [extra credit] -fadein n: n is the number of seconds (floating point number) to slowly increase the volume (from 0) at the start of soundFile1 (assumes one sound file).
    - [extra credit] -fadeout n: n is the number of seconds (floating point number) to slowly decrease the volume (from 1.0 to 0) at the end of soundFile1 (assumes one sound file).
- “soundFile1” is the name of the input .raw file. A second sound file is required for some operations as indicted above.

The sample rate, bit count and number of channels should be used for both the input files and the resulting output file.

Input:

The format of the input .raw audio files is simply a stream of samples (a binary file). If you know the size of each element (8/16 bit and number of channels), and the size of the file (using seekg and tellg as done here: http://www.cplusplus.com/reference/istream/istream/tellg/) you can tell how many samples is contained in the file.

We will only use 8-bit **int** (signed int) and 16-bit **int** (signed int) sound clips, which can be represented as the types int8_t and int16_t (include <cstdint>). Clips will be 1-channel (mono) or 2-channel (stereo). Stereo files contain pairs of integers per sample where the first intN_t sample correspond to the left ear (L) and the second intN_t sample correspond to the right ear (R). You can package your LR data into an std::pair<intN_t,intN_t>, where N is the number of bits.

You can allocate a buffer that is large enough to store the entire audio clip using resize method of std::vector. Your vector should contain either intN_t samples or std::pair<intN_t,intN_t> samples depending on whether mono or stereo samples are being read. The address of the start of the buffer is given by **&(data_vector[0])**. You may find the following formulae useful when reading in audio files:

NumberOfSamples = fileSizeInBytes / (sizeof(intN_t) * channels)

Length of the audio clip in seconds = NumberOfSamples / (float) samplingRate.

Output:

When you modify or create sound files, you must save the ouput as a raw (byte) audio file (.raw extension). To help interpret the files, you should write information into the file name:

Filename_samplingrate_samplesize_monoORstereo.raw

For example, a mono output file saved with the name “spooky”, with 16-bit samples, a sampling rate of 8000 Hz would have the final name: “spooky_8000_16_mono.raw”

Templating:

The Audio class should be templated to handle audio signals which use different bit sizes for samples, depending on the provided audio clips. To handle stereo, you need to specialize your core Audio template to manipulate the data which consists of 1 **pair** of samples per time step, L and R, with L being the left ear data and R the right ear data. Thus, rather than having an array of int’s for your sound clip, you will have an array of pairs of int’s. Each sequence (all L’s or all R’s) can be handled differently.

Functionality required:

You will be required to overload some operators to achieve basic editing. These operations will all produce **new** sound clips.

A | B: concatenate audio file A and B (A and B will have the same sampling, sample size and mono/stereo settings)

A * F: volume factor A with F; F will be a std::pair<float,float> with each float value in range [0.0,1.0] The pair<> allows us to package a separate volume scale for left and right channels. To apply the operation, simply multiply each

sound sample by the volume factor. For mono channels only the first number will be used. This allows you to make one channel louder/softer in relation to the other.

A+B: add sound file amplitudes together (per sample). A and B will have the same sampling, sample size and mono/stereo settings. Each resulting amplitude must be clamped to the maximum value of the sample type. These maximums are available in <cstdint>. Adding two very loud files together may result in saturation.

A^F: F will be a std::pair<int,int> which specifies start and end sample of range of samples to be cut from sound file A. This implements a “cut” operation which produces a shorter clip (A with a portion removed).

Regular overloads and construction:

You must also overload the assignment and move assignment operators and provide the usual constructors (including copy and move) and appropriate destructor.

You should demonstrate that these operators work through simple unit tests using catch.hpp. It may be helpful to create a initializer list constructor to your Audio class to read in a small custom buffer in order to test these operators. Your unit tests should be compiled as separate executables.

Audio transformation:

You must use STL algorithms with custom Functors or Lambdas, as specified below. When ranges are required, you should use an iterator. This will be a simple pointer into your internal audio data buffer.

- Reverse: reverse all samples (this can be done very quickly with the STL)
- Ranged add: select two (same length) sample ranges from two signals and add them together. This differs from the overloaded + which adds entire audio clips together. You should make use of std::copy and your previously defined operator+ to achieve this.
- Compute RMS: use std::accumulate in <numeric> along with a custom lambda to compute the RMS (per channel), according to the following formula:

$$RMS=\sqrt{\left(\frac{1}{M}\sum_{i=0}^{M-1} x_i^2\right)}$$

  This can be seen as an “average” volume of the sound clip.
- Sound normalization: Use std::transform with a custom functor to normalize the sound files to the specified desired rms value (per channel). You will first need to compute the current RMS of the audio clip before

performing the normalization step. You may have to partially specialize the functor to work with both mono and stereo sound files.

Normalization can be done according to the following formula:

$$outputAmp = inputAmp \times \frac{RMS_{desired}}{RMS_{current}}$$

This effectively increases the overall volume of a sound clip to the desired level and can be used to normalize between audio clips. You must clamp the output amplitudes to the minimum and maximum values specified in <cstdint>.

You should demonstrate that these operators work through simple unit tests.

Fading in and out:

Fade-in/Fade-out: use a custom lambda with a simple linear function (ramp) applied to a single audio clip, over a specified range of samples. You can implement this using std::for_each().

Fade-in:

OutputAmp = (FadeSampleNo / (float) rampLength) * inputAmp

Fade-out:

OutputAmp = (1.0 - FadeSampleNo / (float) rampLength) * inputAmp

Where rampLength is the number of samples to apply (rampLength = numSeconds * sampleRate).

Grading:

Implementing everything except Fading in and Fading out will earn you 95%. Implementing the fading in/out operations (with appropriate unit tests) will earn you the last 5% of the mark.

PLEASE NOTE:

- A working *Makefile* must be submitted. If the tutor cannot compile your program on senior lab machines by typing **make**, you will receive 50% of your final mark.
- Your submission must contain a git repo. You must use version control from the get-go. Failure to comply will result in a 10% penalty.
- Do not submit binary files - submit your source code and any other necessary files to build and test your project.
- You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should not

explain any theory that you have used. These will be used by the tutors if they encounter any problems.

- Please ensure that your tarball works and is not corrupt (you can check this by trying to extract the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - no exceptions!
- A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 5 days. Do not hand in any binary files.
- DO NOT COPY. All code submitted must be your own. Copying is punishable by 0 and can cause a blotch on your academic record. Scripts will be used to check that code submitted is unique.