|  |
| --- |
| **Objective** |
| The objective of this project is to gain experience with developing C++ classes and working algorithms by developing software that can manipulate audio signals. In addition, this project requires you to use API documentation on two supplied classes to complete it. |

# Starter Code:

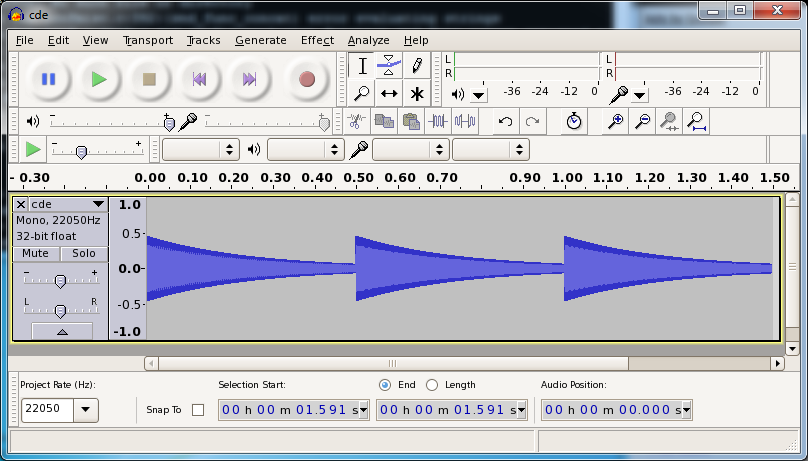
In order to streamline this project the following file(s) are supplied.

* 1. RIFF.h and RIFF.cpp: A helper class to aid reading and writing mono wave files. Refer to the detailed online API documentation (also available in RIFF.h) on how use the RIFFReader and RIFFWriter classes.
  2. NoteMaker.h and NoteMaker.cpp: A helper class to aid converting different musical note string (such as: C#, A) into corresponding audio signals. Refer to the detailed online API documentation (also available in NoteMaker.h) on how to use the class.
  3. cde.wav: This is a simple wave file that can be used for testing. You may download additional wave files from the internet for your own testing (see: <http://www.pacdv.com/sounds/> for a few more sound files for testing).
  4. There are supplied with the following text files that are supplied to you for testing your top-level MusicComposer code (you are expected to write your own tester for AudioSignal as needed):
* jingle\_bells.txt (graduate & undergraduate)
* twinkle.txt (graduate & undergraduate)
* ode\_to\_joy.txt (graduate & undergraduate)
* mary\_lamb.txt (graduate only)
* au\_claire\_de\_la\_lune.txt (graduate only)

## Verification

Verifying: For verification and troubleshooting of your code, you will need to view the wave files and listen to them. You are expected to know how to copy generate wave files to your computer and listen to them on your local PC. For visualization of wave forms you will need to use audacity (<http://audacity.sourceforge.net/>) as shown below:

|  |
| --- |
| **$** module load audacity  **$** audacity cde.wav & |



|  |  |
| --- | --- |
|  | You may compile multiple files together to create one executable using the following command-line:  **$** g++ -std=c++11 -g -Wall RIFF.cpp NoteMaker.cpp AudioSignal.cpp MusicComposer.cpp -o MusicComposer |

This project involves developing a comprehensive AudioSinal class that provides a convenient interface to manipulate audio signals. The AudioSignal class encapsulates information about a digital audio signal that has been sampled using Linear Pulse Code Modulation (LPCM). An audio signal is essentially a series of Sample (alias for 16-bit integers defined in RIFF.h) entries. Example: {0, 10, 15, 10, 0, -10, -15, -10, 0} where each value in the list is of data type Sample. In addition to the actual data, each audio signal also contains the following metadata associated with the signal:

* sampleRate: This value determines the number of samples that constitutes one (actual) second of audio signal. For example, if the sample rate is 1000 then one thousand consecutive numbers provide the necessary data to construct 1 second of audio. Typically the sample rates are at least 8000 for decent quality spoken audio and can be as high as 44100 for CD quality music.
* bitsPerSample: This value determines the minimum and maximum values for each sample in the actual aduio signal/wave. Currently, only 8-bit and 16-bit values are used. 8-bits limit the range of each sample to -128 to +127. On the other hand, 16-bit values provide higher fidelity reproduction by proving a range from -65536 to +65535. Typically 16-bit values are used to provide a good dynamic range and fidelity of music. Read more information about this topic of quality via Wikipedia article: <http://en.wikipedia.org/wiki/Audio_bit_depth>.

Remember that AudioSignal.h contains just the class definition and AudioSignal.cpp should contain the implementation for the various methods.

## API Requirement for AudioSignal class:

|  |  |
| --- | --- |
| **Private Instance variable(s) in AudioSignal:** | |
| **int** sampleRate; | Number of samples per second |
| **short** bitsPerSample; | Bits per sample. Can be 8-bits or 16-bits |
| std::vector<Sample> wave; | The actual data for this audio signal |
| **Public methods in Chart:** | |
| AudioSignal(**const** **int** sampleRate = 22050, **const** **short** bitsPerSample = 16); | Constructor to create a audio signal with a given sample rate and bits per sample. This constructor creates an audio signal that does not have any wave (or actual sound data) associated with it. It merely initializes the appropriate instance variables with suitable values. |
| AudioSignal(RIFFReader& inputWaveFile); | Constructor to create a audio signal from a given RIFF file. This constructor creates an audio signal by loading all the necessary information for various instance variables from a given ".wav"/RIFF file (inputWaveFile).  Note: Do not use istream\_iterator to read data. Instead, use standard while loop as shown in RIFFReader API documentation. [**4/3 points**] |
| Copy constructor | A standard copy constructor. [**2/2 points**] |
| Move constructor | A standard move constructor. [**2/2 points**] |
| Assignment operator | A standard assignment operator [**1/1 points**] |
| AudioSignal(**const** **int** freq, **const** **int** lenInMills, **const** **int** amplitude,  **const** **int** sampleRate = 22050, **const** **short** bitsPerSample = 16,  **const** bool decay = **true**); | This constructor creates a sinusoidal audio signal of the given frequency, maximum amplitude, sample rate, and bits per sample. The freq parameter defines the frequency of the audio signal to be generated by this method while the lenInMills parameter defines the duration of the audio signal in milliseconds. For example, if the frequency is 10 Hz and the lenInMills is 1000 (one second long) then this method generates 10 full sine waves in the given samples. The amplitude parameter defines the maximum amplitude of the signal. The sampleRate parameter indicates the sampling rate for the audio signal. The sampling rate indicates the number of samples for each second of this audio signal. The bitsPerSample indicates the number of bits occupied by each entry in the audio wave associated with this signal. This value must be either 8 or 16. If the decay parameter is true then the amplitude of the audio signal is exponentially decayed to simulate striking of real musical string instrument per the following formula :  exp((double) -i / SampleCount / 0.5)  where i is the zero-based index of the sample.  Note: For this constructor it is best to write routine loops rather than using algorithms. |
| AudioSignal **operator+**(**const** AudioSignal& other) **const**; | This method is used to superposition this audio signal with other audio signal to create and return a new audio signal. For example if this audio signal has wave data {1, 2, 3} and the other audio signal has wave data {4, -2, 6, 100, -100} the resulting audio stream has the data {7, 0, 9, 100, -100} obtained by simply adding corresponding values in two signal). This method assumes that the two signals have the same sample rate but the signals can be of different lengths. The resulting audio signal returned by this method is the same duration of the longer of the two signals.  This method returns a new audio signal (neither this nor other is modified) that is a constructive superposition of the two given audio signals. |
| AudioSignal& **operator&**(**const** AudioSignal& other); | Method to concatenate another audio signal onto this signal. This method provides implementation for the ampersand operator to append another signal to the end of this signal. For example if this audio signal has wave data {1, 2, 3} and the  other audio signal has wave data {100, 200, 300}, then this method converts this to {1, 2, 3, 100, 200, 300}.  This method returns a reference to this signal |
| AudioSignal& **operator\***(**const double** factor); | Amplify/reduce (or scale) each sample in this signal by the given factor. This method is used to suitably scale each sample in the given audio scale. For example if this audio signal has wave data  {2, -4, 6, …} and the scale factor is 2.0 then samples in this wave should be modified to {4, -8, 12, ....}. The method modifies this object and does not create a new object.    If the factor is greater than 1 then the signal is amplified (volume increases). If the factor is less than 1 then the signal is reduced (volume decreases).  This method returns a reference to this object. |
| AudioSignal **operator>>**(**long** timeInMillis) **const**; | Shift the audio signal to the right by the given amount of time (in milliseconds).  This method shifts the audio signal by the necessary number of samples (given the sampling rate and the timeInMillis) by inserting zeros at the beginning of the wave. For example if this audio signal has wave data {5, -17, 25, 3, ....} then shifting it to the right by 1 millisecond (assuming it results in requiring to add 5 samples) results in a new audio signal with the wave data {0, 0, 0, 0, 0, 5, -17, 25, 3, ...}. Note that this operator does not modify this object. The number of zeros to be inserted must be computed based on the samplingRate of the audio signal. |
| AudioSignal **operator<<**(**long** timeInMillis) **const**; | Shift the audio signal to the left by the given amount of time (in milliseconds).  This method shifts the audio signal by the necessary number of samples (given the sampling rate and the timeInMillis) by removing samples at the beginning of the wave. For example if this audio signal has wave data {5, -17, 25, 3, -9, 0, 15, ....} then shifting it to the left by 1 millisecond (assuming it results in requiring to remove 5 samples) results in a new audio signal with the wave data {0, 15, ...}. Note that this operator does not modify this object. The number of samples to be removed must be computed based on the samplingRate of the audio signal |
| long duration() **const;** | Returns the duration of this audio signal in milliseconds (determined using number of entries in wave and the sampleRate) |
| **void** write(**const** std::string& fileName) **const** throw (std::exception); | Write the audio data in this signal to a given fileName using a RIFFWriter object. This method must create a suitable RIFFWriter stream and write the audio data in this signal to the file.  Note: Do not use a ostream\_iterator. Use a for-each loop to write the data to a RIFFWriter. The method is not expected to throw exception but just exposes the exceptions that may be generated by the various methods called by the method. [**4/3 Points**] |
| **void** changeDurationTo(**const** **int** timeInMillis); | Increase or decrease the duration of this audio signal to the given duration of time.  This method changes this audio signal such that the duration of the signal is changed to the specified timeInMills value, which indicates the desired duration of the signal in milliseconds.  This method must modify the duration of the signal in the one of the following two ways:     1. If the current signal is shorter than the specified duration, then the current signal is replicated (or repeated) until the desired duration of samples is obtained. 2. If the current signal is longer than the specified duration then the signal is truncated (by simply discarding samples at the end of the signal) to the necessary duration. |

## Requirements for MusicComposer class:

Once you have developed the AudoSignal class and you have performed necessary tests to ensure all the methods are operating correctly, the next step is to develop the MusicComposer class that provides the user with a convenient mechanism to compose multi-track (undergraduate students work with 1 track) music using the supplied NoteMaker class. Refer to the API documentation on NoteMaker for details on its methods. The MusicComposer essentially uses various methods in AudioSignal and NoteMaker to generate a final output audio wave file.

The MusicComposer as the following two instance variables associated with it:

1. An overall AudioSignal (called music in this document) that indicates the music generated so far by this class.
2. A NoteMaker object to convert note strings (in the form: C#, C4, C#1 etc.) to AudioSignal objects.
3. A vector of AudioSignal objects that represents various tracks currently being operated upon by the user. By default the first track is set as the track to which audio signals are appended by the MusicComposer::operator() shown below:

The MusicComposer is an open ended class and you are expected suitably design it such that it meets the following requirements and operates correctly with the following main function that you may copy-paste into MusicComposer. In order to work correctly with the supplied main MusicComposer must have the following methods:

1. A default constructor.
2. **const** AudioSignal& **getMusic**(): This method must add all of the current tracks into a single AudioSignal and append it to the overall audio signal. All the current tracks are cleared / deleted. A new track-1 is started as the only track. This method return a reference to the overall audio signal.
3. AudioSignal& MusicComposer::**operator**()(**const** std::string& ent): This method must suitably process ent as described below, append the resulting audio signal (if any) to the current audio track and returns the current audio track back. Note that an audio track is simply an AudioSignal object:
   1. If the given ent string does not have a ‘\*’ as the first character then simply pass the string to the NoteMaker’s functor, obtain the AudioSignal, and append it to the current track.
   2. Graduate Students only: If the given ent string has ‘\*’ as the first character then, process the string as follows:
      1. \*t# (where # is an integer in the range 1 to 8): Changes the current track to the value specified. For example \*t2 changes the current track to the second track of music (note that track numbers are 1-based and not zero-based).
      2. \*l*fileName*: Loads the specified wave fileName replacing the audio signal in the current track.
      3. \*e# (where # in an integer in the range 0 to 10000): Adds an echo to the current track by: copying it, shifting copy to left by the specified value, decreasing volume of copy to ¼ of the track, and adding it back to the current track.
      4. \*~# (where # is a number indicating a valid track): Makes the duration of the specified track to be the same as the shortest of the tracks being current operated on.
      5. \*=# (where # is a number indicating a valid track): Makes the duration of the specified track to be the same as the longest of the tracks being current operated on.
      6. \*. : Calls getMusic() method to add all of the current tracks into a single AudioSignal and append it to the overall audio signal. All the current tracks should be cleared / deleted. A new track-1 must be started as the only track.
4. The MusicComposer must have the following main method in it (exactly as shown):

|  |
| --- |
| **int** **main**(**int** argc, **char** \*argv[]) {  MusicComposer mc;  std::istream\_iterator<std::string> noteStream(std::cin), eof;  mc = std::for\_each(noteStream, eof, mc);  mc.getMusic().write(argv[1]);  **return** 0;  } |

# Testing

The best way to test your program is to generate an audio file, copy it to your machine, and listen to it to ensure it sounds correct. To help comparison of operation of your program you are also supplied with a MusicComposer binary that you can use to provide inputs. In addition used audacity to view the signal to verify it is generated exactly the same as those generated by MusicComposer. You may run your program or the supplied MusicComposer using the following command-line:

|  |
| --- |
| $ ./MusicComposer jingle\_bells.wav < jingle\_bells.txt  $ audacity jingle\_bells.txt |