Assignment #4

CS 245, Spring 2018

Due Thursday, February 8

Declare and implement a C++ class for linear resampling of audio data. This class is intended to be *light* in the sense that it uses a pointer to existing data (the data to be resampled), but does *not* make a copy of the data: it assumes that the data pointer remains valid until the class is destroyed. The interface (public and private) of the class is:

```
class Resample {
public:
  explicit Resample(const AudioData *ad_ptr=0, unsigned channel=0,
                    float factor=1, unsigned loop_bgn=0, unsigned loop_end=0);
  float operator()(void);
  void offsetPitch(float cents);
private:
  const AudioData *audio_data;
  unsigned ichannel;
  double findex, findex_increment;
  float findex_multiplier;
  unsigned iloop_bgn, iloop_end;
};
```

(the header file `AudioData.h` has been included). The members of the public interface are defined below. You may not alter the interface, both public and private, in any way. In particular, you may not add additional functionality or variables.

`Resample(ad_ptr,channel,factor,loop_bgn,loop_end)` — (constructor) creates a resampling object from given audio sample data. The `ad_ptr` argument is a pointer to an `AudioData` object (the audio sample data). You should not modify the data itself, you should not make a copy of the data, and you should not deallocate the data. You should assume that the data persists for the lifetime of the `Resample` class; i.e., that the pointer `ad_ptr` is and remains a pointer to a valid `AudioData` object until after the `Resample` class is destroyed. The `channel` argument specifies which channel of audio data to resample (a `Resample` object only samples a single audio channel — multiple `Resample` objects can be used to sample multiple channels). The argument `factor` gives the speed up factor used for playback; e.g., a factor of 1 results in no speed up, a factor of 2 results doubles the speed, and a factor of 0.5 halves the speed.

Optionally, two loop points may be specified. If the value of `loop_bgn` is *strictly less than* that of `loop_end`, then the sample data should be continuously looped between these two indices, after the portion preceding the beginning loop point has been played. It is assumed that `loop_bgn` and `loop_end` are both less than the number of frames of audio sample data (no error checking need be done).

`operator()()` — returns the next value of the resampled data. That is, after the n–th call to the function, it returns the n–th sample. If valid loop points are not specified (that is, if `loop_bgn` is greater than or equal to `loop_end`), then this function will return 0 if the index into the original data corresponding to the index into the resampled data is greater than or equal to the number of frames of audio sample data. Linear interpolation *must* be used to compute the resampled output values.

`offsetPitch(cents)` — alters the speed up factor so as to increase the pitch of the resampled data by the specified cent value (if the value of `cents` is negative, the pitch is decreased).

Remark. The `offsetPitch` function allows the playback speed to be changed *dynamically.* That is, the offset values can change with each output sample. For instance, suppose that for the n–th output of our resampler the speed–up factor is α_n. With the next sample, the $(n+1)$–st output of the resampler, the speed–up factor can be different, say α_{n+1}. For the n–th output, the fractional index computed relative to the start of resampling is

$$s_n = \alpha_n n$$

and for the $(n+1)$–st output, the fractional index is

$$s_{n+1} = \alpha_{n+1}(n+1)$$

The difference between these two values may can be large, even if the difference between speed–up factors is small. For example, if $\alpha_{10000} = 1$ and $\alpha_{10001} = 1.1$, then the difference in fractional indices is

$$\alpha_{10001}(10001) - \alpha_{10000}(10000) = (1.1)(10001) - (1)(10000) = 1000$$

This will cause the output of the resampler to have successive values that "jump around". Note that the difference in fractional values increases with the sample count n. To remedy this, the fractional index should be computed *incrementally*:

$$s_{n+1} = \alpha_{n+1} + s_n$$

This will ensure that the difference between two successive fractional indices will remain small. The price paid here is in terms of numerical precision. If we accumulate the fractional index, we accumulate floating point errors in the process. To minimize this, accumulation variables should be double precision.

For this assignment, you must submit a single file named `Resample.cpp` containing your code for the implementation of the class. You may only include `Resample.h` and any *standard* C++ header file (the header file `AudioData.h` is included by the `Resample.h` header file).