Assignment #1

CS 225, Fall 2018

Due Wednesday, September 19

Overview

In this assignment, you will implement a C–style API for manipulating WAVE audio files. This will require you to use several of the topics and techniques discussed in class: opaque pointers, function pointers, dynamically allocated multi–dimensional arrays, bit manipulation, and binary file i/o.

The API used in this assignment is typical of many software packages. An object is encoded as pointer to an opaque data type, the details of which are hidden from the user of API, and is obtained via an object creation function. An object is then manipulated through the use of the functions in the interface. When the user is finished with an object, an object destruction function is used. Note the parallels with `C++` classes you learned about in CS 170, where an object is an instance of a class. The class constructor and destructor are used for object creation and destruction, and objects are manipulated using class member functions (methods).

API interface

I will give you the header file `Wave.h`, which declares the following interface to our API for reading, writing, and manipulating WAVE files.

```
WaveData waveRead(const char *fname);
int waveChannelCount(WaveData w);
int waveSamplingRate(WaveData w);
void waveSetFilterData(WaveData w, void *vp);
void waveFilter(WaveData w, int channel, short (*filter)(short,void*));
void waveWrite(WaveData w, const char *fname);
void waveDestroy(WaveData w);
```

Here `WaveData` is an opaque pointer to a WAVE data object. Such objects are created using the `waveRead` function, which reads WAVE data from a file. There is only one WAVE data manipulation function: `waveFilter`. To be flexible, `waveFilter` uses a function pointer to a filter function that must be of the form

```
short filterFcn(short sample_in, void *filter_data);
```

This function serves as a call–back function: it is called for each WAVE data sample. The original data sample is replaced by the return value of the filter function.

```
sample_out <-- filterFcn(sample_in,&filter_data)
```

The pointer `filter_data` is used to control the behavior of the filter, and is set using the `waveSetFilterData` function. You can think of this as supplying filter parameters. While the filter function can in principle be anything, you will implement some specific filter functions for changing the gain (volume) of the WAVE data:

```
short waveCut6DB(short in, void *data);
short waveBoost3DB(short in, void *data);
unsigned short waveBoostData(float gain);
short waveBoost(short in, void *data);
```

Here, the function `waveBoostData` is not actually a filter function, but is used to initialize the data used by the filter function `waveBoost`. After filtering, the WAVE data object can written to a file using the `waveWrite` function, and destroyed using the `waveDelete` function.

The details of the API functions are as follows.

`waveRead(fname)` — creates a WAVE data object by reading the named WAVE file `fname`. On success, the function returns a pointer to object. On failure, the null pointer `NULL` is returned.

In your implementation, you only need to be able to read a simplified WAVE file that uses 16 bit audio data. This is described at the end of this handout. There are more complicated types of WAVE files: a file structure other than a header followed by data, compressed audio data, and other bit resolutions (e.g., 8 bit, and 24 bit). If `fname` is in a form other than the simplified type described in this handout, you should return a null pointer.

`waveChannelCount(w)` — returns the number of audio channels (typically 1 or 2) of the audio data represented by the WAVE data object w. This function does not affect the audio data.

It is assumed here, and in the remaining functions, that w is a pointer to a valid (non–null and not yet destroyed) WAVE data object. It is the callers responsibility to ensure that this is the case.

`waveSamplingRate(w)` — returns the sampling rate of the audio data represented by the object w.

`waveSetFilterData(w,vp)` — sets the filter data that is to be used by the filter function specified in subsequent calls to `waveFilter`.

`waveFilter(w,channel,filter)` — filters the audio data represented by the object w. The specified filter function is only applied to the specified channel. Other channels are unaffected.

`waveWrite(w,fname)` — writes the audio data from object w to a WAVE file with name `fname`.

`waveDestroy(w)` — destroys the WAVE data object w.

The details of the filter functions are as follows. With the exception of `waveBoostData`, which is used to initialize filter data for the `waveBoost` filter function, all functions will use a filter data pointer which is set by calling `waveSetFilterData`.

`waveCut6DB(in,data)` — returns the result of scaling the value of `in` by 0.5 (in decibels, we have $20\log(0.5) \approx -6\,\text{dB}$).

In your implementation of this functions, you may **not** use explicit multiplication. You may only use bitwise operations. You will ignore the value of `data` here.

`waveBoost3DB(in,data)` — returns the result of scaling the value of `in` by 1.41 (in decibels, $20\log(1.41) \approx 3\,\text{dB}$).

Again, you may **not** use explicit multiplication. You may use only bitwise operations and addition of integers. Note that

$$1.41 \approx 1 + \tfrac{1}{2^2} + \tfrac{1}{2^3} + \tfrac{1}{2^5}$$

However, you need to be careful of overflow here: if x is a signed 16 bit (`short`) value, then $1.41x$ (when truncated to an integer) can be a larger/smaller than the largest/smallest 16 bit signed integer value. So you will need to work with signed 32 bit (`int`) values and clamp the result to the range $-(2^{15})$ to $(2^{15}-1)$. E.g.,

```
#define MAX_INT16 32767
short waveBoost3DB(short in, void *data) {
  int ix = (int)x;
  /* compute 1.41*ix */
  if (ix > MAX_INT16)
    ix = MAX_INT16;
  /* clamp to MIN_INT16 as well */
  return (short)ix;
}
```

Again, you will ignore the value of `data` here.

`waveBoost(in,data)` — returns result of scaling the value of `in` by a gain factor that is determined by the passed–in data pointer. The gain factor is a floating point value determined by

$$gain = \tfrac{b_0}{2^0} + \tfrac{b_1}{2^1} + \tfrac{b_2}{2^2} \cdots + \tfrac{b_{15}}{2^{15}} \tag{1}$$

where $b_0 b_1 b_2 \cdots b_{15}$ are the bits in the 16 bit unsigned integer n pointed to by `data`. That is, the bits in the value

```
unsigned short n = *((unsigned short*)data);
```

Once again, you are not allowed to use explicit multiplication in your implementation: only bitwise operations and integer addition. You will need to avoid numeric overflow as well.

`waveBoostData(gain)` — returns the unsigned 16 bit integer n whose bits give the value of `gain` when formula (1) is applied. So to apply a gain factor of 1.234 to channel 0 of the WAVE data object w, we would write

```
unsigned short data = waveBoostData(1.234f);
waveSetFilterData(w,&data);
waveFilter(w,0,waveBoostData);
```

I will provide you with the implementation of this function.

WAVE file format (simplified)

The simplest audio file format is the WAVE (or WAV) file format. Although it is actually organized in a hierarchical fashion, a WAVE file may be viewed as a binary file in the form

$$\text{(header)} + \text{(data)}$$

where the header gives information about audio samples in the data portion (such as the sampling rate, number of bits per sample, and the number of channels). Indeed for our purposes, the header can be written as a structure of the form

```
struct WaveHeader {
  char           riff_label[4];    // (offset  0) = {'R','I','F','F'}
  unsigned       riff_size;        // (offset  4) = 36 + data_size
  char           file_tag[4];      // (offset  8) = {'W','A','V','E'}
  char           fmt_label[4];     // (offset 12) = {'f','m','t',' '}
  unsigned       fmt_size;         // (offset 16) = 16
  unsigned short audio_format;     // (offset 20) = 1
  unsigned short channel_count;    // (offset 22) = 1 or 2
  unsigned       sampling_rate;    // (offset 24) = <anything>
  unsigned       bytes_per_second; // (offset 28) = <ignore>
  unsigned short bytes_per_sample; // (offset 32) = <ignore>
  unsigned short bits_per_sample;  // (offset 34) = 16
  char           data_label[4];    // (offset 36) = {'d','a','t','a'}
  unsigned       data_size;        // (offset 40) = <# of bytes of data>
};
```

Note that this structure has a size of 44 bytes. The header of a WAVE file is followed by a sequence of audio samples, the data, which is an array of `short` (2 byte) values:

```
|<--sample #0-->|<--sample #1-->|<--sample #2-->| ... et cetera ...
```

In the case of stereo data, the left and right channels are interleaved:

```
|L0|R0|L1|R1|L2|R2| ... et cetera ...
```

So that if there are n samples of audio data with c channels, the value of `data_size` in the above structure has a value of $2cn$ bytes.

What to turn in

You may use either `C` or `C++` for this assignment. In either case, your assignment submission will consist of a *single* source file. If you use `C`, your assignment submission should be named `Wave.c`, and you may only use the header files `Wave.h`, `stdio.h`, `stdlib.h`, and `string.h`. On the other hand if you use `C++`, your submission should be named `Wave.cpp`, and you may only use the header files `Wave.h`, `iostream`, `fstream`, and `cstring`.