Assignment: Building C-Style Libraries

Learning Outcomes

- Gaining experience in implementing C-style APIs using opaque pointers
- Gaining experience in applying a variety of techniques discussed in lectures including function pointers, bitwise operators, binary file I/O. two-dimensional arrays to solve practical problems
- Reading, understanding, and interpreting C++ code written by others

Task

In this assignment, you will implement a C-style Application Programming Interface [API] for manipulating WAVE audio files. This will require you to use several of the topics and techniques discussed in class lectures: 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 API's user, and is obtained via an object creation function. An object is then manipulated through the use of the API's functions. When the user is finished with an object, an object destruction function is used. Note the parallels with C++ classes you learned about in HLP2, 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 header file `wave.h`, which declares the following interface to our API for reading, writing, and manipulating WAVE files.

```
using WaveData = struct WaveData_tag*;
using WaveDataReadOnly = struct WaveData_tag const*;

char *waveRead(char const *file_name);
WaveData waveParse(char const *file_contents);
void waveWrite(WaveData w, char const *filename);
void waveDestroy(WaveData w);
int waveChannelCount(WaveDataReadOnly w);
int waveFrameCount(WaveDataReadOnly w);
int waveSamplingRate(WaveDataReadOnly w);
void waveSetFilterData(WaveData w, void *vp);
void waveFilter(WaveData w, int channel, short (*filter)(short, void*));
short waveRetrieveSample(WaveDataReadOnly w, int frame, int channel);
```

Here `WaveData` is an opaque pointer to a WAVE data object while `WaveDataReadOnly` is an opaque pointer to a `const` or read-only WAVE data object. A WAVE data object is created using the `waveParse` function, which reads WAVE data from a `char` buffer. The `char` buffer was previously filled with the contents of a WAVE file by the `waveRead` function. 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 callback 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 be written to a file using the `waveWrite` function, and destroyed using the `waveDelete` function.

The details of the API functions are as follows:

- `waveRead(file_name)`: opens a [binary] WAVE file `file_name` and dumps the entire contents of the file into a dynamically allocated `char` buffer. The size of this `char` buffer is exactly equivalent to the number of bytes in the binary WAVE file. The function returns the address of the first element of the dynamically allocated `char` buffer. If for some reason the file cannot be opened, you should return a `nullptr`. Note that this buffer is not terminated with a `\0` character.
- `waveParse(file_contents)`: creates a WAVE data object by parsing the contents of the `char` buffer `file_contents`. The number of bytes in this buffer will have to be determined by parsing the WAVE header. On success, the function returns a pointer to a WAVE data object. On failure, `nullptr` 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 `file_contents` contains a form other than the simplified type described in this handout, you should return `nullptr`.

  > ***It is the responsibility of function*** `waveParse` ***to free the dynamically allocated memory pointed to by*** `file_contents`.

- `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.
- `waveWrite(w,fname)`: writes audio data from object `w` to a WAVE file with name `fname`.
- `waveDestroy(w)`: destroys WAVE data object `w`. Remember that Valgrind must not report any memory leaks.
- `waveFrameCount(w)`: returns the number of audio frames or audio samples in the data represented by WAVE data object `w`.
- `waveSamplingRate(w)`: returns the sampling rate of the audio data represented by 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 object `w`. The specified filter function is only applied to the specified channel. Other channels are unaffected.
- `waveRetrieveSample(w, channel, frame)`: returns the audio sample data in WAVE file object `w` for channel `c` and frame `f`.

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 `in` by `0.5` [in decibels, we have $20 \log(0.5) \approx -6$ dB]. You'll parameter `data`.

  > ***Important: In your implementation of this function, you may not use explicit multiplication. You may only use bitwise operations.***

- `waveBoost3DB(in,data)`: returns the result of scaling the value `in` by 1.41 [in decibels, $20 \log(1.41) \approx 3$ dB]. You'll ignore parameter `data`.

  > ***Important: In your implementation of this function, you may not use explicit multiplication. You may only use bitwise operations.***

  How? Note that

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

  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 larger/smaller than the largest/smallest 16 bit signed integer value. So you'll need to work with 32 bit [`int32`] values and clamp the result to a range from $-(2^{15})$ to $(2^{15} - 1)$. For example,

  ```
  short waveBoost3DB(short x, void *data) {
    int ix = static_cast<int>(x);
    // compute 1.41*ix
    // header <limits> provides information about this class template
    if (output > std::numeric_limits<short>::max()) {
      output = std::numeric_limits<short>::max();
    }
    // clamp to the minimum value of short types as well
    return static_cast<short>(ix);
  }
  ```

- `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

  $$gain = \frac{b_{15}}{2^{15}} + \frac{b_{14}}{2^{14}} + \cdots + \frac{b_2}{2^2} + \frac{b_1}{2^1} + \frac{b_0}{2^0} \tag{1}$$

  where $b_{15}b_{14} \cdots b_2 b_1 b_0$ are the bits in the 16 bit unsigned integer n pointed to by $data$ with b_0 and b_{15} specified as the least and most significant bits of the 16 bit integer. 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,waveBoost);
```

  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

$$(header) + (data)$$

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) = <see below>
  unsigned short bytes_per_sample; // (offset 32) = <see below>
  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. For mono [1 channel] data, the array can be pictured as

$$mono: |<--sample\#0-->|<--sample\#1-->|<--sample\#2-->|...etc...$$

As each sample is 2 bytes, the size of the audio data portion of a WAVE file is

$$(\text{size of mono data in bytes}) = (2 \text{ bytes})(\text{frame count})$$

where for mono data, a frame is the same as a single audio sample. In the case of stereo [2 channel] data, the left and right channels are interleaved:

$$\begin{array}{l} stereo: \\ \end{array} \begin{array}{|c|c|c|c|c|} |<--frame\#0-->| & |<--frame\#1-->| & |<--frame\#2-->| & \\ \text{L0, R0} & \text{L1, R1} & \text{L2, R2} & \end{array} \begin{array}{l} ...etc... \\ ...etc... \end{array}$$

For stereo data, a frame consists of a left and right pair of samples. So that for stereo data, the size of the data portion of a WAVE file is

$$(\text{size of stereo data in bytes}) = (2 \text{ bytes})(2 \text{ samples per frame})(\text{frame count})$$

In general, for any number of channels per frame of audio data, the size of the data portion of a WAVE file is given by

$$(\text{size of data in bytes}) = (2 \text{ bytes})(\text{channel count})(\text{frame count})$$

In the assignment, you will need to access a given channel from a given frame of audio data. You can do this by viewing the audio data as a two-dimensional array, with each row storing the samples within a given frame. So to access the j—th channel from the i—th frame of audio data stored in the array `data16`, you can write

```
short x = data16[c*i+j];
```

where `c` is the number of channels [`c` is 1 for mono and 2 for stereo].

The `bytes_per_second` (offset 28) and `bytes_per_sample` (offset 32) fields contain information that is used primarily for compressed data. As we are using uncompressed data, the values are redundant: they are computed using the formulas

$$(\text{bytes per second}) = (2 \text{ bytes})(\text{channel count})(\text{sampling rate})$$
$$(\text{bytes per sample}) = (2 \text{ bytes})(\text{channel count})$$

Some implementations will ignore these fields, others do not.

Submission Details

Header and source files

There is no header file to be submitted. Your implementation must be placed in `wave.cpp`, and this will be the only file that you will submit.

Compiling, executing, and testing

Download `wave.h`, `wave.cpp`, `wavetest.cpp`, and `wave-files.zip` containing input and *correct* output WAVE files. You'll have to edit one of the previous `makefile`s [from previous assignments] to make it suitable for this assignment.

Some of the filter functions have to ignore a parameter. To avoid the compiler from issuing warnings [which will then flag an error], you must add the `-Wno-unused-parameter` switch when compiling with `g++` and `clang++`.

Suppose you have created an executable `wavetest.out`. You are given two input WAVE files to test your code with: `WaveTest.1.wav` and `WaveTest.2.wav`. The first file is mono [one channel], and the second is stereo [two channels]. You're also given the two corresponding output WAVE files: `WaveTest.1.output.wav` and `WaveTest.2.output.wav`, that give the expected results of running the test driver `wavetest.cpp` with the respective input file, that is, running

```
$ ./wavetest WaveTest.1.wav
```

produces output file `WaveTest.wav`. This file should be the same as file `WaveTest.1.output.wav`. To see if the files are the same, you can use the Audacity program which is straightforward to install on your machines. When you open a [valid] WAVE file in Audacity, a graphical depiction of the audio data will be displayed. You can visually compare your output file with the expected output file. Another option to compare two binary files is to use a hex editor. A third option is to use the Linux `cmp` program to perform a byte-by-byte comparison of two files. You can get more information about `cmp` by running the `man` command.

Valgrind is required

Since your code is interacting directly with physical memory, things can go wrong at the slightest provocation. The range of problems that can arise when writing code dealing with low-level details has been covered in lectures [and a handout that goes into all the ugly details of memory errors and leaks]. You should use Valgrind to detect any potential issues that may lurk under the surface. The online server will execute your submission through Valgrind. If Valgrind detects memory-related problems with your submission, the diagnostic output generated by Valgrind will be added to your program's output causing that output to be different than the correct output.

Documentation

This module will use Doxygen to tag source and header files for generating html-based documentation. Every source and header file *must* begin with *file-level* documentation block. Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. If you're not sure of what to do, read the introduction to Doxygen on the course web page plus the necessary requirements for submission in assignment 0.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the source file.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - F grade if your `waveCut6DB` or `waveBoost3DB` or `waveBoost` functions perform explicit multiplication. Remember they're required to be implemented using bitwise operators.
    - F grade if Valgrind reports memory errors and/or leaks.
    - The auto-grader will use different files and different scale values for the filters. It will require your output `.wav` files ***to be exactly equivalent*** to the correct `.wav` files. I will test with a mono file and a stereo file. If your program generates the correct result for both files, the submission will be assigned an A grade. If your program generates incorrect results for both files [even if they differ by a single byte], your submission will be assigned a F grade.
    - A deduction of one letter grade for each missing documentation block in submission. Your submission must have **one** file-level documentation block and **twelve** function-level documentation blocks. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted [with irrelevant information from some previous assessment] block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an A grade and one documentation block is missing, your grade will be later reduced from A to B. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to E.