A1 Memory and Files

Assignment 1: Memory and Files

Due: Electronically, by 10:00 PM on Jan 29

Note: Due to some bugs in the solution code that generated the expected output files, please see corrections below. If you have already done the assignment and your output matches the original output files, then you do not need to change your code, and I'm sorry that we didn't detect the problem earlier and save you some time. Please just submit it. If you are just getting started on the assignment, please use the new output files.

Introduction

For this assignment, you'll write two C programs that manipulate sounds. The first program, `remvocals`, removes vocals from stereo sound files; the second, `addecho`, adds echo to a mono sound file. You'll use C commandline-processing, IO, and memory functions.

Sounds are waves of air pressure. When a sound is generated, a sound wave consisting of compressions (increases in pressure) and rarefactions (decreases in pressure) moves through the air. This is similar to what happens if you throw a stone into a pond: the water rises and falls in a repeating wave.

When a microphone records sound, it takes a measure of the air pressure and returns it as a value. These values are called samples and can be positive or negative corresponding to increases or decreases in air pressure. Each time the air pressure is recorded, we are **sampling** the sound. Each sample records the sound at an instant in time; the faster we sample, the more accurate is our representation of the sound. The **sampling rate** refers to how many times per second we sample the sound. For example, CD-quality sound uses a sampling rate of 44100 samples per second; sampling someone's voice for use in a VOIP conversation uses far less than this. Sampling rates of 11025 (voice quality), 22050, and 44100 (CD quality) are common.

For mono sounds (those with one sound channel), a sample is simply a positive or negative integer that represents the amount of compression in the air at the point the sample was taken. For stereo sounds, a sample is actually made up of two integer values: one for the left speaker and one for the right.

Program 1: `remvocals.c`

For this part, you'll write a C program to remove vocals from **stereo** PCM sounds stored in the canonical wav format. You can read more about this wav format (though this won't be too helpful until you get to part 2 of this assignment).

For example, your program will be able to start with a wav file with vocals and produce a wav file with vocals removed. (Is that Dan at UTM singing? You'll never know.)The original output wav file was incorrect and has been replace.

For this part, you will write a program that supports the following synopsis (**synopsis** just means that we are telling you the correct way that you program is to be called):

```
remvocals sourcewav destwav
```

where `sourcewav` is the **stereo** wav sound file that already exists, and `destwav` is the new stereo output wav file to create that has vocals removed. That is, your program should take two commandline parameters: the first is the input wav file, and the second is the output wav file. If exactly two parameters are not provided, or if one of the files cannot be opened, issue appropriate errors and terminate. Be sure to open both the input and output files in binary mode.
Here's how the algorithm works.

- Copy the first 44 bytes from the input file to the output file without changing those bytes. Those 44 bytes contain important header information that should not be modified by your program.
- Next, treat the rest of the input file as a sequence of `short`s. Take each pair of shorts `left` and `right`, and compute `combined = (left - right) / 2`. Write two copies of`combined` to the output file.

Please use the C stdio block functions (King 22.6) `fread` and `fwrite`, not the low-level Unix functions like `read` and `write`. Be sure to include all necessary error-checking.

Why Does This Algorithm Work?

For the curious, a brief explanation of the vocal-removal algorithm is in order. As you noticed from the algorithm, we are simply subtracting one channel from the other (and then dividing by 2 to keep the volume from getting too loud). So why does subtracting the left channel from the right channel magically remove vocals?

When music is recorded, it is sometimes the case that vocals are recorded by a single microphone, and that single vocal track is used for the vocals in both channels. The other instruments in the song are recorded by multiple microphones, so that they sound different in both channels. Subtracting one channel from the other takes away everything that is ``in common'' between those two channels which, if we're lucky, means removing the vocals.

Of course, things rarely work so well. Try your vocal remover on this badly-behaved wav file. Sure, the vocals are gone, but so is the body of the music! Apparently, some of the instruments were also recorded ``centred'', so that they are removed along with the vocals when channels are subtracted.

Program 2: `addecho.c`

Listen to this crow and that same crow, with echo. Or this woman saying welcome and that same welcome, with echo. In both cases, echo is being added to a sound. We want to write a

program that takes an existing wav file and creates a new wav file that has echo added to it. Note that these two files haven't changed.

How is echo added? There are two important ideas to understand here:

- Volume: multiplying each sample value by a number greater than 1 increases the volume; multiplying each sample by a number between 0 and 1 (exclusive) decreases the volume. For example, multiplying everything by 2 doubles the volume; multiplying everything by 0.5 halves the volume.
- Mixing: if we mix two sounds together, what we mean is that they play at the same time. Mixing two sounds involves adding corresponding samples together. For example, if one sound has three samples: 2, 4, and 6; and another sound has four samples: 10, 11, 12, and 13; mixing them yields a sound of four samples: 12, 15, 18, and 13.

Suppose we have two parameters: `delay` and `volume_scale`. `delay` tells us how many samples to wait before mixing-in the echoing copy of the sound with the original copy of the sound. `volume_scale` tells us the amount by which to scale down the volume of the echoing copy of the sound. For example, a setting of 2 for `volume_scale` means to divide the volume of the original sound by 2 when producing the echo.

Now we can describe conceptually how to add echo to a sound. We will have two sounds: (1) the original sound `orig`, and (2) a new sound `echo` that has `delay` samples of value 0, followed by a volume-scaled version of `orig`. Mix `orig` and `echo` together and output that as a new sound file. That new sound file will have the echo added to it, and it will have the same number of samples as `echo`.

Unfortunately, that technique would require the entire `orig` and `echo` sounds to be loaded into memory. Sound files can be huge, so that technique isn't advised. Instead, for this assignment, you are required to do the following:

- Keep only the most recent `delay` samples of the original sound, scaled by `volume`, in an echo buffer.
- When writing the output sound, there are several phases:
    - If you are not yet at sample `delay`, just write the same sample from `orig` to your output sound. (Why? Because we can't start the echo yet!)
    - If you are at sample `delay` or later, start mixing-in the samples from the echo buffer.
    - Once you've finished reading the original sound, write `x` 0 samples to the output sound, where `x` is `delay` minus the number of samples in `orig`. If `x` is zero or less, do nothing for this step. (Why are we doing this? Because the original sound might not have `delay` samples in it, and we have to wait a total of `delay` samples before producing the echo.)
    - Finally, write the rest of the echo buffer to the output sound. (Without doing this, the echoing copy of the sound would be cut off before it finishes.)

For this part, you will write a program that supports the following synopsis:

`addecho [-d delay] [-v volume_scale] sourcewav destwav`
where `delay` and `volume_scale` are as described above, `sourcewav` is the **mono** wav sound file that already exists, and `destwav` is the new mono output wav file to create that includes the echo. If `-d` is not provided, use a default value of 8000; if `-v` is not provided, use a default value of 4. (Note: the mono sound files for this part of the assignment have a sampling rate of 22050. This means, for example, that a delay of 8000 samples causes the echo to start `8000 / 22050 = 0.3628` seconds after the sound starts.)
You are required to process commandline options using `getopt`. (`man 3 getopt` is very helpful; don't use just `man getopt`, because that gives you a shell command, not the C library function we want. Also, check `man 3 strtol` for a function that converts strings to integers.)

Remember in `remvocals` how we wrote the first 44 bytes of the wav file without changing them? Well, we can't do that for `addecho`, because within those 44 bytes are two integers that each tell us the size of the wav file. We have to increase each of those integers by `delay * 2`, otherwise they will not accurately represent the length of the wav files with echo in them! If we number the bytes 0 to 43, then the integers start at bytes 4 and 40 (or shorts 2 and 20), respectively. Be sure to fix both! Here is a hint for some code that you may want to use:

```
#define HEADER_SIZE 22
short header[HEADER_SIZE];
/* code omitted that reads from the input file into header */

/* The next line says that we want to treat the memory starting at the
 * address header + 2 as if it were an unsigned int */
sizeptr = (unsigned int *)(header + 2);
```

One more important difference between this `addecho` program and `remvocals`: in `addecho`, we don't know the size of the echo buffer until runtime. You are required to use `malloc`to allocate the proper amount of memory based on the `delay` parameter.

More Examples

Here we have a door slamming. It is a sound with 32512 samples in it. (You can tell because `32512*2 + 44 = 65068` bytes, and that is the size of the wav file in bytes.) Here are some calls of `addecho`; in each case, we start with the original `door.wav` and use various `delay` and `volume_scale` parameters.

The first set of output files are the ones that match the specification:

- `addecho -d 12000 -v 4 door.wav new_door_12000_4.wav`. File Created
- `addecho -d 12000 -v 2 door.wav new_door_12000_2.wav`. File Created
- `addecho -d 20000 -v 4 door.wav new_door_20000_4.wav`. File Created
- `addecho -d 35000 -v 2 door.wav new_door_35000_2.wav`. File Created

- `addecho -d 60000 -v 1 door.wav new_door_60000_1.wav`. File Created

The following set of wav files are the original ones that were created with buggy solution code (yell at Dan). We will accept programs that match these output files, but if you are just getting start on the assignment, do **not** use these files.

- `addecho -d 12000 -v 4 door.wav door_12000_4.wav`. File Created
- `addecho -d 12000 -v 2 door.wav door_12000_2.wav`. File Created
- `addecho -d 20000 -v 4 door.wav door_20000_4.wav`. File Created
- `addecho -d 35000 -v 2 door.wav door_35000_2.wav`. File Created
- `addecho -d 60000 -v 1 door.wav door_60000_1.wav`. File Created

Here is a very short wav file with only 20 samples that is easier to compare to the expected output. Running `addecho -d 3 -v 2 short.wav short_3_2.wav` produces the fileshort_3_2.wav.(This file is too short to listen to.)

```
# The first line shows the samples in short.wav
# The second line shows the samples in short_3_2.wav
2  2  2  2  2  8  8  8  8  8 16 16 16 16 16  4  4  4  4  4
2  2  2  3  3  9  9  9 12 12 20 20 20 24 24 12 12 12  6  6 2 2 2
```

(You should be able to produce similar short files for testing fairly easily.)

Comparing Binary Files

For full marks, the wav files created by your programs must match our sample wav files exactly. Wav files are binary files, and a useful Unix utility for comparing binary files is `cmp`. Read the man page for `cmp`, especially its `-l` option. This can tell you whether two files differ and, if they do, the bytes in the files that do not match. Note that `cmp -l` outputs byte offsets in decimal but byte values themselves in octal (find a converter online if you do not know how to convert from octal to decimal).

No Crashes

Your programs should never segfault or crash. For example:

- Be sure to error-check all C library calls, including `malloc`, `fopen`, and so on.
- Make sure the user calls your programs correctly; if not, print an informative error message and exit. In particular, this means that you should not allow incorrect calls of your programs to crash or even get passed the argument-checks you do.

What to submit

You will commit to your repository in the `a1` directory two files: `remvocals.c` and `addecho.c`. Only files with exactly these names will be graded. You may submit additional files (e.g. test files), but they will not be marked.

Make sure that you've done `svn add` on every file. Check MarkUs, or checkout your repository into a new directory, to verify that everything is there. There's nothing we can do if you forget to submit a file.

Submission checklist:

You can make your life a lot simpler by ensuring that your submission is complete. When you first sit down to do some work on this assignment, check out the repository, create a file called remvocals.c with a main function that you can compile without error. Do the same with addecho.c. Then use svn to add and commit the file to your repo. After you have added these files to svn, you can commit your work after every time you do some work.

Here is a list of things that you should do at least an hour before the assignment deadline:

- Create a temporary directory in your account (not one that is a subdirectory of your working directory for your repository).
- Check out your repository into this empty directory to be sure that the correct files have been committed.
- Run `gcc -Wall -g -o remvocals remvocals.c` and `gcc -Wall -g -o addecho addecho.c` to be sure that both files compile without warnings or error.
- Run a few of the examples shown above and compare the output to be sure that you have committed the right version of your code.