This is CS50

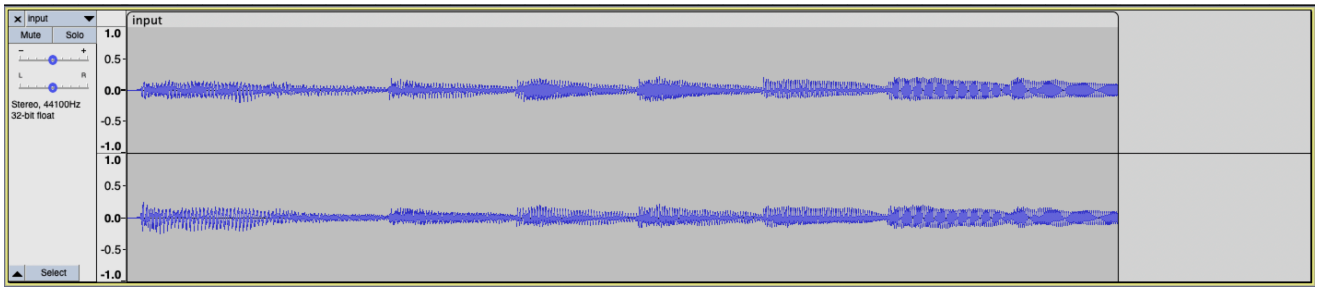
CS50's Introduction to Computer Science

OpenCourseWare

Donate (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Volume



Problem to Solve

WAV files (https://docs.fileformat.com/audio/wav/) are a common file format for representing audio. WAV files store audio as a sequence of "samples": numbers that represent the value of some audio signal at a particular point in time. WAV files begin with a 44-byte "header" that contains information about the file itself, including the size of the file, the number of samples per second, and the size of each sample. After the header, the WAV file contains a sequence of samples, each a single 2-byte (16-bit) integer representing the audio signal at a particular point in time.

Scaling each sample value by a given factor has the effect of changing the volume of the audio. Multiplying each sample value by 2.0, for example, will have the effect of doubling the volume of the origin audio. Multiplying each sample by 0.5, meanwhile, will have the effect of cutting the volume in half.

In a file called `volume.c` in a folder called `volume`, write a program to modify the volume of an audio file.

Demo

Recorded with **asciinema**

Distribution Code

For this problem, you'll extend the functionality of code provided to you by CS50's staff.

▼ **Download the distribution code**

Log into cs50.dev (https://cs50.dev/), click on your terminal window, and execute `cd` by itself. You should find that your terminal window's prompt resembles the below:

```
$
```

Next execute

```
wget https://cdn.cs50.net/2023/fall/psets/4/volume.zip
```

in order to download a ZIP called `volume.zip` into your codespace.

Then execute

```
unzip volume.zip
```

to create a folder called `volume`. You no longer need the ZIP file, so you can execute

```
rm volume.zip
```

and respond with "y" followed by Enter at the prompt to remove the ZIP file you downloaded.

Now type

```
cd volume
```

followed by Enter to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
volume/ $
```

If all was successful, you should execute

```
ls
```

and see a file named `volume.c`. Executing `code volume.c` should open the file where you will type your code for this problem set. If not, retrace your steps and see if you can determine where you went wrong!

Implementation Details

Complete the implementation of `volume.c`, such that it changes the volume of a sound file by a given factor.

- The program should accept three command-line arguments. The first is `input`, which represents the name of the original audio file. The second is `output`, which represents the name of the new audio file that should be generated. The third is `factor`, which is the amount by which the volume of the original audio file should be scaled.
  - For example, if `factor` is `2.0`, then your program should double the volume of the audio file in `input` and save the newly generated audio file in `output`.
- Your program should first read the header from the input file and write the header to the output file.
- Your program should then read the rest of the data from the WAV file, one 16-bit (2-byte) sample at a time. Your program should multiply each sample by the `factor` and write the new sample to the output file.
  - You may assume that the WAV file will use 16-bit signed values as samples. In practice, WAV files can have varying numbers of bits per sample, but we'll assume

16-bit samples for this problem.

- Your program, if it uses `malloc`, must not leak any memory.

Hints

Click the below toggles to read some advice!

▼ Understand the code in `volume.c`

Notice first that `volume.c` is already set up to take three command-line arguments, `input`, `output`, and `factor`.

- `main` takes both an `int`, `argc`, and an array of `char *`s (strings!), `argv`.
- If `argc`, the number of arguments at the command-line including the program itself, is not equal to 4, the program will print its proper usage and exit with status code 1.

```
int main(int argc, char *argv[])
{
    // Check command-line arguments
    if (argc != 4)
    {
        printf("Usage: ./volume input.wav output.wav factor\n");
        return 1;
    }

    // ...
}
```

Next, `volume.c` uses `fopen` (https://manual.cs50.io/3/fopen) to open the two files provided as command-line arguments.

- It's best practice to check if the result of calling `fopen` is `NULL`. If it is, the file wasn't found or wasn't able to be opened.

```
// Open files and determine scaling factor
FILE *input = fopen(argv[1], "r");
if (input == NULL)
{
    printf("Could not open file.\n");
    return 1;
}

FILE *output = fopen(argv[2], "w");
if (output == NULL)
{
    printf("Could not open file.\n");
    return 1;
}
```

Later, these files are closed with `fclose`. Whenever you call `fopen`, you should later call `fclose`!

```
// Close files
fclose(input);
fclose(output);
```

Before closing the files, though, notice that we have a few TODOs.

```
// TODO: Copy header from input file to output file

// TODO: Read samples from input file and write updated data to output file
```

Odds are you'll need to know the factor by which to scale the volume, hence why `volume.c` already converts the third command-line argument to a `float` for you!

```
float factor = atof(argv[3]);
```

▼ Copy WAV header from input file to output file

Your first TODO is to copy the WAV file header from `input` and write it to `output`. First, though, you'll need to learn about a few special data types.

So far, we've seen a number of different types in C, including `int`, `bool`, `char`, `double`, `float`, and `long`. However, inside a header file called `stdint.h` are the declarations of a number of *other* types that allow us to very precisely define the size (in bits) and sign (signed or unsigned) of an integer. Two types in particular will be useful to us when working with WAV files:

- `uint8_t` is a type that stores an 8-bit (hence `8`!) unsigned (i.e., not negative) integer (hence `uint`!). We can treat each byte of a WAV file's header as a `uint8_t` value.
- `int16_t` is a type that stores a 16-bit signed (i.e., positive or negative) integer. We can treat each sample of audio in a WAV file as an `int16_t` value.

You'll likely want to create an array of bytes to store the data from the WAV file header that you'll read from the input file. Using the `uint8_t` type to represent a byte, you can create an array of `n` bytes for your header with syntax like

```
uint8_t header[n];
```

replacing `n` with the number of bytes. You can then use `header` as an argument to `fread` (https://manual.cs50.io/3/fread) or `fwrite` (https://manual.cs50.io/3/fwrite) to read into or write from the header.

Recall that a WAV file's header is always exactly 44 bytes long. Note that `volume.c` already defines a variable for you called `HEADER_SIZE`, equal to the number of bytes in the header.

The below is a pretty big hint, but here's how you could accomplish this TODO!

```
// Copy header from input file to output file
uint8_t header[HEADER_SIZE];
fread(header, HEADER_SIZE, 1, input);
fwrite(header, HEADER_SIZE, 1, output);
```

▼ Write updated data to output file

Your next TODO is to read samples from `input`, update those samples, and write the updated samples to `output`. When reading files, it's common to create a "buffer" in which to temporarily store data. There, you can modify the data and—once it's ready—write the buffer's data to a new file.

Recall that we can use the `int16_t` type to represent a sample of a WAV file. To store an audio sample, then, you can create a buffer variable with syntax like:

```
// Create a buffer for a single sample
int16_t buffer;
```

With a buffer for samples in place, you can now read data into it, one sample at a time. Try using `fread` for this task! You can use `&buffer`, the address of `buffer`, as an argument to `fread` or `fwrite` to read into or write from the buffer. (Recall that the `&` operator is used to get the address of the variable.)

```
// Create a buffer for a single sample
int16_t buffer;

// Read single sample into buffer
fread(&buffer, sizeof(int16_t), 1, input);
```

Now, to increase (or decrease) the volume of a sample, you need only multiply it by some factor.

```
// Create a buffer for a single sample
int16_t buffer;

// Read single sample into buffer
fread(&buffer, sizeof(int16_t), 1, input);

// Update volume of sample
buffer *= factor;
```

And finally, you can write that updated sample to `output`:

```
// Create a buffer for a single sample
int16_t buffer;

// Read single sample from input into buffer
fread(&buffer, sizeof(int16_t), 1, input);

// Update volume of sample
```

```
buffer *= factor;

// Write updated sample to new file
fwrite(&buffer, sizeof(int16_t), 1, output);
```

There's just one problem: you'll need to *continue* reading a sample into your buffer, updating its volume, and writing the updated sample to the output file while there are still samples left to read.

- Thankfully, per its documentation, `fread` will return the number of items of data successfully read. You may find this useful to check for when you've reached the end of the file!
- Keep in mind there's no reason you can't call `fread` inside of a `while` loop's conditional. You could, for example, make a call to `fread` like the following:

  ```
  while (fread(...))
  {

  }
  ```

It's quite the hint, but see the below for an efficient way to solve this problem:

```
// Create a buffer for a single sample
int16_t buffer;

// Read single sample from input into buffer while there are samples left to read
while (fread(&buffer, sizeof(int16_t), 1, input) != 0)
{
    // Update volume of sample
    buffer *= factor;

    // Write updated sample to new file
    fwrite(&buffer, sizeof(int16_t), 1, output);
}
```

Because the version of C you're using treats non-zero values as `true` and zero values as `false`, you could simplify the above syntax to the following:

```
// Create a buffer for a single sample
int16_t buffer;

// Read single sample from input into buffer while there are samples left to read
while (fread(&buffer, sizeof(int16_t), 1, input))
{
    // Update volume of sample
    buffer *= factor;

    // Write updated sample to new file
    fwrite(&buffer, sizeof(int16_t), 1, output);
}
```

Walkthrough

▼ Not sure how to solve?

How to Test

Your program should behave per the examples below.

```
$ ./volume input.wav output.wav 2.0
```

When you listen to `output.wav` (as by control-clicking on `output.wav` in the file browser, choosing **Download**, and then opening the file in an audio player on your computer), it should be twice as loud as `input.wav`!

```
$ ./volume input.wav output.wav 0.5
```

When you listen to `output.wav`, it should be half as loud as `input.wav`!

Correctness

```
check50 cs50/problems/2024/x/volume
```

Style

```
style50 volume.c
```

How to Submit

```
submit50 cs50/problems/2024/x/volume
```