

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.clubhouse.com/@davidjmalan>) 

(<https://www.facebook.com/dmalan>)  (<https://github.com/dmalan>) 

(<https://www.instagram.com/davidjmalan/>)  (<https://www.linkedin.com/in/malan/>)

 (<https://orcid.org/0000-0001-5338-2522>) 

(<https://www.quora.com/profile/David-J-Malan>) 

(<https://www.reddit.com/user/davidjmalan>) 

(<https://www.tiktok.com/@davidjmalan>)  (<https://davidjmalan.t.me/>) 

(<https://twitter.com/davidjmalan>)

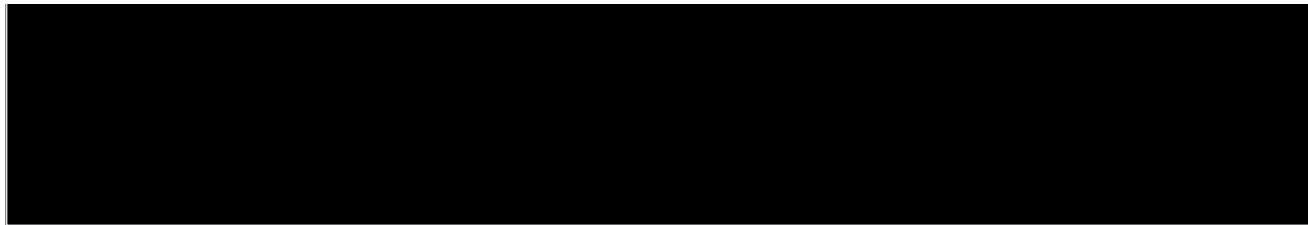
Reverse

Implement a program that reverses a WAV file, per the below.

```
./reverse input.wav output.wav
```

Background





In Electric Light Orchestra’s “Fire on High”, there’s something a little off about the first minute or so of the music. If you take a listen, it sounds almost like the audio is playing backwards. As it turns out, if you play the beginning section of the song in reverse, you’ll hear the following:

“The music is reversible. Time is not. Turn back, turn back!”

Creepy, right? This is a technique called “backmasking,” or hiding messages in music that can only be heard when the song is played backwards. Many artists have used (or been suspected of using) this technique in their songs. To be able to do our own investigation into backmasking, we’ve asked you to write a program that can reverse WAV files for us!

Unlike MP3 audio files, WAV files not compressed. This makes the files much easier to edit and manipulate, which is useful for the task at hand. To learn a little more about WAV files, we need to take a closer look at the WAV file format.

Getting Started

Open [VS Code \(https://code.cs50.io/\)](https://code.cs50.io/).

Start by clicking inside your terminal window, then execute `cd` by itself. You should find that its “prompt” resembles the below.

```
$
```

Click inside of that terminal window and then execute

```
wget https://cdn.cs50.net/2022/fall/psets/4/reverse.zip
```

followed by Enter in order to download a ZIP called `reverse.zip` in your codespace. Take care not to overlook the space between `wget` and the following URL, or any other character for that matter!

Now execute

```
unzip reverse.zip
```

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

```
rm reverse.zip
```

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

Now type

```
cd reverse
```

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

```
reverse/ $
```

If all was successful, you should execute

```
ls
```

and see a file named `reverse.c`. Executing `code reverse.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!

The WAV File Format

Notice that, in the visual below, a WAV file is broken into three chunks. Each chunk has a few blocks of data inside of it.

The first chunk contains information about the file’s type. In particular, see how the “File Format” block in the first chunk spells out ‘W’ ‘A’ ‘V’ ‘E’ in bytes 8–11, to indicate the file is a WAV file.

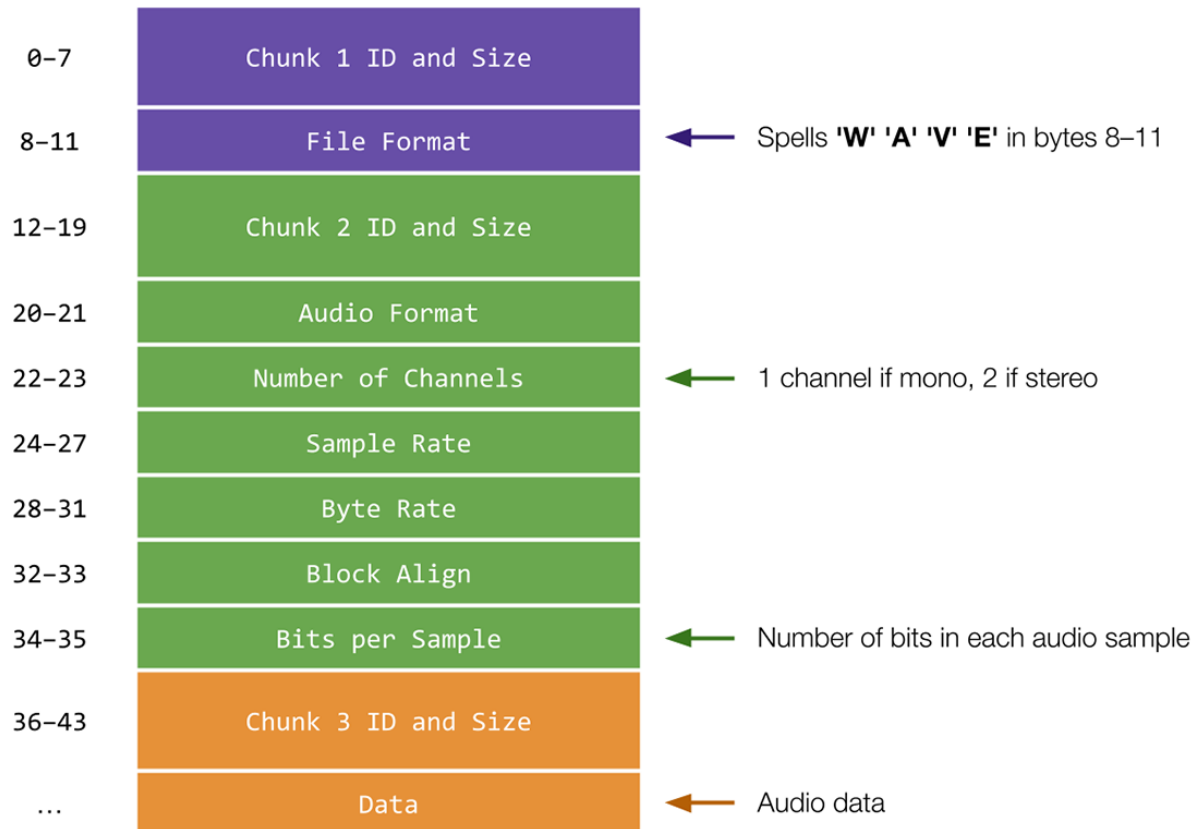
The second chunk contains information about the upcoming audio data, including how many “channels” of audio are present and how many bits are in each audio “sample”. Audio files have 1 channel when they’re “monophonic”: if you were to wear headphones, you’d hear the same audio in your left and right ear. Audio files have 2 channels when they’re “stereophonic”: wearing headphones, you’d hear slightly different audio in your left and right ear, creating a sense of spaciousness. Samples are the individual chunks of bits which make up the audio you hear. With more bits per sample, an audio file can have greater clarity (at the cost of more memory used!).

Finally, the third chunk contains the audio data itself—those samples we mentioned just above.

Everything before the audio data is considered part of the WAV “header”. Recall that a file header is simply some metadata about the file. In this case, the header is 44 bytes long.

Offset
(bytes)

Block Title



A more technical explanation of WAV headers can be found [here](http://soundfile.sapp.org/doc/WaveFormat/) (<http://soundfile.sapp.org/doc/WaveFormat/>), which is the resource by which this visual was

inspired. Notice that we've included a file, `wav.h`, which implements all these details for you in a struct called `WAVEHEADER`.

Specification

Let's write a program called `reverse` that enables us to reverse a WAV file given by the user and create a new WAV file that contains the resulting reversed audio. For simplicity's sake, we'll limit the files we deal with to the WAV format. At the time the user executes the program, they should provide, using two command-line arguments, the name of the input file to be read and reversed, and the name of the output file they would like to save the resulting audio in. A successfully executed program should not output any text, and should create a WAV file with the user-specified name that plays the audio of the input WAV file in reverse. For example:

```
$ ./reverse input.wav output.wav
```

In `reverse.c`, you'll notice that a few helpful libraries have been included, as well as a header file, `wav.h`. You'll likely find these to be useful when implementing your program. We've left eight `TODO`s and two helper functions for you to fill in, and we recommend you tackle them in order from 1 to 8.

- In the first `TODO`, you should ensure the program accepts two command-line arguments: the name of the input WAV file and the name of the output WAV file. If the program does

not meet these conditions, you should print an appropriate error message and return `1`, ending the program.

⊕ Hint

- In the second `TODO`, you should open your input file. We'll need to open the input file in "read-only" mode, since we'll only read data from the input file. It may be wise to check that the file has been opened successfully. Otherwise, you should print an appropriate error message and return `1`, exiting the program. We should hold off on opening the output file, though, lest we create a new WAV file before knowing the input file is valid!

⊕ Hint

- In the third `TODO`, you should read the header from the input file. Recall that, in `wav.h`, we've already implemented a struct that can store a WAV file's header. Since we've written `#include "wav.h"` at the top of `reverse.c`, you, too, can use the `WAVHEADER` struct.
- In the fourth `TODO`, you should complete the `check_format` function. `check_format` takes a single argument, a `WAVHEADER` called `header`, representing a struct containing the input file's header. If `header` indicates the file is indeed a WAV file, the `check_format` function should return `true`. If not, `check_format` should return `false`. To check if a file is of the WAV format, we can compare the elements from the input file header to those we would expect from a WAV file. It suffices to show the

"WAVE" marker characters are found in the `format` member of the `WAVHEADER` struct (see [Background](#) for more detail on WAV file headers).

- In the fifth `TODO`, you can now safely open the output file for writing. It would still be wise to check that the file has been opened successfully.

⊕ Hints

This may be a good place to stop and test that your program behaves as expected. If implemented properly, your program should open a new file when executed with the proper command-line arguments.

If at any point you find it necessary to delete a file, execute the following command in your current working directory.

```
$ rm file_name.wav
```

If you'd rather not be prompted to confirm each deletion, execute the command below instead.

```
$ rm -f file_name.wav
```

Just be careful with that `-f` switch, as it "forces" deletion without prompting you.

- Next, now that the file type has been verified, the sixth `TODO` tells us to write the header to the output file. The reversed WAV file will still have the same underlying file structure

as the input file (same size, number of channels, bits per sample, etc.), so it suffices to copy the header we read in from the input file in the third `TODO` to the output file.

- In the seventh `TODO`, you should implement the `get_block_size` function. `get_block_size`, like `check_format`, takes a single argument: this is a `WAVHEADER` called `header`, representing the struct containing the input file's header. `get_block_size` should return an integer representing the **block size** of the given WAV file, in bytes. We can think of a *block* as a unit of auditory data. For audio, we calculate the size of each block with the following calculation: **number of channels** multiplied by **bytes per sample**. Luckily, the header contains all the information we need to compute these values. Be sure to reference the [Background](#) section for a more in-depth explanation as to what these values mean and how they are stored. See too `wav.h`, to determine which members of `WAVHEADER` might be useful.

⊕ Hints

- The eighth and final `TODO` is where the actual reversing of the audio takes place. To do this, we need to read in each block of auditory data starting from the very end of the input file and moving backwards, simultaneously writing each block to the output file so they are written in reverse order. First, we should declare an array to store each block we read in. Then, it's up to you to iterate through the input file audio data. You'll want to be sure you read through all of the audio, but don't erroneously copy any of the data from the header! Additionally, for testing purposes, we would like to maintain the order of the channels for each audio block. For example, in a WAV file with two channels (stereophonic sound), we want to make sure that the first channel of the last audio block in the input becomes the first channel of the first audio block in the output.

⊕ Hints

- Finally, be sure to close any files you've opened!

Usage

Here are a few examples of how the program should work. For example, if the user omits one of the command-line arguments:

```
$ ./reverse input.wav
Usage: ./reverse input.wav output.wav
```

Or if the user omits both of the command-line arguments:

```
$ ./reverse
Usage: ./reverse input.wav output.wav
```

Here's how the program should work if the user provides an input file that is not an actual WAV file:

```
$ ./reverse image.jpg output.wav
```

```
Input is not a WAV file.
```

You may assume the user enters a valid output filename, such as `output.wav`.

A successfully executed program should not output any text, and should create a WAV file with the user-specified name that plays the audio of the input WAV file in reverse. For example:

```
$ ./reverse input.wav output.wav
```

Testing

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2023/x/reverse
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 reverse.c
```

How to Submit

In your terminal, execute the below to submit your work.

```
submit50 cs50/problems/2023/x/reverse
```

