Install, Test, and Modify ffmpeg

Please remember that this is a pair-programming project. You *must* have a partner. You or your partner's latest file submission (any submission) will be graded and scored according to that date. Please have a contingency plan in place to deal with personal issues or internet outages as no extensions will be given for personal/technical issues.

You should submit the five required files through Gradescope. (No peer review on this one, so no Canvas submission.)

Checkpoint requirements

1. Select a partner (required). No partner = no credit. Follow these pair programming guidelines. Start a log of the time you spend and the work you do on this project.
2. Review the project specifications and ffmpeg installation instructions (available here). You will learn about the scope of the project and use the installation instructions.
3. Download, install, and test ffmpeg. One or both of the partners may install ffmpeg. It is required that you change your ffmpeg directory permissions so that the directory is not accessible by other students.
   If you and your partner would like to maintain separate files, please choose and use version control software on your own. (You learned about this in CS 3500.) I recommend Git, as you will use Git for cloning the ffmpeg source code. Make sure that any repository you choose is inaccessible to other students (private).

Alter ffmpeg so that when it loads .bmp image files, or .pcm or .wav audio files, it prints out two lines like this to the screen:

```
    *** CS 3505 Spring 2020:  Running code in (place function and file names here) ***
```

4. ```
   *** CS 3505 Spring 2020:  Changed by (place your names here) ***
   ```
   Find the source code in ffmpeg that reads *and decodes* each type of file, and alter it so that it prints out the required lines *exactly once if a file of that type is decoded*. If multiple files of the same type are loaded at the same time, do not print out the information multiple times. (This can be difficult to test -- feel free to discuss testing with other teams. It's very easy to solve with a local static variable.)
   ```

You must find the code that decodes each type of image or sound, and modify it. It is not sufficient to modify ffplay's main, or to find the code that parses filenames. The goal is for you to find the code that actually processes the bytes of images or sounds as they are read in. (This helps you later on.) Use ffmpeg's functions for outputting log messages, and choose an appropriate message type. Confine your changes to three .c files. The audio and video codecs may be in different directories.
You will need to locate your own test images/sounds and you will need to perform a variety of tests. Note that ffmpeg can load a .wav and convert it to a .pcm file for you, but you'll need to look up the command line parameters that make this work. (It's great for converting files...)

5. Both partners should submit exactly five files (unzipped):
    - In Gradescope, find the assignment, then select your partner for this assignment (required). Your submission will count for the both of you.
    - Submit the three .c files you modified.
    - Submit a three line text file named "locations.txt" that contains the relative pathnames of each file you modified, one per line. (We will use this to copy in your files.) Each line should start with "ffmpeg/" and finish with the rest of the relative pathname (including the filename) for each file you changed. This file will be automatically parsed, so don't include other text of any kind. Your file should just contain three valid relative pathnames, one per line, that shows us where to copy your files.
    - Submit a short text document titled "partner.pdf". In it, list your name and your partner's name, and report the time and effort that you and your partner put into the project. A simple log is sufficient, but it should include a brief description of the tasks or problems solved each time you worked on the project.
6. In our testing, we will copy the three .c files into the ffmpeg project in the locations you specified, then we will rebuild the project and run our tests. Don't do anything that will break the tests. (For example, don't create new .h or .c files or rename files.)

Modifying and using ffmpeg in an application

Overview

Media processing encompasses many software engineering topics. Video and audio processing is complex and computationally expensive and requires efficient algorithms and implementations. Image and audio storage requires thoughtful and unambiguous designs. Applications tend to be large, intricate compositions of codecs, libraries, and implementation code. In short, media processing is an excellent platform for studying topics in this course.

In this project you will use and modify a media library called "ffmpeg". In addition, you will use the ffmpeg libraries to read and write images and/or sounds in a stand-alone application. The purpose of this project is to give you additional experience with these topics:

- Experience working within a large, unfamiliar codebase - The best way to see how to construct (or *not* how to construct) a large body of code is to work in one. In this project, you will both modify the ffmpeg libraries and use them in a separate application. To succeed, you will need to investigate both source code and documentation within ffmpeg, and you will need to learn a bit about image and/or sound representations.
  Succeeding in this project will involve a great deal of independent problem solving. You will have to learn how to navigate the ffmpeg codebase, you will need to find the correct ways to modify it, and you will have to adjust makefile(s) to compile any code you add. This will involve reading lots of code and comments, making notes, finding documentation, and browsing examples and discussions online. In addition, you will need to conduct experiments to test your theories, and you may even need to revert to a clean ffmpeg install if you break your existing install with your changes.
- Teamwork - This project is a team project, requiring teams of two. We will cover best practices for teams during class, and it is expected that you will apply them during this project. Because teamwork practice is so important, students who do not select their own partner will be assigned one randomly. Rules for changing partners will be discussed in class.
  I highly recommend that you select a partner whose schedule matches your own.
  Finally, team members who are retaking this class **must not** consult their old code. Delete your old ffmpeg installation! I will inject significant changes in the code requirements this semester that will invalidate old solutions.

I would like to stress two important facets of this project: First, a great deal of independent problem solving will be required. You will learn a lot about technologies that you will see for the first time in this project. We cannot (and will not) give you complete instructions for every small step of this project. We will, however, try to help you get started and help resolve any obnoxious issues. Second, the project specification may be altered if necessary to fix problems or to improve the educational value of this work. (This is my primary concern - I want everyone to benefit from this work and I will adapt as necessary.)

Overview -- FFMPEG

If you ever want to access audio or video data, ffmpeg is one of the best libraries to use in C++. You can read and write audio, images, and videos with C function calls. In previous semesters, I had students load 'picture.jpg' (using ffmpeg function calls) and repeatedly draw a ball on the image to create frames of a movie. Students wrote out these frames to 'cool' image files:

Students then used the ffmpeg executable to stitch together their frames into a movie (which could be viewed using any video player).

Unfortunately, student code for this animation are easily found on github. We'll switch to something similar with audio data.

This semester, students will create their own audio format, then modify and blend audio with video. The project will happen in three stages:

Part 1: Students will locate both audio and video codecs within ffmpeg and inject simple modifications. The goal of this step is for students to gain familiarity with ffmpeg.

Part 2: Students will create their own audio format for ffmpeg. Details will be coming soon.

Part 3: Students will write an application to load images or sound, create audio, and output sound files. Students will then blend their audio with video to create a final movie.

Note the following:

Program interfaces: No GUIs are allowed for the student portions of the project. Command line arguments will be used for both ffmpeg and your final audio application. You may also send a limited amount of clear debugging output to the console. (No long listings of hexadecimal image dumps, etc.)

Modifying ffmpeg: You are required to only make minimal changes to ffmpeg. (You will be adding support for a new audio file type.) Do not remove existing ffmpeg behavior. Your goal should be to modify as few existing files as possible.

It is expected that you will need to modify makefiles to include your new .c files in the build. This is OK, but again, try to minimize your changes.

Build instructions: For part #3, you will be required to supply a makefile for your application. We will give you further details in the final phase of the project

In addition, we must be able to test all your work in a simple manner. Building your changes will take us quite a while, and we cannot make specific configuration changes for every student. Your code and changes must work with the standard configuration specified below. (No custom environment variables, absolute file locations, external libraries, etc.)

Commenting / Documentation: Commenting all code is required. You will include documentation in your written reports (if assigned).

Additional clarifications and requirements will be posted as the project moves along. Please keep in mind that the intent of this project is for you to work in a large code base, and significant independent learning will be required. (Procrastination will not be indulged.)

Installing ffmpeg

Installing ffmpeg is straightforward. Follow these instructions in your CADE lab account. (Students who want to do this on their own computer or in a different environment **must** verify their code works with a standard CADE lab build of ffmpeg.) These instructions were tested on February 4, 2020.

1. Get ffmpeg from the Git repository. Change to your home directory, then use this Git command:

```
git  clone  git://source.ffmpeg.org/ffmpeg.git  ffmpeg
```
A copy of the project will be placed inside a directory named "ffmpeg".

There are two 'git' executables in the lab. /usr/local/bin/git had problems last year, but seems to be fine now, and /usr/bin/git (untested, but should still work just fine). If needed, type 'which git' to see which one you're using.

You'll need a fair bit of storage for this project. Use 'quota' to see your lab storage quota.

2. Change the permissions on the ffmpeg directory so that it is not accessible to other students:
```
chmod 2700 ffmpeg
```

Set environment variables to point to your ffmpeg location, and to add the ffmpeg binary directory to your path. Place these lines in your .cshrc file in your home directory:

```
# For ffmpeg

setenv FFMPEG_LIB_DIR "/home/userid/ffmpeg"
setenv PATH "${PATH}:$FFMPEG_LIB_DIR/bin"
setenv PKG_CONFIG_PATH "${FFMPEG_LIB_DIR}/lib/pkgconfig"
```

3. Now, edit the first line, and replace "userid" with your CADE lab login name. The path on the first line should point to the ffmpeg directory within your home directory.
4. Either **log out and log in again**, or close your terminal window and open a new one. **Verify that the environment variables are set**:
```
    printenv  |  sort
```
Double-check the environment variables, and make sure the ffmpeg binaries directory is in your path. If this is wrong, you'll run into problems soon.
5. Change into your ffmpeg directory, and run the configure script that is supplied with ffmpeg. You will supply two additional parameters: First, the prefix argument specifies the destination for the compiled libraries.
```
    ./configure  --prefix=$FFMPEG_LIB_DIR
```
Be patient. The configure script is entirely silent for a good long while, then it prints out a lot of information about the build settings for ffmpeg. If the configure script fails, you will not be able to make the ffmpeg project. (If needed, fix any problems that are preventing the configure script from completing.)
You can verify that it worked properly by examining the output. You should have decoders for the image types we're using, as well as these programs: ffmpeg, ffplay, ffprobe. (If these programs are not listed in the program

section, your configure failed and you won't be able to run code. Make sure your environment variables are set. Some environment variables set up by students for other classes in MechE or EE are incompatible and should be removed for the semester.)

6. Build the project. Switch to the ffmpeg directory, then make it:
   ```
   make
   ```
   The build process takes several minutes. (If you get an error on fate.texi, just rebuild it again.) You will see *lots* of warnings when you build. (This annoys me, but there is nothing we can do about them.)
   If the build stops due to any obvious errors, you'll need to resolve them. (Remember, rebuild if you get an error on fate.texi.)
7. After the build, you need to 'install' the ffmpeg files. This is just the process of making the libraries and copying documentation and include files to the appropriate directories. Use this command to complete the installation:
   ```
   make install
   ```
   Verify that the ffmpeg libraries are within the ffmpeg/lib directory, and that the ffmpeg executables are within the ffmpeg/bin directory (the binaries directory). If not, you will need to check your environment variables, then try the configuration and build steps again. Also, after installation, you may need to close and reopen your shell.
8. Go get an image file from somewhere, and save it. Then, try viewing a file with ffplay:
   ```
   ffplay  mypicture.tiff
   ```
   You'll need to change the filename, of course. The ffplay program will load the image, then open a window and display it as if it were a movie. Closing the window will close the program.
   If your path is incorrect, the ffplay executable will not be found. You can try logging out and logging in again (see the previous step). Also, you can try executing the binary by specifying its exact path:
   ```
   /home/yourloginhere/ffmpeg/bin/ffplay  mypicture.tiff
   ```
   I had to resolve a few problems when I got to this step. I tried this on a machine other than a lab1 machine, and I got a 'missing library' warning. I also forgot to forward X11 when I logged in to a lab1 machine, and I got a "No available video device' error. Once I got those worked out, this worked fine.
9. If you get this far, you've finished installing ffmpeg. You can now modify its source code files to change its behavior! Each time you modify the files, you'll need to make it again, then install it again. (Don't configure it again, unless you added a codec or you like to wait.)

Create an "asif" codec

This assignment is due on March 28 (EDIT: due to the unusual circumstances, this has been extended to April 17, accepted through April 21), but I recommend that you complete it early to avoid doubling up your tasks. Due to the long time frame, no late work will be accepted. Please have a contingency plan in place to deal with backup files and internet outages as **no extensions will be given for technical issues**. Wise students will hand in their drafts as they complete major steps.

'Asif' is a file extension that stands for "audio slope information file". This audio format does not exist -- you will create it.

Checkpoint requirements

For this checkpoint, you will be creating an audio codec and audio format (as a pair). Your codec and format serve as a data translator with two capabilities: Your codec/muxer will be able to take audio data and prepare it for writing to an .asif file, and your codec/demuxer will be able to take data from an .asif file and reconstruct the audio. Implementing a data format, discovering the way to make this work within ffmpeg, and programming the solution are the tasks for this assignment. Follow these requirements:

1. Work with a partner (required). (The team programming guidelines and rules remain in place. Please review them.) We cannot help you find a partner late into the assignment - find one early!
2. Create .c and .h files that are compatible with ffmpeg that implement formatting (muxing and demuxing) and decoding and encoding of .asif files. Specifically:
    - Review this page that describes the audio format you must implement. (FFMPEG - Audio File Format)
    - Follow these instructions for adding code to ffmpeg. (FFMPEG - Registering Codecs and Formats) These instructions ensure that your changes will be compatible with our build. *It is important that you do not rename our identifiers during this step. Use the given identifiers and filenames.*
    - Implement the needed source code files (at least four of them). Do not 'hack' other parts of ffmpeg -- limit your changes to your source code files. (You may, of course, make the changes specified in the instructions.) Your solution should work similarly to other ffmpeg codec and formats.

    - Your code must be your own. *You can use ffmpeg code from other codecs as a guide* and you should cite borrowed code. You cannot use major portions of someone else's codec design as your own.
    - Ensure that your source code is documented. Even though ffmpeg is not well documented, you should still follow the documentation requirements for assignments in this class.
    - Test your changes by running ffmpeg or ffplay to create or listen to files written in your .asif format.
3. Ensure that you can do the following with your codec: Use ffmpeg to convert any music file to the .asif format. Use ffmpeg to convert a .asif audio file to other audio types. Use ffplay to listen to .asif audio files.
4. Submit your work. Submit as a group through Gradescope.
    - Submit a .zip of your source code files (.c and .h only) and makefiles for your .asif codec and format. See below.
    - A short readme.txt file that lists both partner names (just in case).

To create the .zip, just add the seven files to the zip (using relative pathing exactly as shown below):

```
cd ffmpeg

zip solution.zip libavcodec/Makefile

zip solution.zip libavcodec/asifenc.c

zip solution.zip libavcodec/asifdec.c

zip solution.zip libavformat/Makefile

zip solution.zip libavformat/asifmux.c

zip solution.zip libavformat/asifdemux.c


# Follow this pattern to add any other files you created.  Make sure to be in your ffmpeg directory at

# all times when adding to the .zip, because that's where we'll unzip your solution and the filenames

# need to be prefixed with the relative directory name.
```

You can list the contents of a .zip easily. Note the -l (lowercase ell):

```
unzip -l solution.zip
```

```
Archive:  solution.zip
 Length      Date    Time    Name
---------  ---------- -----   ----
   73418  04-02-2020 18:49   libavcodec/Makefile
   56211  04-02-2020 19:29   libavcodec/asifdec.c
    3708  04-14-2020 22:33   libavcodec/asifenc.c
   38774  04-02-2020 19:06   libavformat/Makefile
    4815  04-02-2020 19:43   libavformat/asifdemux.c
    1392  04-02-2020 19:40   libavformat/asifmux.c
---------                     -------
  178318                     6 files
```

(It is important that each file contains the relative directory name. Note that my file sizes and dates will be very different from yours.)

Hand in "solution.zip" along with your "readme.txt", and if submitting as partners, make sure to indicate your partner in Gradescope.

We will test your code in our ffmpeg test directory. For functionality testing, we will use a CADE lab1 machine and we will:

- Complete the setup instructions,
- unzip your source code files into ffmpeg using these commands:
    - cd ffmpeg
    - unzip solution.zip
- reconfigure the project,
- rebuild the project, and
- run tests to make sure your file format works.

Much of this will occur within automated scripts. We will not correct your errors to make the build work. If your code fails to build or run, we cannot give you credit for functionality (no matter how much time you spent on it).

A small amount of credit (15%) will be reserved for code style and documentation.

Hints, tips, and information

If you have not yet reviewed these pages, do so now:

- FFMPEG - Audio File Format
- FFMPEG - Registering Codecs and Formats

While you cannot submit another codec or formatter as your own, that doesn't mean you cannot copy one into your own files for initial tests.

This is what I do every year -- I start with an existing codec/format, copy it into my files, and change the identifiers in the structure at the ends to my identifiers. Once this compiles and runs, I am free to modify/replace it with my own code. Usually, I end up stripping out tons of stuff leaving only a few dozen lines of code that allocate memory and copy data around.

Each year I have to fix up these instructions. I've made the fixes for 2020 and I will retest it at the start of spring break. Because of the change from video to audio, I expect a few corrections will be needed. Don't let this stop you -- a lot of progress can be made by reviewing existing formats and codecs to see how they work.

In a previous year, I had trouble running my compiled version of ffmpeg. The problem is that there was an incomplete build of ffmpeg installed in lab1 in /usr/local/bin. (Type 'which ffmpeg'. If this maps to /usr/local/bin, you're not running your compiled version of ffmpeg.)

To solve this, I made sure ffmpeg was the first entry in my path. In my .cshrc file, I modified my path as follows:

```
setenv PATH "$FFMPEG_LIB_DIR/bin:${PATH}"
```

When my path was set correctly, typing 'which ffmpeg' reported that my compiled version of ffmpeg would run (instead of /usr/local/bin/ffmpeg).

Additionally, don't forget to do 'make install'. :(

Many students post some tutorial help or questions in the discussion forum. If you have not checked it out yet, I recommend that you take a look at it. Just don't post code, step-by-step instructions, or other solutions.

A previous student found a link to ffmpeg's doc: http://www.ffmpeg.org/doxygen/trunk/

Quote: "It has hyperlinks for quick navigation. It also has a handful of great, somewhat-documented examples, accessible via the tab link at the top. One of them is a good encode-decode example, showing how frames, packets, and codec contexts are used. You can click on the hyperlinks and get right to the API for a struct/enum/def etc!"

Every codec has a structure that defines capabilities, data storage, and entry points for the codec. Here is the structure from a video codec, pngdec.c:

```
AVCodec ff_png_decoder = {
    .name           = "png",
    .long_name      = NULL_IF_CONFIG_SMALL("PNG (Portable
Network Graphics) image"),
    .type           = AVMEDIA_TYPE_VIDEO,
    .id             = AV_CODEC_ID_PNG,
    .priv_data_size = sizeof(PNGDecContext),
    .init           = png_dec_init,
    .close          = png_dec_end,
    .decode         = decode_frame,
    .init_thread_copy = ONLY_IF_THREADS_ENABLED(png_dec_init),
    .update_thread_context =
ONLY_IF_THREADS_ENABLED(update_thread_context),
    .capabilities   = CODEC_CAP_DR1 | CODEC_CAP_FRAME_THREADS
/*| CODEC_CAP_DRAW_HORIZ_BAND*/,
};
```

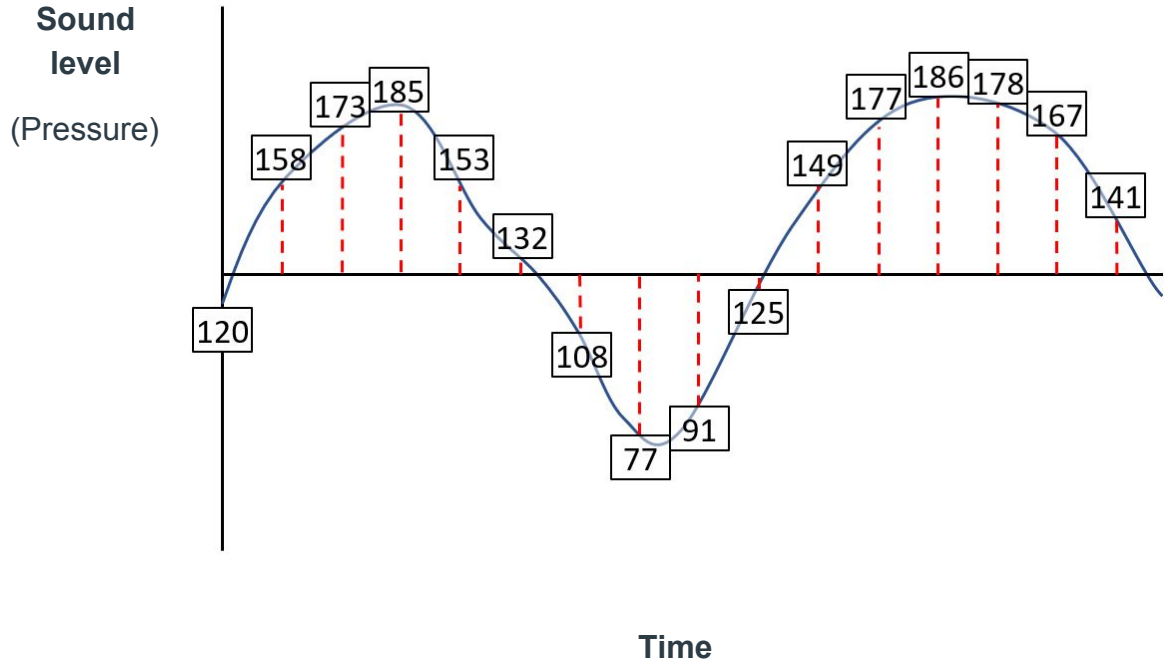
Audio codecs have similar structure. The structure name (identifier) must match the name used when the codec is registered. Our AVCodec structures will have the names ff_asif_encoder and ff_asif_decoder. These structures define entry points in the code - function names are specified to indicate which function initializes the codec, etc. Also note that the .id field should match the ID we defined earlier.

Formats also have these structures. Again, the fields in the struct capture function pointers, then ffmpeg calls our functions through the pointers stored in each struct.

FFMPEG - Audio File Format

Brief Audio Tutorial

Sound is rapid, small pressure changes in some medium (vibrating air, vibrating guitar strings, etc.). To capture sound (audio) information in a file, we simply need to record these pressure changes (vibrations) over time. A microphone provides the needed data signal, and the simplest model draws the sound data as a wave:



In the diagram above:

- The sound wave is the blue line that rises and falls over time.
- The red dashed lines mark moments in time when we'd like to gather a data *sample*.
- The numbers represent these data samples -- the strength (amplitude) of the audio signal at some moment in time.
- These data samples are in the range [0..255]. They are 8-bit unsigned data samples. Audio files are commonly 8-bit unsigned samples (128 centerline), or 16-bit signed samples (0 centerline).
- Data samples must be taken at regular intervals. One common data sampling rate is 44,100 samples per second.
- For stereo music, there are two waveforms that must be sampled and captured.

To capture a sound wave, we simply record all the data samples:

120	158	173	185	153	...etc...

To play a sound, these data samples are used to reconstruct the sound wave over time.

For stereo data (left and right *channels*), samples are taken from both waveforms and interleaved. There are twice as many samples for two channel audio:

Left sample 0	Right sample 0	Left sample 1	Right sample 1	Left sample 2	...etc...

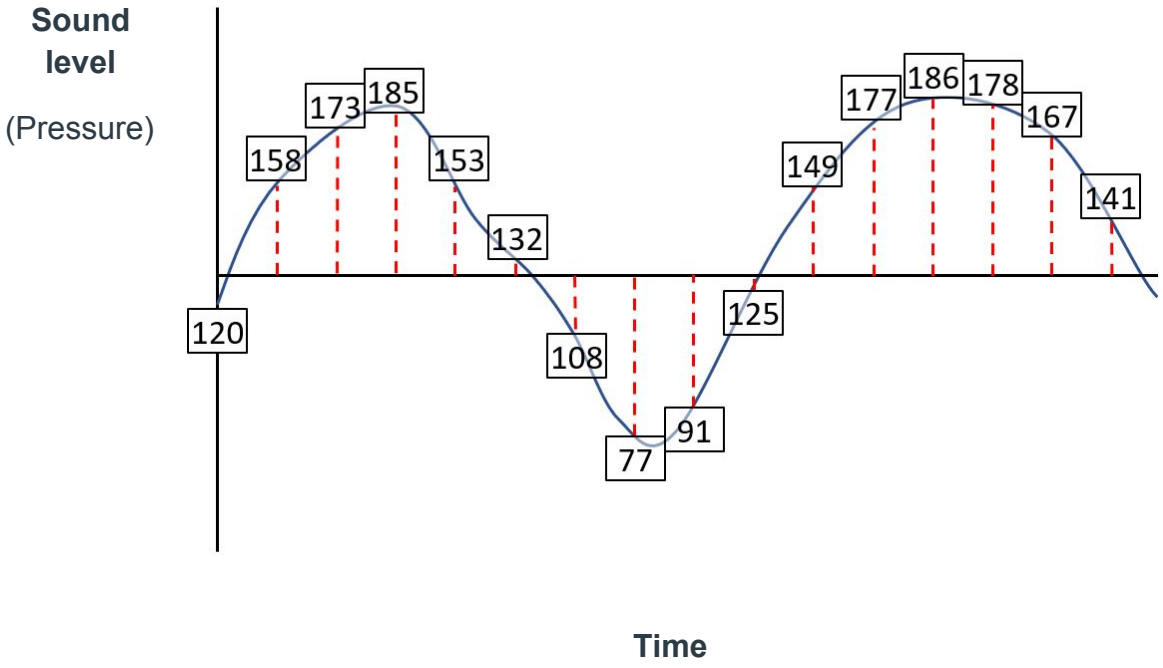
If there are multiple channels, the first sample from each channel is listed first, then the second sample for each channel is listed, and so on. This is called *interleaved* audio data. Note that in the format below I ask you to *not* interleave the data.

Our Audio File Format:

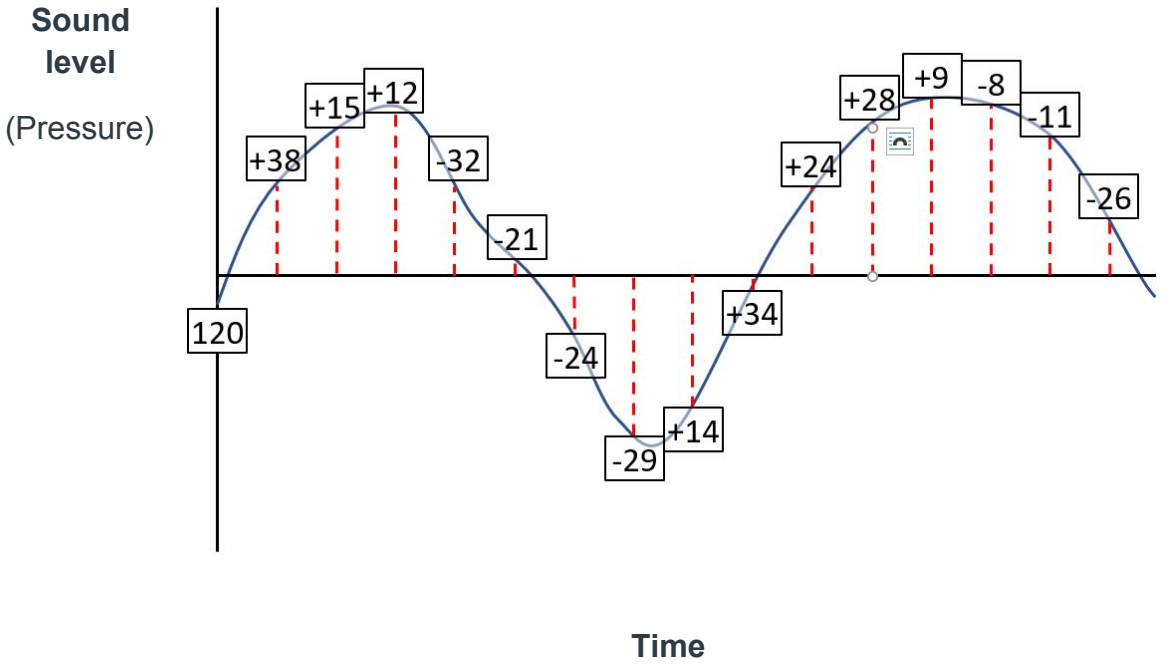
For ffmpeg part #2, students will be encoding and decoding audio data:

- During encoding, your format and codec will receive audio information from ffmpeg and produce a data packet that will be streamed or written into a file.
- During decoding, your format and codec will receive a data stream (from a file), and you will extract a data packet and decode it into audio data in a format that ffmpeg can use.
- Your encoder will organize the audio data into our "asif" format. Your decoder will extract audio data from the same "asif" format.

We will make a small change to the audio data to facilitate simple data compression techniques. Again consider the initial waveform:



The samples broadly range from [0..255] (an 8-bit unsigned integer). By taking the first derivative of the data, we can cluster the data samples nearer to 0 (which will help some simple compression techniques):



The first derivative just computes the difference in position between samples. The audio information is transformed into a sample, followed by deltas (change in the signal strength). It's the same data, just expressed in a different form. Given the original data:

120	158	173	185	153	...etc...

Record the first sample (unsigned), but then record the changes needed to calculate the additional samples (8-bit signed values):

120	+38	+15	+12	-32	...etc...

Our "asif" file format will store samples using this 'delta' format. Note that the "asif" delta format cannot accurately capture waves with deltas greater than 127 or less than -128. If a delta is too great (or too small), clamp it to be within [-128..127], and then catch up in the next delta sample. Never 'wrap around' the number wheel (overflow) when processing audio samples.

The "asif" file format is required to have the following data structure:

Byte Offset	**Field**	**Meaning**
+0	Four characters "asif"	To identify the data as "asif" data
+4	32-bit little endian int	Sample rate (frequency)
+8	16-bit little endian int	Number of channels
+10	32-bit little endian int	Number of samples per channel (n)
+14	8-bit unsigned data samples for channel 0	sample and (n-1) deltas (in order) for channel 0.

+14+n	8-bit unsigned data samples for channel 1	sample and (n-1) deltas (in order) for channel 1. (optional, may not be present in single channel data)
...etc...	...etc...	Samples for additional channels may be present

It is expected that all students will encode identical audio data into identical files (using the above format).