I was looking for a tutorial/book that would teach me how to start to use FFmpeg as a library (a.k.a. libav) and then I found the "How to write a video player in less than 1k lines" tutorial. Unfortunately it was deprecated, so I decided to write this one.

Most of the code in here will be in c **but don't worry**: you can easily understand and apply it to your preferred language. FFmpeg libav has lots of bindings for many languages like python, go and even if your language doesn't have it, you can still support it through the `ffi` (here's an example with Lua).

We'll start with a quick lesson about what is video, audio, codec and container and then we'll go to a crash course on how to use `FFmpeg` command line and finally we'll write code, feel free to skip directly to the section Learn FFmpeg libav the Hard Way.

Some people used to say that the Internet video streaming is the future of the traditional TV, in any case, the FFmpeg is something that is worth studying.

Table of Contents

Intro

video - what you see!

If you have a sequence series of images and change them at a given frequency (let's say 24 images per second), you will create an illusion of movement. In summary this is the very basic idea behind a video: **a series of pictures / frames running at a given rate**.

undefined

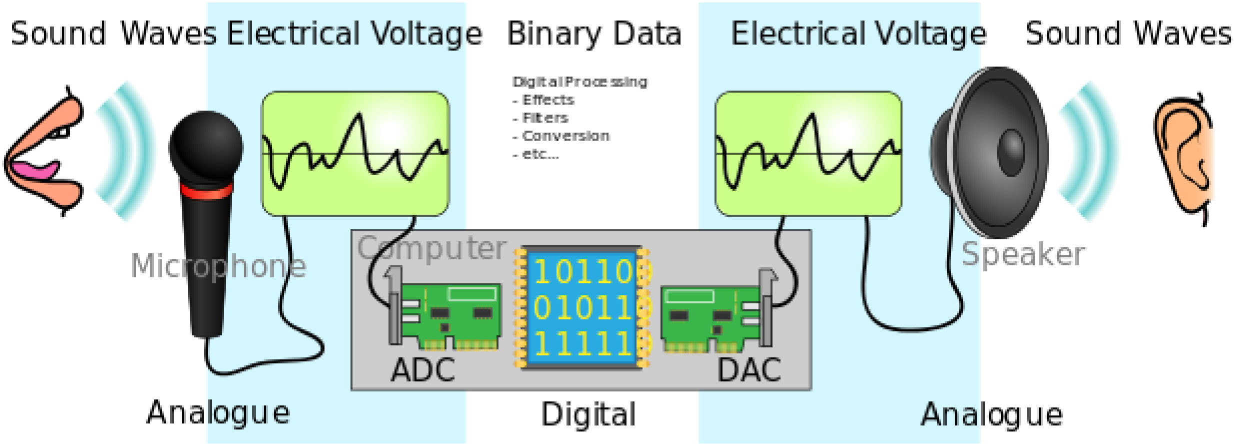
Zeitgenössische Illustration (1886)

audio - what you listen!

Although a muted video can express a variety of feelings, adding sound to it brings more pleasure to the experience.

Sound is the vibration that propagates as a wave of pressure, through the air or any other transmission medium, such as a gas, liquid or solid.

> In a digital audio system, a microphone converts sound to an analog electrical signal, then an analog-to-digital converter (ADC)—typically using pulse-code modulation—converts (PCM) the analog signal into a digital signal.



> Source

codec - shrinking data

> CODEC is an electronic circuit or software that **compresses or decompresses digital audio/video.** It converts raw (uncompressed) digital audio/video to a compressed format or vice versa. https://en.wikipedia.org/wiki/Video_codec

But if we chose to pack millions of images in a single file and called it a movie, we might end up with a huge file. Let's do the math:

Suppose we are creating a video with a resolution of `1080 x 1920` (height x width) and that we'll spend `3 bytes` per pixel (the minimal point at a screen) to encode the color (or 24 bit color, what gives us 16,777,216 different colors) and this video runs at `24 frames per second` and it is `30 minutes` long.

```
toppf = 1080 * 1920 //total_of_pixels_per_frame
cpp = 3 //cost_per_pixel
tis = 30 * 60 //time_in_seconds
fps = 24 //frames_per_second

required_storage = tis * fps * toppf * cpp
```

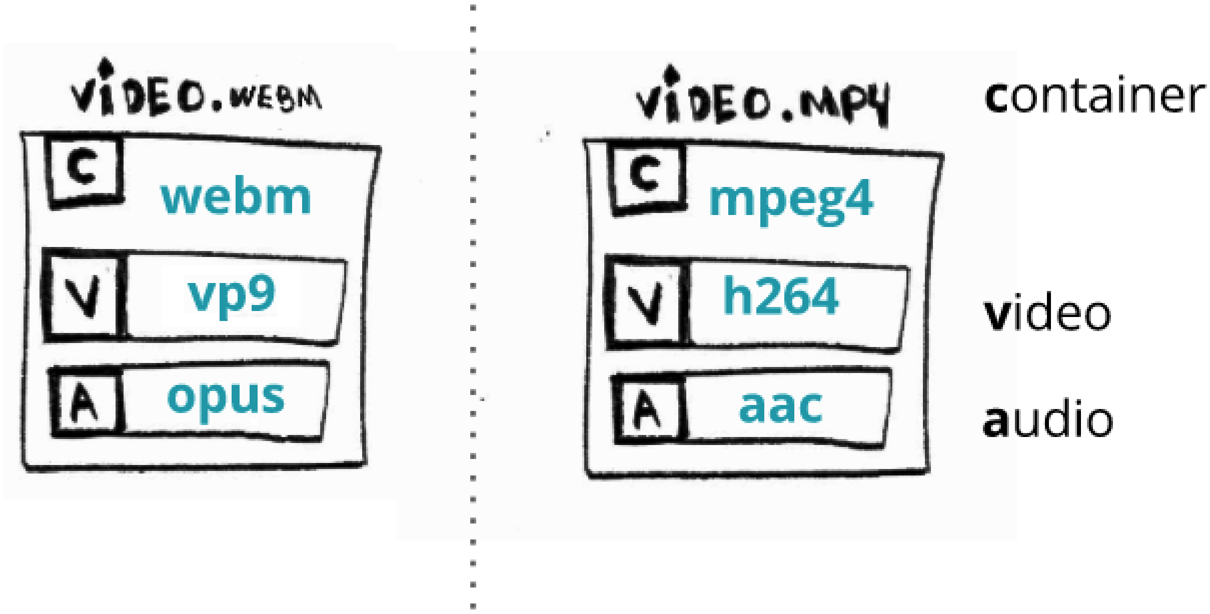
This video would require approximately `250.28GB` of storage or `1.11Gbps` of bandwidth! That's why we need to use a [CODEC](#).

container - a comfy place for audio and video

> A container or wrapper format is a metafile format whose specification describes how different elements of data and metadata coexist in a computer file. https://en.wikipedia.org/wiki/Digital_container_format

A **single file that contains all the streams** (mostly the audio and video) and it also provides **synchronization and general metadata**, such as title, resolution and etc.

Usually we can infer the format of a file by looking at its extension: for instance a `video.webm` is probably a video using the container `webm`.



FFmpeg - command line

> A complete, cross-platform solution to record, convert and stream audio and video.

To work with multimedia we can use the AMAZING tool/library called [FFmpeg](#). Chances are you already know/use it directly or indirectly (do you use [Chrome?](#)).

It has a command line program called `ffmpeg`, a very simple yet powerful binary. For instance, you can convert from `mp4` to the container `avi` just by typing the follow command:

```
$ ffmpeg -i input.mp4 output.avi
```

We just made a **remuxing** here, which is converting from one container to another one. Technically FFmpeg could also be doing a transcoding but we'll talk about that later.

FFmpeg command line tool 101

FFmpeg does have a documentation that does a great job of explaining how it works.

To make things short, the FFmpeg command line program expects the following argument format to perform its actions `ffmpeg {1} {2} -i {3} {4} {5}`, where:

1. global options
2. input file options
3. input url
4. output file options
5. output url

The parts 2, 3, 4 and 5 can be as many as you need. It's easier to understand this argument format in action:

```
# WARNING: this file is around 300MB
$ wget -O bunny_1080p_60fps.mp4
http://distribution.bbb3d.renderfarming.net/video/mp4/bbb_sunflower_1080p_60fps_normal.mp4

$ ffmpeg \
-y \ # global options
-c:a libfdk_aac -c:v libx264 \ # input options
-i bunny_1080p_60fps.mp4 \ # input url
-c:v libvpx-vp9 -c:a libvorbis \ # output options
bunny_1080p_60fps_vp9.webm # output url
```

This command takes an input file `mp4` containing two streams (an audio encoded with `aac` CODEC and a video encoded using `h264` CODEC) and convert it to `webm`, changing its audio and video CODECs too.

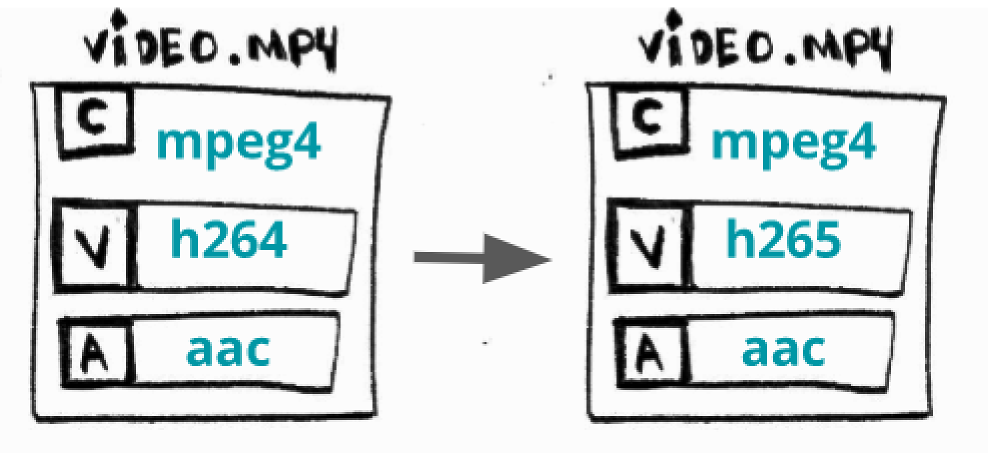
We could simplify the command above but then be aware that FFmpeg will adopt or guess the default values for you. For instance when you just type `ffmpeg -i input.avi output.mp4` what audio/video CODEC does it use to produce the `output.mp4`?

Werner Robitza wrote a must read/execute tutorial about encoding and editing with FFmpeg.

Common video operations

While working with audio/video we usually do a set of tasks with the media.

Transcoding



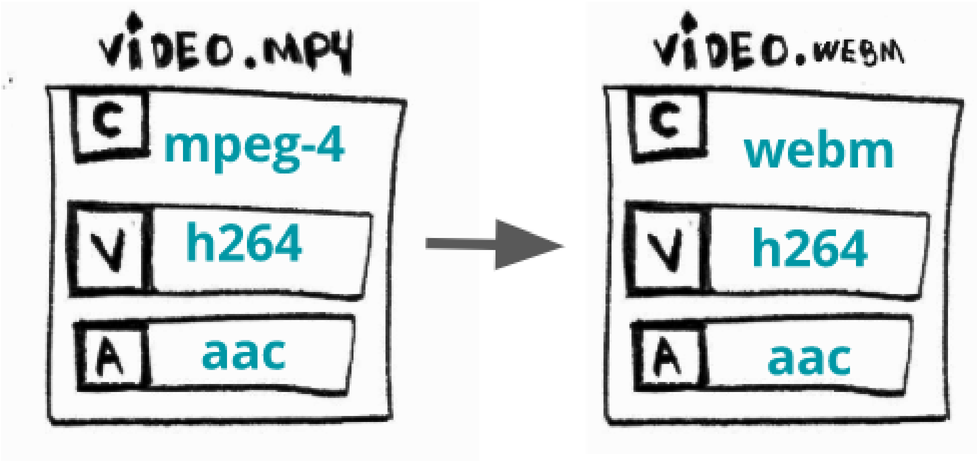
What? the act of converting one of the streams (audio or video) from one CODEC to another one.

Why? sometimes some devices (TVs, smartphones, console and etc) doesn't support X but Y and newer CODECs provide better compression rate.

How? converting an `H264` (AVC) video to an `H265` (HEVC).

```
$ ffmpeg \
-i bunny_1080p_60fps.mp4 \
-c:v libx265 \
bunny_1080p_60fps_h265.mp4
```

Transmuxing



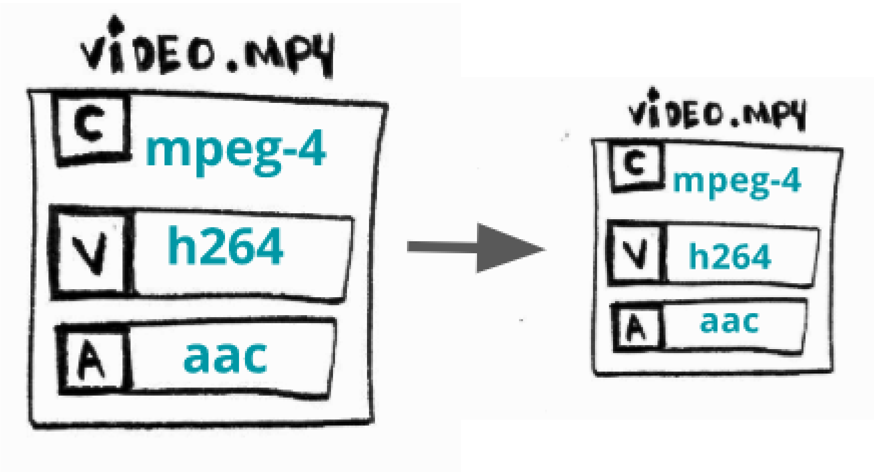
What? the act of converting from one format (container) to another one.

Why? sometimes some devices (TVs, smartphones, console and etc) doesn't support X but Y and sometimes newer containers provide modern required features.

How? converting a `mp4` to a `webm`.

```
$ ffmpeg \
-i bunny_1080p_60fps.mp4 \
-c copy \ # just saying to ffmpeg to skip encoding
bunny_1080p_60fps.webm
```

Transrating



What? the act of changing the bit rate, or producing other renditions.

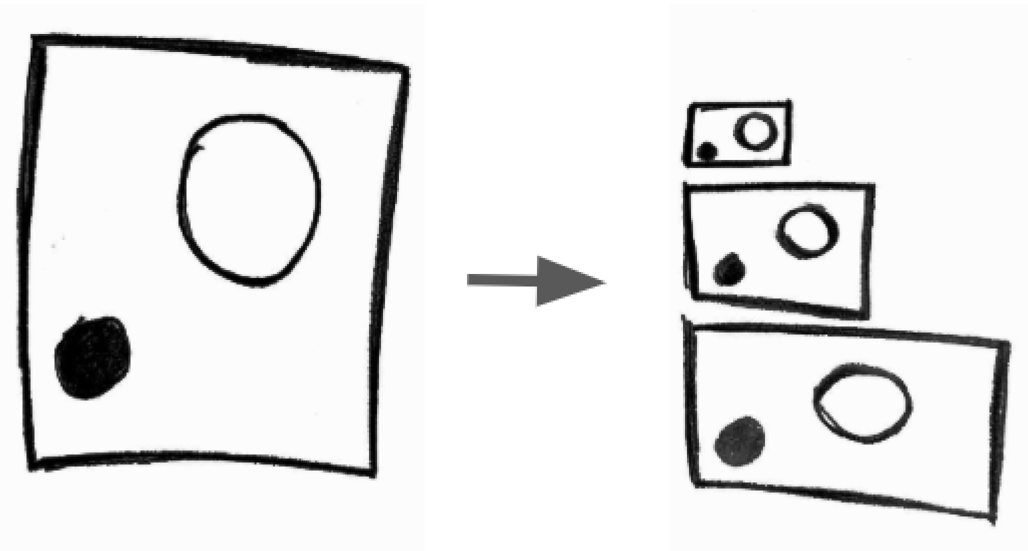
Why? people will try to watch your video in a `2G` (edge) connection using a less powerful smartphone or in a `fiber` Internet connection on their 4K TVs therefore you should offer more than on rendition of the same video with different bit rate.

How? producing a rendition with bit rate between 3856K and 2000K.

```
$ ffmpeg \
-i bunny_1080p_60fps.mp4 \
-minrate 964K -maxrate 3856K -bufsize 2000K \
bunny_1080p_60fps_transrating_964_3856.mp4
```

Usually we'll be using transrating with transsizing. Werner Robitza wrote another must read/execute series of posts about FFmpeg rate control.

Transsizing



What? the act of converting from one resolution to another one. As said before transsizing is often used with transrating.

Why? reasons are about the same as for the transrating.

How? converting a `1080p` to a `480p` resolution.

```
$ ffmpeg \
-i bunny_1080p_60fps.mp4 \
-vf scale=480:-1 \
bunny_1080p_60fps_transsizing_480.mp4
```

Bonus Round: Adaptive Streaming

What? the act of producing many resolutions (bit rates) and split the media into chunks and serve them via http.

Why? to provide a flexible media that can be watched on a low end smartphone or on a 4K TV, it's also easy to scale and deploy but it can add latency.

How? creating an adaptive WebM using DASH.

```
# video streams
$ ffmpeg -i bunny_1080p_60fps.mp4 -c:v libvpx-vp9 -s 160x90 -b:v 250k -keyint_min 150 -g
150 -an -f webm -dash 1 video_160x90_250k.webm

$ ffmpeg -i bunny_1080p_60fps.mp4 -c:v libvpx-vp9 -s 320x180 -b:v 500k -keyint_min 150 -g
150 -an -f webm -dash 1 video_320x180_500k.webm

$ ffmpeg -i bunny_1080p_60fps.mp4 -c:v libvpx-vp9 -s 640x360 -b:v 750k -keyint_min 150 -g
150 -an -f webm -dash 1 video_640x360_750k.webm

$ ffmpeg -i bunny_1080p_60fps.mp4 -c:v libvpx-vp9 -s 640x360 -b:v 1000k -keyint_min 150 -g
150 -an -f webm -dash 1 video_640x360_1000k.webm

$ ffmpeg -i bunny_1080p_60fps.mp4 -c:v libvpx-vp9 -s 1280x720 -b:v 1500k -keyint_min 150 -g
150 -an -f webm -dash 1 video_1280x720_1500k.webm

# audio streams
$ ffmpeg -i bunny_1080p_60fps.mp4 -c:a libvorbis -b:a 128k -vn -f webm -dash 1
audio_128k.webm

# the DASH manifest
$ ffmpeg \
 -f webm_dash_manifest -i video_160x90_250k.webm \
 -f webm_dash_manifest -i video_320x180_500k.webm \
 -f webm_dash_manifest -i video_640x360_750k.webm \
 -f webm_dash_manifest -i video_640x360_1000k.webm \
 -f webm_dash_manifest -i video_1280x720_500k.webm \
```

```
  -f webm_dash_manifest -i audio_128k.webm \
  -c copy -map 0 -map 1 -map 2 -map 3 -map 4 -map 5 \
  -f webm_dash_manifest \
  -adaptation_sets "id=0,streams=0,1,2,3,4 id=1,streams=5" \
  manifest.mpd
```

PS: I stole this example from the Instructions to playback Adaptive WebM using DASH

Going beyond

There are many and many other usages for FFmpeg. I use it in conjunction with *iMovie* to produce/edit some videos for YouTube and you can certainly use it professionally.

Learn FFmpeg libav the Hard Way

> Don't you wonder sometimes 'bout sound and vision? **David Robert Jones**

Since the FFmpeg is so useful as a command line tool to do essential tasks over the media files, how can we use it in our programs?

FFmpeg is composed by several libraries that can be integrated into our own programs. Usually, when you install FFmpeg, it installs automatically all these libraries. I'll be referring to the set of these libraries as **FFmpeg libav**.

> This title is a homage to Zed Shaw's series Learn X the Hard Way, particularly his book Learn C the Hard Way.

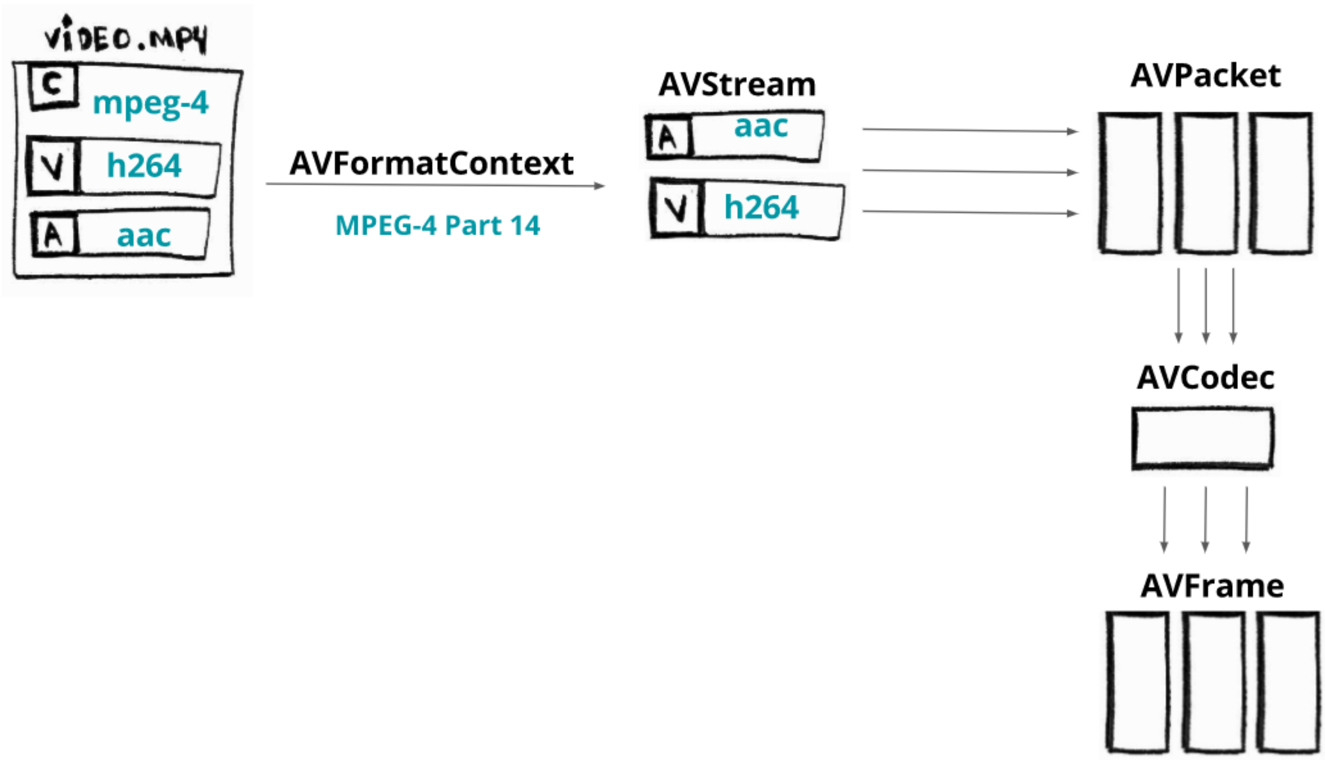
Chapter 0 - The infamous hello world

This hello world actually won't show the message `"hello world"` in the terminal 👅 Instead we're going to **print out information about the video**, things like its format (container), duration, resolution, audio channels and, in the end, we'll **decode some frames and save them as image files**.

FFmpeg libav architecture

But before we start to code, let's learn how **FFmpeg libav architecture** works and how its components communicate with others.

Here's a diagram of the process of decoding a video:



You'll first need to load your media file into a component called [`AVFormatContext`] (the video container is also known as format). It actually doesn't fully load the whole file: it often only reads the header.

Once we loaded the minimal **header of our container**, we can access its streams (think of them as a rudimentary audio and video data). Each stream will be available in a component called [`AVStream`].

> Stream is a fancy name for a continuous flow of data.

Suppose our video has two streams: an audio encoded with AAC CODEC and a video encoded with H264 (AVC) CODEC. From each stream we can extract **pieces (slices) of data** called packets that will be loaded into components named `AVPacket`.

The **data inside the packets are still coded** (compressed) and in order to decode the packets, we need to pass them to a specific `AVCodec`.

The `AVCodec` will decode them into `AVFrame` and finally, this component gives us **the uncompressed frame**. Noticed that the same terminology/process is used either by audio and video stream.

Requirements

Since some people were facing issues while compiling or running the examples **we're going to use `Docker` as our development/runner environment,** we'll also use the big buck bunny video so if you don't have it locally just run the command `make fetch_small_bunny_video`.

Chapter 0 - code walkthrough

> ### TLDR; show me the code and execution.
>
> ```
> $ make run_hello
> ```

We'll skip some details, but don't worry: the source code is available at github.

We're going to allocate memory to the component `AVFormatContext` that will hold information about the format (container).

```
AVFormatContext *pFormatContext = avformat_alloc_context();
```

Now we're going to open the file and read its header and fill the `AVFormatContext` with minimal information about the format (notice that usually the codecs are not opened). The function used to do this is `avformat_open_input`. It expects an `AVFormatContext`, a `filename` and two optional arguments: the `AVInputFormat` (if you pass `NULL`, FFmpeg will guess the format) and the `AVDictionary` (which are the options to the demuxer).

```
avformat_open_input(&pFormatContext, filename, NULL, NULL);
```

We can print the format name and the media duration:

```
printf("Format %s, duration %lld us", pFormatContext->iformat->long_name, pFormatContext->duration);
```

To access the `streams`, we need to read data from the media. The function `avformat_find_stream_info` does that. Now, the `pFormatContext->nb_streams` will hold the amount of streams and the `pFormatContext->streams[i]` will give us the `i` stream (an `AVStream`).

```
avformat_find_stream_info(pFormatContext,  NULL);
```

Now we'll loop through all the streams.

```
for (int i = 0; i < pFormatContext->nb_streams; i++)
{
  //
}
```

For each stream, we're going to keep the `AVCodecParameters`, which describes the properties of a codec used by the stream `i`.

```
AVCodecParameters *pLocalCodecParameters = pFormatContext->streams[i]->codecpar;
```

With the codec properties we can look up the proper CODEC querying the function `avcodec_find_decoder` and find the registered decoder for the codec id and return an `AVCodec`, the component that knows how to en**CO**de and **DEC**ode the stream.

```
AVCodec *pLocalCodec = avcodec_find_decoder(pLocalCodecParameters->codec_id);
```

Now we can print information about the codecs.

```
// specific for video and audio
if (pLocalCodecParameters->codec_type == AVMEDIA_TYPE_VIDEO) {
  printf("Video Codec: resolution %d x %d", pLocalCodecParameters->width,
pLocalCodecParameters->height);
} else if (pLocalCodecParameters->codec_type == AVMEDIA_TYPE_AUDIO) {
  printf("Audio Codec: %d channels, sample rate %d", pLocalCodecParameters->channels,
pLocalCodecParameters->sample_rate);
}
// general
printf("\tCodec %s ID %d bit_rate %lld", pLocalCodec->long_name, pLocalCodec->id,
pCodecParameters->bit_rate);
```

With the codec, we can allocate memory for the `AVCodecContext`, which will hold the context for our decode/encode process, but then we need to fill this codec context with CODEC parameters; we do that with `avcodec_parameters_to_context`.

Once we filled the codec context, we need to open the codec. We call the function `avcodec_open2` and then we can use it.

```
AVCodecContext *pCodecContext = avcodec_alloc_context3(pCodec);
avcodec_parameters_to_context(pCodecContext, pCodecParameters);
avcodec_open2(pCodecContext, pCodec, NULL);
```

Now we're going to read the packets from the stream and decode them into frames but first, we need to allocate memory for both components, the `AVPacket` and `AVFrame`.

```
AVPacket *pPacket = av_packet_alloc();
AVFrame *pFrame = av_frame_alloc();
```

Let's feed our packets from the streams with the function `av_read_frame` while it has packets.

```
while (av_read_frame(pFormatContext, pPacket) >= 0) {
  //...
}
```

Let's **send the raw data packet** (compressed frame) to the decoder, through the codec context, using the function `avcodec_send_packet`.

```
avcodec_send_packet(pCodecContext, pPacket);
```

And let's **receive the raw data frame** (uncompressed frame) from the decoder, through the same codec context, using the function `avcodec_receive_frame`.

```
avcodec_receive_frame(pCodecContext, pFrame);
```

We can print the frame number, the PTS, DTS, frame type and etc.

```
printf(
    "Frame %c (%d) pts %d dts %d key_frame %d [coded_picture_number %d,
display_picture_number %d]",
    av_get_picture_type_char(pFrame->pict_type),
    pCodecContext->frame_number,
    pFrame->pts,
    pFrame->pkt_dts,
    pFrame->key_frame,
    pFrame->coded_picture_number,
    pFrame->display_picture_number
);
```

Finally we can save our decoded frame into a simple gray image. The process is very simple, we'll use the `pFrame->data` where the index is related to the planes Y, Cb and Cr, we just picked `0` (Y) to save our gray image.

```
save_gray_frame(pFrame->data[0], pFrame->linesize[0], pFrame->width, pFrame->height,
frame_filename);

static void save_gray_frame(unsigned char *buf, int wrap, int xsize, int ysize, char
*filename)
{
    FILE *f;
    int i;
    f = fopen(filename,"w");
    // writing the minimal required header for a pgm file format
    // portable graymap format -> https://en.wikipedia.org/wiki/Netpbm_format#PGM_example
    fprintf(f, "P5\n%d %d\n%d\n", xsize, ysize, 255);

    // writing line by line
    for (i = 0; i < ysize; i++)
        fwrite(buf + i * wrap, 1, xsize, f);
    fclose(f);
}
```

And voilà! Now we have a gray scale image with 2MB:

Chapter 1 - syncing audio and video

> **Be the player** - a young JS developer writing a new MSE video player.

Before we move to code a transcoding example let's talk about **timing**, or how a video player knows the right time to play a frame.

In the last example, we saved some frames that can be seen here:

When we're designing a video player we need to **play each frame at a given pace**, otherwise it would be hard to pleasantly see the video either because it's playing so fast or so slow.

Therefore we need to introduce some logic to play each frame smoothly. For that matter, each frame has a **presentation timestamp** (PTS) which is an increasing number factored in a **timebase** that is a rational number (where the denominator is known as **timescale**) divisible by the **frame rate (fps)**.

It's easier to understand when we look at some examples, let's simulate some scenarios.

For a `fps=60/1` and `timebase=1/60000` each PTS will increase `timescale / fps = 1000` therefore the **PTS real time** for each frame could be (supposing it started at 0):

- `frame=0, PTS = 0, PTS_TIME = 0`
- `frame=1, PTS = 1000, PTS_TIME = PTS * timebase = 0.016`
- `frame=2, PTS = 2000, PTS_TIME = PTS * timebase = 0.033`

For almost the same scenario but with a timebase equal to `1/60`.

- `frame=0, PTS = 0, PTS_TIME = 0`
- `frame=1, PTS = 1, PTS_TIME = PTS * timebase = 0.016`
- `frame=2, PTS = 2, PTS_TIME = PTS * timebase = 0.033`
- `frame=3, PTS = 3, PTS_TIME = PTS * timebase = 0.050`

For a `fps=25/1` and `timebase=1/75` each PTS will increase `timescale / fps = 3` and the PTS time could be:

- `frame=0, PTS = 0, PTS_TIME = 0`
- `frame=1, PTS = 3, PTS_TIME = PTS * timebase = 0.04`
- `frame=2, PTS = 6, PTS_TIME = PTS * timebase = 0.08`
- `frame=3, PTS = 9, PTS_TIME = PTS * timebase = 0.12`
- ...
- `frame=24, PTS = 72, PTS_TIME = PTS * timebase = 0.96`
- ...
- `frame=4064, PTS = 12192, PTS_TIME = PTS * timebase = 162.56`

Now with the `pts_time` we can find a way to render this synched with audio `pts_time` or with a system clock. The FFmpeg libav provides these info through its API:

- fps = [`AVStream->avg_frame_rate`]
- tbr = [`AVStream->r_frame_rate`]
- tbn = [`AVStream->time_base`]

Just out of curiosity, the frames we saved were sent in a DTS order (frames: 1,6,4,2,3,5) but played at a PTS order (frames: 1,2,3,4,5). Also, notice how cheap are B-Frames in comparison to P or I-Frames.

```
LOG: AVStream->r_frame_rate 60/1
LOG: AVStream->time_base 1/60000
...
LOG: Frame 1 (type=I, size=153797 bytes) pts 6000 key_frame 1 [DTS 0]
LOG: Frame 2 (type=B, size=8117 bytes) pts 7000 key_frame 0 [DTS 3]
LOG: Frame 3 (type=B, size=8226 bytes) pts 8000 key_frame 0 [DTS 4]
LOG: Frame 4 (type=B, size=17699 bytes) pts 9000 key_frame 0 [DTS 2]
LOG: Frame 5 (type=B, size=6253 bytes) pts 10000 key_frame 0 [DTS 5]
LOG: Frame 6 (type=P, size=34992 bytes) pts 11000 key_frame 0 [DTS 1]
```

Chapter 2 - remuxing

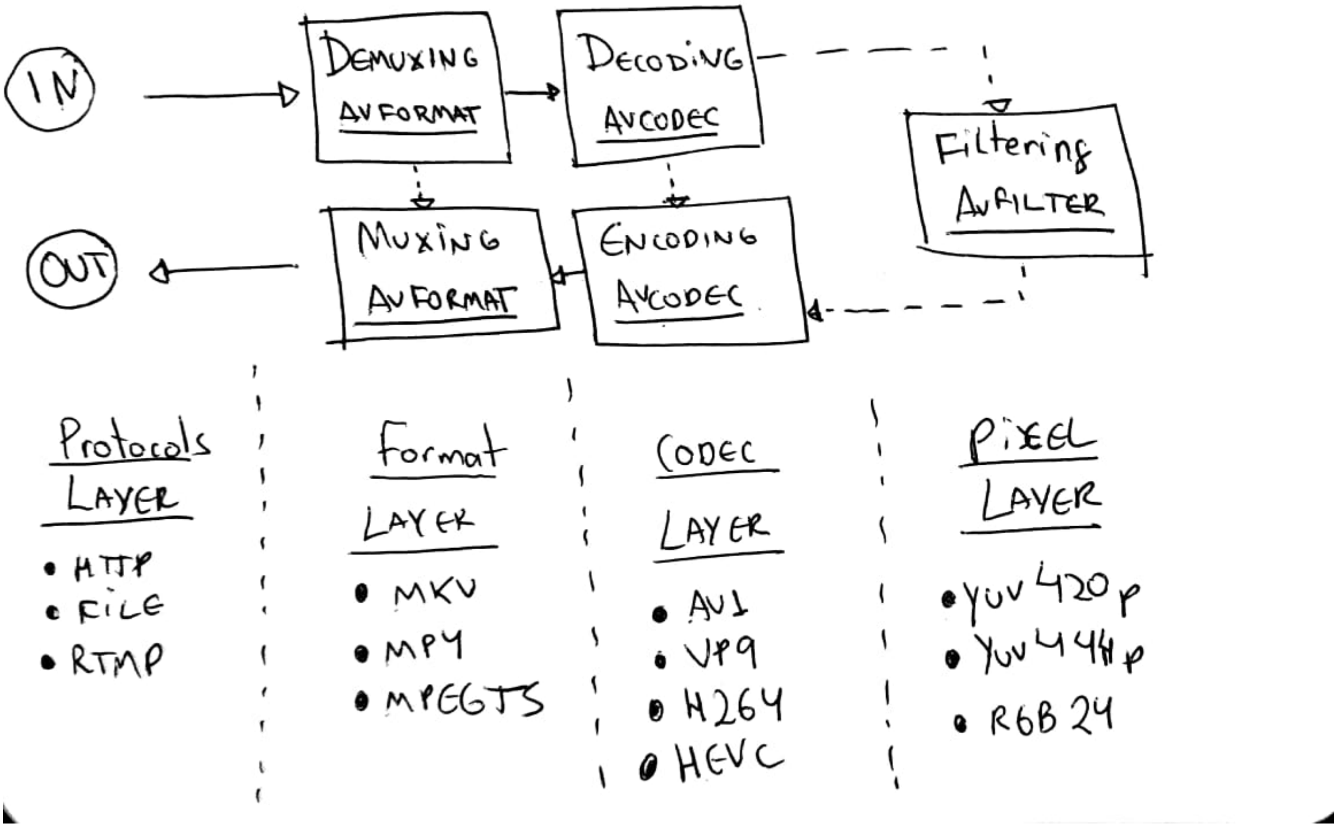
Remuxing is the act of changing from one format (container) to another, for instance, we can change a MPEG-4 video to a MPEG-TS one without much pain using FFmpeg:

```
ffmpeg input.mp4 -c copy output.ts
```

It'll demux the mp4 but it won't decode or encode it (`-c copy`) and in the end, it'll mux it into a `mpegts` file. If you don't provide the format `-f` the ffmpeg will try to guess it based on the file's extension.

The general usage of FFmpeg or the libav follows a pattern/architecture or workflow:

- **protocol layer** - it accepts an `input` (a `file` for instance but it could be a `rtmp` or `HTTP` input as well)
- **format layer** - it `demuxes` its content, revealing mostly metadata and its streams
- **codec layer** - it `decodes` its compressed streams data *optional*
- **pixel layer** - it can also apply some `filters` to the raw frames (like resizing)*optional*
- and then it does the reverse path
- **codec layer** - it `encodes` (or `re-encodes` or even `transcodes`) the raw frames*optional*
- **format layer** - it `muxes` (or `remuxes`) the raw streams (the compressed data)
- **protocol layer** - and finally the muxed data is sent to an `output` (another file or maybe a network remote server)



> This graph is strongly inspired by Leixiaohua's and Slhck's works.

Now let's code an example using libav to provide the same effect as in `ffmpeg input.mp4 -c copy output.ts`.

We're going to read from an input (`input_format_context`) and change it to another output (`output_format_context`).

```
AVFormatContext *input_format_context = NULL;
AVFormatContext *output_format_context = NULL;
```

We start doing the usually allocate memory and open the input format. For this specific case, we're going to open an input file and allocate memory for an output file.

```
if ((ret = avformat_open_input(&input_format_context, in_filename, NULL, NULL)) < 0) {
  fprintf(stderr, "Could not open input file '%s'", in_filename);
  goto end;
}
if ((ret = avformat_find_stream_info(input_format_context, NULL)) < 0) {
  fprintf(stderr, "Failed to retrieve input stream information");
  goto end;
}

avformat_alloc_output_context2(&output_format_context, NULL, NULL, out_filename);
if (!output_format_context) {
  fprintf(stderr, "Could not create output context\n");
  ret = AVERROR_UNKNOWN;
  goto end;
}
```

We're going to remux only the video, audio and subtitle types of streams so we're holding what streams we'll be using into an array of indexes.

```
number_of_streams = input_format_context->nb_streams;
streams_list = av_mallocz_array(number_of_streams, sizeof(*streams_list));
```

Just after we allocated the required memory, we're going to loop throughout all the streams and for each one we need to create new out stream into our output format context, using the avformat_new_stream function. Notice that we're marking all the streams that aren't video, audio or subtitle so we can skip them after.

```
for (i = 0; i < input_format_context->nb_streams; i++) {
  AVStream *out_stream;
  AVStream *in_stream = input_format_context->streams[i];
  AVCodecParameters *in_codecpar = in_stream->codecpar;
  if (in_codecpar->codec_type != AVMEDIA_TYPE_AUDIO &&
      in_codecpar->codec_type != AVMEDIA_TYPE_VIDEO &&
      in_codecpar->codec_type != AVMEDIA_TYPE_SUBTITLE) {
    streams_list[i] = -1;
    continue;
  }
  streams_list[i] = stream_index++;
  out_stream = avformat_new_stream(output_format_context, NULL);
  if (!out_stream) {
    fprintf(stderr, "Failed allocating output stream\n");
    ret = AVERROR_UNKNOWN;
    goto end;
  }
  ret = avcodec_parameters_copy(out_stream->codecpar, in_codecpar);
  if (ret < 0) {
    fprintf(stderr, "Failed to copy codec parameters\n");
    goto end;
  }
}
```

Now we can create the output file.

```
if (!(output_format_context->oformat->flags & AVFMT_NOFILE)) {
  ret = avio_open(&output_format_context->pb, out_filename, AVIO_FLAG_WRITE);
  if (ret < 0) {
    fprintf(stderr, "Could not open output file '%s'", out_filename);
    goto end;
  }
}

ret = avformat_write_header(output_format_context, NULL);
if (ret < 0) {
  fprintf(stderr, "Error occurred when opening output file\n");
  goto end;
}
```

After that, we can copy the streams, packet by packet, from our input to our output streams. We'll loop while it has packets (`av_read_frame`), for each packet we need to re-calculate the PTS and DTS to finally write it (`av_interleaved_write_frame`) to our output format context.

```
while (1) {
  AVStream *in_stream, *out_stream;
  ret = av_read_frame(input_format_context, &packet);
  if (ret < 0)
    break;
  in_stream  = input_format_context->streams[packet.stream_index];
  if (packet.stream_index >= number_of_streams || streams_list[packet.stream_index] < 0) {
    av_packet_unref(&packet);
    continue;
  }
  packet.stream_index = streams_list[packet.stream_index];
  out_stream = output_format_context->streams[packet.stream_index];
  /* copy packet */
  packet.pts = av_rescale_q_rnd(packet.pts, in_stream->time_base, out_stream->time_base,
AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX);
  packet.dts = av_rescale_q_rnd(packet.dts, in_stream->time_base, out_stream->time_base,
AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX);
  packet.duration = av_rescale_q(packet.duration, in_stream->time_base, out_stream-
>time_base);
  // https://ffmpeg.org/doxygen/trunk/structAVPacket.html#ab5793d8195cf4789dfb3913b7a693903
  packet.pos = -1;


//https://ffmpeg.org/doxygen/trunk/group__lavf__encoding.html#ga37352ed2c63493c38219d935e71d
b6c1
  ret = av_interleaved_write_frame(output_format_context, &packet);
  if (ret < 0) {
    fprintf(stderr, "Error muxing packet\n");
    break;
  }
  av_packet_unref(&packet);
}
```

To finalize we need to write the stream trailer to an output media file with av_write_trailer function.

```
av_write_trailer(output_format_context);
```

Now we're ready to test it and the first test will be a format (video container) conversion from a MP4 to a MPEG-TS video file. We're basically making the command line `ffmpeg input.mp4 -c copy output.ts` with libav.

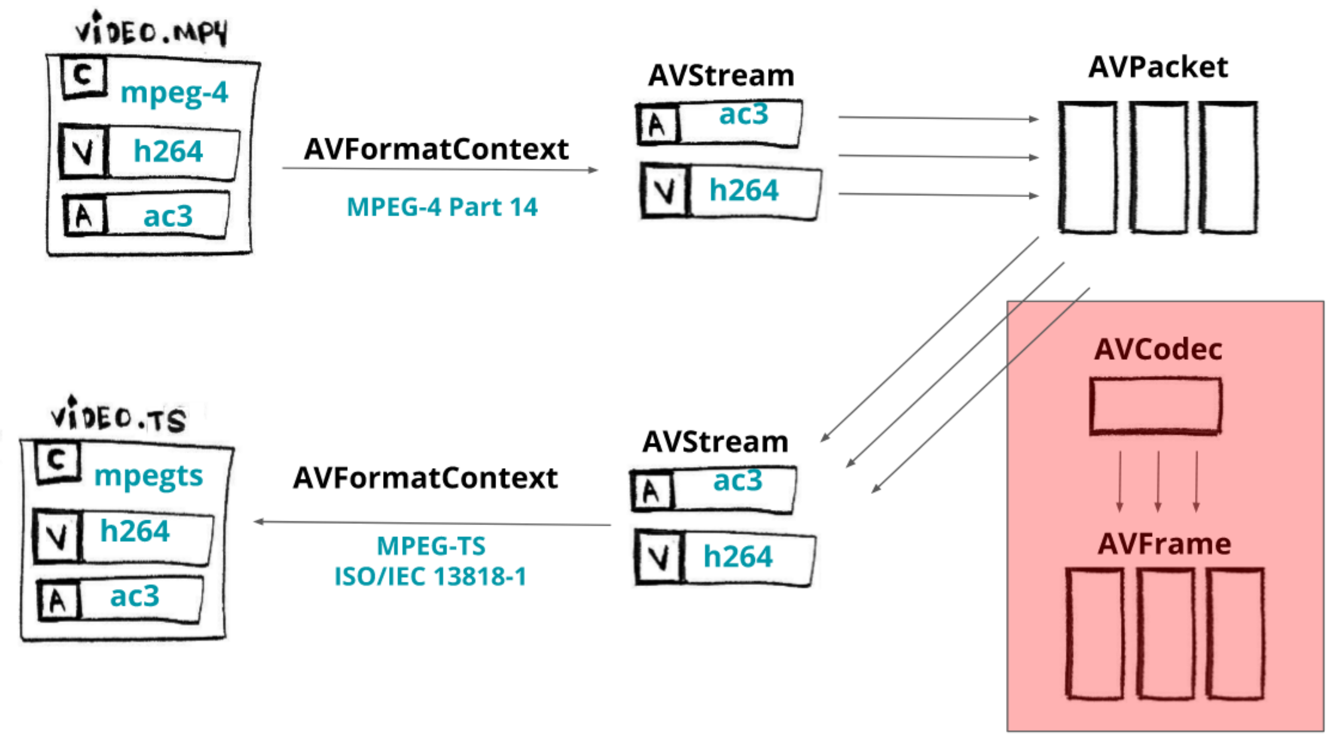
```
make run_remuxing_ts
```

It's working!!! don't you trust me?! you shouldn't, we can check it with `ffprobe`:

```
ffprobe -i remuxed_small_bunny_1080p_60fps.ts

Input #0, mpegts, from 'remuxed_small_bunny_1080p_60fps.ts':
  Duration: 00:00:10.03, start: 0.000000, bitrate: 2751 kb/s
  Program 1
    Metadata:
      service_name    : Service01
      service_provider: FFmpeg
    Stream #0:0[0x100]: Video: h264 (High) ([27][0][0][0] / 0x001B), yuv420p(progressive),
1920x1080 [SAR 1:1 DAR 16:9], 60 fps, 60 tbr, 90k tbn, 120 tbc
    Stream #0:1[0x101]: Audio: ac3 ([129][0][0][0] / 0x0081), 48000 Hz, 5.1(side), fltp, 320
kb/s
```

To sum up what we did here in a graph, we can revisit our initial [idea about how libav works](#) but showing that we skipped the codec part.



Before we end this chapter I'd like to show an important part of the remuxing process, **you can pass options to the muxer**. Let's say we want to delivery [MPEG-DASH](#) format for that matter we need to use [fragmented mp4](#) (sometimes referred as `fmp4`) instead of MPEG-TS or plain MPEG-4.

With the command line we can do that easily.

```
ffmpeg -i non_fragmented.mp4 -movflags frag_keyframe+empty_moov+default_base_moof
fragmented.mp4
```

Almost equally easy as the command line is the libav version of it, we just need to pass the options when write the output header, just before the packets copy.

```
AVDictionary* opts = NULL;
av_dict_set(&opts, "movflags", "frag_keyframe+empty_moov+default_base_moof", 0);
ret = avformat_write_header(output_format_context, &opts);
```

We now can generate this fragmented mp4 file:

```
make run_remuxing_fragmented_mp4
```

But to make sure that I'm not lying to you. You can use the amazing site/tool gpac/mp4box.js or the site http://mp4parser.com/ to see the differences, first load up the "common" mp4.

```
     0        20 ftyp
    20         8 free
    28    2edf27 mdat
2edf4f      2a7c moov
```

As you can see it has a single `mdat` atom/box, **this is place where the video and audio frames are**. Now load the fragmented mp4 to see which how it spreads the `mdat` boxes.

```
      0        18 ftyp
     18       481 moov
    499       c6c moof
   1105    1d7c9e mdat
 1d8da3       884 moof
 1d9627     bd84a mdat
 296e71       3b4 moof
 297225     58a4f mdat
 2efc74        ba mfra
```