This is CS50

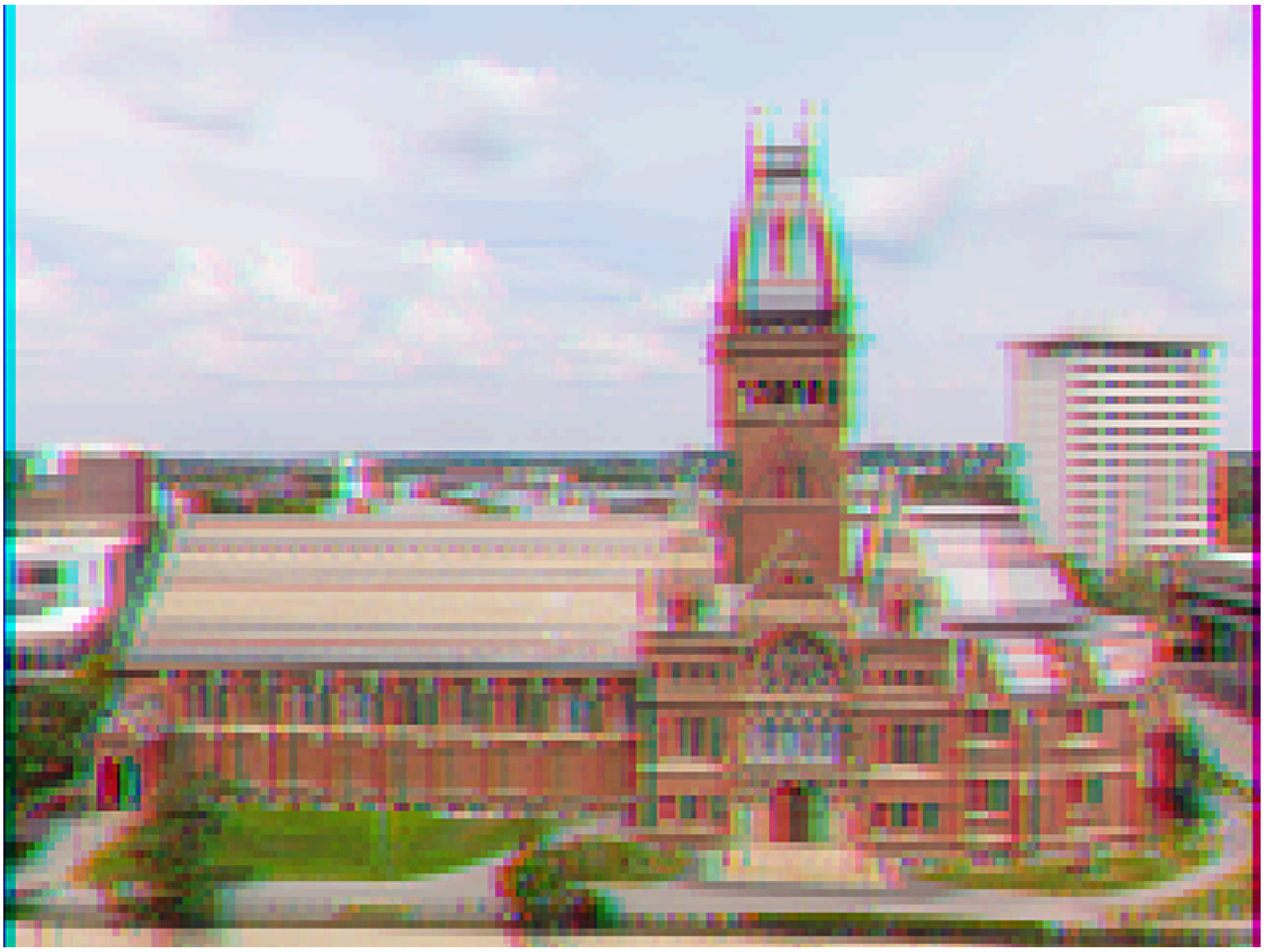
CS50's Introduction to Computer Science

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)

Recover



Problem to Solve

In anticipation of this problem, we spent the past several days taking photos around campus, all of which were saved on a digital camera as JPEGs on a memory card. Unfortunately, we somehow deleted them all! Thankfully, in the computer world, "deleted" tends not to mean "deleted" so much as "forgotten." Even though the camera insists that the card is now blank, we're pretty sure that's not quite true. Indeed, we're hoping (er, expecting!) you can write a program that recovers the photos for us!

In a file called `recover.c` in a folder called `recover`, write a program to recover JPEGs from a memory card.

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/recover.zip
```

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

Then execute

```
unzip recover.zip
```

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

```
rm recover.zip
```

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

Now type

```
cd recover
```

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

```
recover/ $
```

Execute `ls` by itself, and you should see two files: `recover.c` and `card.raw`.

Background

Even though JPEGs are more complicated than BMPs, JPEGs have “signatures,” patterns of bytes that can distinguish them from other file formats. Specifically, the first three bytes of JPEGs are

```
0xff 0xd8 0xff
```

from first byte to third byte, left to right. The fourth byte, meanwhile, is either `0xe0`, `0xe1`, `0xe2`, `0xe3`, `0xe4`, `0xe5`, `0xe6`, `0xe7`, `0xe8`, `0xe9`, `0xea`, `0xeb`, `0xec`, `0xed`, `0xee`, or `0xef`. Put another way, the fourth byte’s first four bits are `1110`.

Odds are, if you find this pattern of four bytes on media known to store photos (e.g., my memory card), they demarcate the start of a JPEG. To be fair, you might encounter these patterns on some disk purely by chance, so data recovery isn’t an exact science.

Fortunately, digital cameras tend to store photographs contiguously on memory cards, whereby each photo is stored immediately after the previously taken photo. Accordingly, the start of a JPEG usually demarks the end of another. However, digital cameras often initialize cards with a FAT file system whose “block size” is 512 bytes (B). The implication is that these cameras only write to those cards in units of 512 B. A photo that’s 1 MB (i.e., 1,048,576 B) thus takes up 1048576 ÷ 512 = 2048 “blocks” on a memory card. But so does a photo that’s, say, one byte smaller (i.e., 1,048,575 B)! The wasted space on disk is called “slack space.” Forensic investigators often look at slack space for remnants of suspicious data.

The implication of all these details is that you, the investigator, can probably write a program that iterates over a copy of my memory card, looking for JPEGs’ signatures. Each time you find a signature, you can open a new file for writing and start filling that file with bytes from my memory card, closing that file only once you encounter another signature. Moreover, rather than read my memory card’s bytes one at a time, you can read 512 of them at a time into a buffer for efficiency’s sake. Thanks to FAT, you can trust that JPEGs’ signatures will be “block-aligned.” That is, you need only look for those signatures in a block’s first four bytes.

Realize, of course, that JPEGs can span contiguous blocks. Otherwise, no JPEG could be larger than 512 B. But the last byte of a JPEG might not fall at the very end of a block. Recall the possibility of slack space. But not to worry. Because this memory card was brand-new when I started snapping photos, odds are it’d been “zeroed” (i.e., filled with 0s) by the manufacturer, in which case any slack space will be filled with 0s. It's okay if those trailing 0s end up in the JPEGs you recover; they should still be viewable.

Now, I only have one memory card, but there are a lot of you! And so I've gone ahead and created a "forensic image" of the card, storing its contents, byte after byte, in a file called `card.raw`. So that you don't waste time iterating over millions of 0s unnecessarily, I've only imaged the first few megabytes of the memory card. But you should ultimately find that the image contains 50 JPEGs.

Specification

Implement a program called `recover` that recovers JPEGs from a forensic image.

- Implement your program in a file called `recover.c` in a directory called `recover`.
- Your program should accept exactly one command-line argument, the name of a forensic image from which to recover JPEGs.
- If your program is not executed with exactly one command-line argument, it should remind the user of correct usage, and `main` should return `1`.
- If the forensic image cannot be opened for reading, your program should inform the user as much, and `main` should return `1`.
- The files you generate should each be named `###.jpg`, where `###` is a three-digit decimal number, starting with `000` for the first image and counting up.
- Your program, if it uses `malloc`, must not leak any memory.

Hints

Click the below toggles to read some advice!

▼ Write some pseudocode before writing more code

If unsure how to solve the larger problem, break it down into smaller problems that you can probably solve first. For instance, this problem is really only a handful of problems:

1. Accept a single command-line argument: the name of a memory card
2. Open the memory card
3. While there's still data left to read in the memory card
    1. Create JPEGs from the data

Let's write some pseudcode as comments to remind you to do just that:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
```

```
{
    // Accept a single command-line argument

    // Open the memory card

    // While there's still data left to read from the memory card

        // Create JPEGs from the data
}
```

▼ Convert the pseudocode to code

First, consider how to accept a single command-line argument. If the user misuses the program, you should tell them the program's proper usage.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    // Accept a single command-line argument
    if (argc != 2)
    {
        printf("Usage: ./recover FILE\n");
        return 1;
    }

    // Open the memory card

    // While there's still data left to read from the memory card

        // Create JPEGs from the data
}
```

Now that you've checked for proper usage, you can open the memory card. Keep in mind that you can open `card.raw` programmatically with `fopen`, as with the below.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    // Accept a single command-line argument
    if (argc != 2)
    {
        printf("Usage: ./recover FILE\n");
        return 1;
    }

    // Open the memory card
    FILE *card = fopen(argv[1], "r");

    // While there's still data left to read from the memory card
```

```
        // Create JPEGs from the data
}
```

You should, of course, check to be sure the file was opened properly! If it wasn't, tell the user and exit the program: we'll leave this part up to you.

Next, your program should read the data from the card you've opened, until there is no longer any data to read. Along the way, your program should recover every one of the JPEGs from `card.raw`, storing each as a separate file in your current working directory.

First consider how to read `card.raw` all the way through. Recall that, to read data from a file, you need to temporarily store that data in a "buffer." And recall further that `card.raw` stores data in blocks of 512 bytes. As such, you'll likely want to create a buffer of 512 bytes to store blocks of data as you read them sequentially. One way of doing so is to use the `uint8_t` type from `stdint.h`, which stores exactly 8 bits (1 byte). The type is called `uint8_t` since it stores an unsigned/positive/non-negative integer that requires 8 bits of space (i.e., one byte).

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    // Accept a single command-line argument
    if (argc != 2)
    {
        printf("Usage: ./recover FILE\n");
        return 1;
    }

    // Open the memory card
    FILE *card = fopen(argv[1], "r");

    // Create a buffer for a block of data
    uint8_t buffer[512];

    // While there's still data left to read from the memory card

        // Create JPEGs from the data
}
```

It's probably *not* the best idea, though, to use 512 as a "magic number" here. Odds are you could improve this design further!

Now, consider how to read data from the memory card. Per its manual page (https://man.cs50.io/3/fread), `fread` returns the number of bytes that it has read, in which case it should either return `512` or `0`, given that `card.raw` contains some number of 512-byte blocks. In order to read every block from `card.raw`, after opening it with `fopen`, it should suffice to use a loop like this.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    // Accept a single command-line argument
    if (argc != 2)
    {
        printf("Usage: ./recover FILE\n");
        return 1;
    }

    // Open the memory card
    FILE *card = fopen(argv[1], "r");

    // Create a buffer for a block of data
    uint8_t buffer[512];

    // While there's still data left to read from the memory card
    while (fread(buffer, 1, 512, card) == 512)
    {
        // Create JPEGs from the data

    }
}
```

That way, as soon as `fread` returns `0` (which is effectively `false`), your loop will end.

Finally, it's up to you to determine how to programmatically create JPEGs as you continue to read from `card.raw`. For this, you might find the below walkthrough of use to you.

Keep in mind your program should number the files it outputs by naming each `###.jpg`, where `###` is three-digit decimal number from `000` on up. Befriend `sprintf` (https://man.cs50.io/3/sprintf) and note that `sprintf` stores a formatted string at a location in memory. Given the prescribed `###.jpg` format for a JPEG's filename, how many bytes should you allocate for that string? (Don't forget the NUL character!)

To check whether the JPEGs your program spit out are correct, simply double-click and take a look! If each photo appears intact, your operation was likely a success!

And of course, remember to `fclose` every file you've opened with `fopen`!

▼ Keep your working directory clean

Odds are the JPEGs that the first draft of your code spits out won't be correct. (If you open them up and don't see anything, they're probably not correct!) Execute the command below to delete all JPEGs in your current working directory.

```
rm *.jpg
```

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

```
rm -f *.jpg
```

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

Walkthrough

How to Test

Running the Program

```
./recover card.raw
```

Correctness

```
check50 cs50/problems/2024/x/recover
```

Style

```
style50 recover.c
```

How to Submit

```
submit50 cs50/problems/2024/x/recover
```