Recover

Implement a program that recovers JPEGs from a forensic image, per the below.

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

Background

In anticipation of this problem, we spent the past several days taking photos of people we know, all of which were saved on a digital camera as JPEGs on a memory card. (Okay, it's possible we actually spent the past several days on Facebook instead.) 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!

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.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE and then, in a terminal window, execute each of the below.

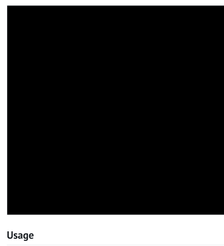
1. Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
2. If you haven't already, execute `mkdir pset4` to make (i.e., create) a directory called `pset4` in your home directory.
3. Execute `cd pset4` to change into (i.e., open) your `pset4` directory.
4. Execute `wget https://cdn.cs50.net/2019/fall/psets/4/recover/recover.zip` to download a (compressed) ZIP file with this problem's distribution.
5. Execute `unzip recover.zip` to uncompress that file.
6. Execute `rm recover.zip` followed by `yes` or `y` to delete that ZIP file.
7. Execute `ls`. You should see a directory called `recover`, which was inside of that ZIP file.
8. Execute `cd recover` to change into that directory.
9. Execute `ls`. You should see this problem's distribution, including `card.raw` and `recover.c`.

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

Walkthrough



Usage

Your program should behave per the examples below.

```
$ ./recover
Usage: ./recover image
```

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

Hints

Keep in mind that you can open `card.raw` programmatically with `fopen`, as with the below, provided `argv[1]` exists.

```
FILE *file = fopen(argv[1], "r");
```

When executed, your program should recover every one of the JPEGs from `card.raw`, storing each as a separate file in your current working directory. Your program should number the files it outputs by naming each `###.jpg`, where `###` is three-digit decimal number from `000` on up. (Befriend `sprintf`.) You need not try to recover the JPEGs' original names. 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!

Odds are, though, 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.

If you'd like to create a new type to store a byte of data, you can do so via the below, which defines a new type called `BYTE` to be a `uint8_t` (a type defined in `stdint.h`, representing an 8-bit unsigned integer).

```
typedef uint8_t BYTE;
```

Keep in mind, too, that you can read data from a file using `fread`, which will read data from a file into a location in memory and return the number of items successfully read from the file.

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/2020/x/recover
```

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

```
style50 recover.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/recover
```