CS133C PROGRAMMING IN C

CRN 26949

Fall 2021

Week 8

Due Tue 11/30/21

It is time to bring together some concepts that we have been learning! This assignment is adapted from CS50’s [Recover](https://cs50.harvard.edu/x/2021/psets/4/recover/) under Creative Commons [Attribution-NonCommercial-ShareAlike 4.0 International](https://creativecommons.org/licenses/by-nc-sa/4.0/) (CC BY-NC-SA 4.0) license.

## Instructions

NOTE: if you are working with a partner, only ONE person needs to upload the zipped folder to Moodle.

1. Download the **cs133c\_recover.zip** from Moodle and save it in under your **…/cs133c/week8 directory**.
2. Unzip the **cs133c\_recover.zip** folder, and you will see two files in **cs133c\_recover** folder: **card.raw** and **recover.c**.
3. You will need to implement your program in the file called **recover.c**.
4. Commenting:
   1. At the top of the program, please write your **name** and **date** completed. In addition, briefly write about what does this program do.
      1. If you are working with a partner, include **both team members’ names** and indicates **each member’s responsibility**.
   2. Provide comments throughout the code, if needed, especially for the blocks that are complicated or tricky.

\*\*\* I don’t mean that we need to comment excessively. Just think about you looking at another person’s code, or showing your code to yourself one month later, how do we make it readable and easy to understand? (Using clear function names and variable names are helpful too!) \*\*\*

1. When you are done with your program, compile (**gcc -o recover recover.c**) and run it (**./recover card.raw**), you should be able to recover **50 JPEG** **images** from the **card.raw** file. (yay, successed!)
2. At the terminal, under the **…/cs133c\_recover** directory, **remove** the 50 JPEG image files you just recovered using **rm -f \*.jpg**(this will force delete all the JPEG image files in the current directory). If this command doesn’t work, you can try **Remove-Item -force \*.jpg** (for PowerShell) or **del -r \*.jpg** (for cmd).
3. Finally, after you make sure that the **recover.c** can help you successfully recover the JPEG image files in **card.raw** and then **delete the recovered JPEG files**. There should be three files in your **recover** folder at this point: **card.raw**, the complete **recover.c**, and **recover.exe** (or **recover.out**).
4. **Compress** this **cs133c\_recover** folder and name it **cs133c\_recover\_initials** (for example, **cs133c\_recover\_nw**; please **leave no space** in the file name).
5. Upload the **.zip** folder **cs133c\_recover\_initials** to Moodle.

## Problem -- [Recover](https://cs50.harvard.edu/x/2021/psets/4/recover/#recover)

### [Background](https://cs50.harvard.edu/x/2021/psets/4/recover/#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.

### [Specification](https://cs50.harvard.edu/x/2021/psets/4/recover/#specification)

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

* Implement your program in a file called **recover.c** under your **../cs133c\_recover directory**.
* Your program should **accept two command-line arguments**, the program name **./recover** and the name of a forensic image from which to recover JPEGs, **card.raw**. See below:

***./recover card.raw***

* If your program is not executed with exactly two command-line arguments, it should remind the user of correct usage, and the main function should return 1.
* If the forensic image cannot be opened for reading, your program should inform the user as much, and the main function should return 1.
* The files you generate should each be named **###.jpg**, where**###** is a three-digit decimal number, starting with **001** (pay attention here, image name should start from 001, not 000)for the first image and counting up.
* Your program, if it uses malloc, must not leak any memory.

### [Walkthrough](https://cs50.harvard.edu/x/2021/psets/4/recover/#walkthrough)

11 mins [walkthrough video](https://cs50.harvard.edu/x/2021/psets/4/recover/#walkthrough) (opens in a new window) by Brian Yu.

### [Hints](https://cs50.harvard.edu/x/2021/psets/4/recover/#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], "**rb**"); (pay attention to the mode here!)

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 **001** on up. Befriend [sprintf() function](https://manual.cs50.io/3/sprintf) (opens in a new window) 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.

Again, if the command above doesn’t work, you can try **Remove-Item -force \*.jpg** (for PowerShell) or **del -r \*.jpg** (for cmd).

* 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](https://man.cs50.io/3/fread), which will read data from a file into a location in memory and return the number of items successfully read from the file.