

NOTICE TO PODCAST SUBSCRIBERS

The source code for CS 50's library as well as for Fall 2007's problem sets in general can be found at <http://cs50.tv/>.

Problem Set 4: Forensics

out of 66 points

due by 7:00 P.M. on Friday, 2 November 2007

Goals.

The goals of this problem set are to:

- Familiarize you with file I/O.
- Get you comfortable with hexadecimal, pointers, and arrays as buffers.
- Help Mr. Boddy.
- Get you outdoors.

Recommended Reading.

Per the syllabus, no books are required for this course. If you feel that you would benefit from some supplementary reading, though, below are some recommendations.

- Section 21 – 26, 31, 32, 35, and 40 of <http://www.howstuffworks.com/c.htm>.
- Chapters 18, 24, 25, 27, and 28 of *Absolute Beginner's Guide to C*.
- Chapters 9, 11, 14, and 16 of *Programming in C*.

Collaboration.

Lest there be confusion otherwise, know that the Hacker Edition of this problem set explicitly allows collaboration with a partner but this standard edition does **not** allow any such collaboration.

(Shhh, theirs is harder.)

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this class that you have submitted or will submit to another. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

All forms of cheating will be dealt with harshly.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the staff.

Grading Metrics.

Each question is worth the number of points specified parenthetically in line with it.

Your responses to questions requiring exposition will be graded on the basis of their clarity and correctness. Your responses to questions requiring code will be graded along the following axes.

Correctness. To what extent does your code adhere to the problem's specifications?

Design. To what extent is your code written clearly, efficiently, elegantly, and/or logically?

Style. To what extent is your code commented and indented, your variables aptly named, *etc.*?

Rest assured that grades will be normalized across sections at term's end.

Getting Started.

0. SSH to `nice.fas.harvard.edu` and execute the command below.

```
cp -r ~/cs50/pub/ps/distributions/ps4/standard ~/cs50/ps4
```

Navigate your way to `~/cs50/ps4/`. If you list the contents of your current working directory, you should see the below. If you don't, don't hesitate to ask the staff for assistance.

```
questions.txt  recover/  whodunit/
```

As this output implies, most of your work for this problem set will be organized within two subdirectories. Let's get started.

1. (4 points.) If you've ever seen Windows XP's default wallpaper (think rolling hills and blue skies), then you've seen a BMP. If you've ever looked at a webpage, you've probably seen a GIF. If you've ever looked at a digital photo, you've probably seen a JPEG. Read up a bit on the BMP, GIF, and JPEG file formats.¹ Then, in `ps4/questions.txt`, tell us the below.
- How many different colors does each format support?
 - Which of these formats supports animation?
 - What's the difference between lossy and lossless compression?
 - Which of these formats is lossy-compressed?
2. (2 points.) Curl up with the article from MIT below.

```
http://www.fas.harvard.edu/~cs50/pub/ps/shared/ps4/article.pdf
```

Though somewhat technical, you should find the article's language quite accessible. Once you've read the article, answer each of the following questions in a sentence or more in `ps4/questions.txt`.

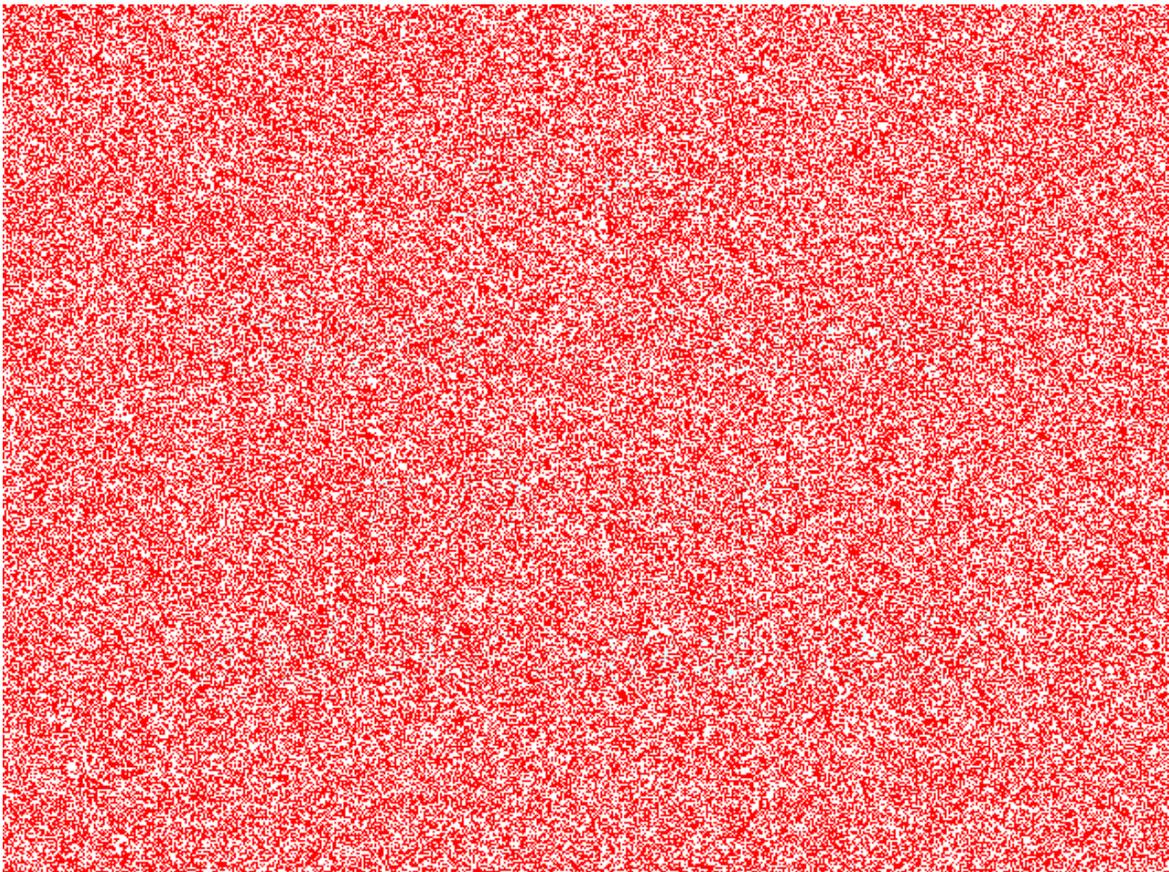
- What happens, technically speaking, when a file is deleted on a FAT file system?
- What can someone like you do to ensure (with high probability) that files you delete cannot be recovered?

¹ For this question, you're welcome to consult *How Computers Work*, Google, Wikipedia, a friend, or anyone else, so long as your words are ultimately your own!

Whodunit.

3. Welcome to Tudor Mansion. Your host, Mr. John Boddy, has met an untimely end—he’s the victim of foul play. To win this game, you must determine the answer to these three questions: Who done it? Where? And with what weapon?

Unfortunately for you (though even more unfortunately for Mr. Boddy), the only evidence you have is a 24-bit BMP file called `clue.bmp`, pictured below, that he whipped up for his estranged granddaughter in his final moments.² Hidden among this file’s red “noise” is a message from Mr. Boddy.

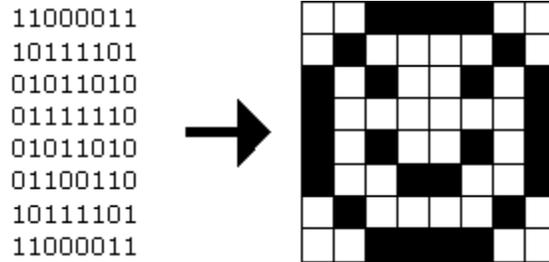


You long ago threw away that piece of red plastic from childhood that would solve this mystery for you, and so you must attack it as the computer scientist that you almost are.

But, first, some background.

² Realize that this BMP is in color even though you might have printed this document in black and white.

4. Perhaps the simplest way to represent an image is with a grid of pixels (*i.e.*, dots), each of which can be of a different color. For black-and-white images, we thus need 1 bit per pixel, as 0 could represent black and 1 could represent white, as in the below.³



In this sense, then, is an image just a bitmap (*i.e.*, a map of bits). For more colorful images, you simply need more bits per pixel. A file format (like GIF) that supports “8-bit color” uses 8 bits per pixel. A file format (like BMP or JPEG) that supports “24-bit color” uses 24 bits per pixel.⁴ (We just spoiled #1!)

A 24-bit BMP like Mr. Boddy’s uses 8 bits to signify the amount of red in a pixel’s color, 8 bits to signify the amount of green in a pixel’s color, and 8 bits to signify the amount of blue in a pixel’s color. If you’ve ever heard of RGB color, well, there you have it: red, green, blue.

If the R, G, and B values of some pixel in a BMP are, say, 0xff, 0x00, and 0x00 in hexadecimal, that pixel is purely red, as 0xff (otherwise known as 255 in decimal) implies “a lot of red,” while 0x00 and 0x00 imply “no green” and “no blue,” respectively. Given how red Mr. Boddy’s BMP is, it clearly has a lot of pixels with those RGB values. But it also has a few with other values.

Incidentally, XHTML and CSS (languages in which webpages can be written) model colors in this same way. In fact, for more RGB “codes,” see the URL below.

http://www.w3schools.com/html/html_colors.asp

Now let’s get more technical. Recall that a file is just a sequence of bits, arranged in some fashion. A 24-bit BMP file, then, is essentially just a sequence of bits, every 24 of which happen to represent some pixel’s color. But a BMP file also contains some “metadata,” information like an image’s height and width. That metadata is stored at the beginning of the file in the form of two data structures generally referred to as “headers” (not to be confused with C’s header files). The first of these headers, called `BITMAPFILEHEADER`, is 14 bytes long. (Recall that 1 byte equals 8 bits.) The second of these headers, called `BITMAPINFOHEADER`, is 40 bytes long. Immediately following these headers is the actual bitmap: an array of bytes, triples of which represent a pixel’s color. However, BMP stores these triples backwards (*i.e.*, as BGR), with 8 bits for blue, followed by 8 bits for green, followed by 8 bits for red.⁵ In

³ Image adapted from <http://www.brackeen.com/vga/bitmaps.html>.

⁴ BMP actually supports 1-, 4-, 8-, 16-, 24-, and 32-bit color.

⁵ Some BMPs also store the entire bitmap backwards, with an image’s top row at the end of the BMP file. But we’ve stored this problem set’s BMPs as described herein, with each bitmap’s top row first and bottom row last.

other words, were we to convert the 1-bit smiley above to a 24-bit smiley, substituting red for black, a 24-bit BMP would represent his bitmap as follows, where `0000ff` signifies red and `ffffff` signifies white; we've highlighted in red all instances of `0000ff`.

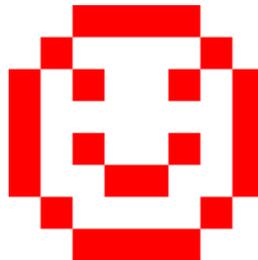
```

ffffff  fffffff  0000ff  0000ff  0000ff  0000ff  fffffff  fffffff
ffffff  0000ff  fffffff  fffffff  fffffff  fffffff  0000ff  fffffff
0000ff  fffffff  0000ff  fffffff  fffffff  0000ff  fffffff  0000ff
0000ff  fffffff  fffffff  fffffff  fffffff  fffffff  fffffff  0000ff
0000ff  fffffff  0000ff  fffffff  fffffff  0000ff  fffffff  0000ff
0000ff  fffffff  fffffff  0000ff  0000ff  fffffff  fffffff  0000ff
ffffff  0000ff  fffffff  fffffff  fffffff  fffffff  0000ff  fffffff
ffffff  fffffff  0000ff  0000ff  0000ff  0000ff  fffffff  fffffff
  
```

To be clear, recall that a hexadecimal digit represents 4 bits. Accordingly, `ffffff` in hexadecimal actually signifies `111111111111111111111111` in binary.

Okay, stop! Don't proceed further until you're sure you understand why `0000ff` represents a red pixel in a 24-bit BMP file.

- Okay, let's transition from theory to practice. Navigate your way to `~/cs50/ps4/whodunit/`. In that directory is a file called `smiley.bmp`. If you feel like SFTPing that file to your hard drive and double-clicking it, you'll see that it resembles the below, albeit much smaller (since it's only 8 pixels by 8 pixels).



Open this file in `xxd`, a "hex editor," by executing the command below.

```
xxd -c 24 -g 3 -s 54 smiley.bmp
```

You should see the below; we've again highlighted in red all instances of `0000ff`.

```

0000036: fffffff fffffff 0000ff 0000ff 0000ff 0000ff fffffff fffffff .....
000004e: fffffff 0000ff fffffff fffffff fffffff fffffff 0000ff fffffff .....
0000066: 0000ff fffffff 0000ff fffffff fffffff 0000ff fffffff 0000ff .....
000007e: 0000ff fffffff fffffff fffffff fffffff fffffff fffffff 0000ff .....
0000096: 0000ff fffffff 0000ff fffffff fffffff 0000ff fffffff 0000ff .....
00000ae: 0000ff fffffff fffffff 0000ff 0000ff fffffff fffffff 0000ff .....
00000c6: fffffff 0000ff fffffff fffffff fffffff fffffff 0000ff fffffff .....
00000de: fffffff fffffff 0000ff 0000ff 0000ff 0000ff fffffff fffffff .....
00000f6: 0000 .....
  
```

In the leftmost column above are addresses within the file or, equivalently, offsets from the file's first byte, all of them given in hex. Note that `0000036` in hexadecimal is 54 in decimal. You're thus looking at byte 54 onward of `smiley.gif`, all in hexadecimal. Recall that a 24-bit

BMP's first $14 + 40 = 54$ bytes are filled with metadata. If you really want to see that metadata in addition to the bitmap, execute the command below.

```
xxd -c 24 -g 3 smiley.bmp
```

If `smiley.bmp` actually contained ASCII characters, you'd see them in `xxd`'s rightmost column instead of all of those dots.

Okay, back to Mr. Boddy.

- (30 points.) Open up `whodunit.h` with Nano and look over our (well, Microsoft's) definitions of `BITMAPFILEHEADER` and `BITMAPINFOHEADER`. You don't need to understand everything going on in that file, but know that we're going to use those structures in order to open `clue.bmp`. Take a look at the URLs in that file if curious as to what each of the structure's fields means.

Now open up `whodunit.c` with Nano and read it over carefully. Be sure you understand what each function call is doing, particularly those to `fopen`, `fEOF`, `fread`, `fwrite`, and `fclose`. Turn to those functions' man pages or the URL below for usage information.

<http://www.cppreference.com/stdio/>

Note that all this program does at present is copy a BMP file. Confirm as much by building the program with Make (remember how?) and executing the command below.

```
whodunit smiley.bmp new.bmp
```

If you list the contents of your current working directory, you should now see `new.bmp`, the size of which is equal to that of `smiley.bmp`. Moreover, if you open `new.bmp` with `xxd`, you'll find that its contents are identical to `smiley.bmp`'s. And if you SFTP `new.bmp` to your hard drive, you'll even find that it looks the same as `smiley.bmp`! Wow, we've just re-implemented `cp`. And only for BMP files, no less.

Odds are, though, you're going to thank us. Because you've still got work to do.

Modify `whodunit.c` so that the program doesn't just copy a BMP file but instead reveals Mr. Boddy's final words.

OMG, what? How?

Well, think back to childhood when you held that piece of red plastic over similarly hidden messages.⁶ Essentially, the plastic turned everything red. But, certain colors still shone through. Implement that same idea in `whodunit`. You needn't futz with `BITMAPFILEHEADER` or `BITMAPINFOHEADER`. Focus instead on changing what the program does within its `while`

⁶ If you remember no such piece of plastic, best to ask a friend or TF about his or her childhood.

loop. Ultimately, if you execute a command like the below, stored in `answer.bmp` should be Mr. Boddy's message.

```
whodunit clue.bmp answer.bmp
```

If what you need do is a mystery to you, 'tis precisely the point! Think about it. Try out some ideas. Find one that works. Do it for Mr. Boddy.

There's nothing hidden in `smiley.bmp`, but feel free to test your program out on its pixels nonetheless, if only because that BMP is small and you can thus compare it and your own program's output with `xxd` during development.

Rest assured that more than one solution is possible. So long as your program's output is readable (by your teaching fellow), Mr. Boddy will rest in peace.

Be sure to check in your code often with RCS! (Remember how?)

Free Dinner.

7. (30 points.) Just the other day, two tourists were strolling about Harvard's campus with a Canon EOS 20D digital camera snapping photos, all of which were stored as JPEGs on a 1GB CompactFlash (CF) card. Rather than act like typical tourists, taking photos of John Harvard's foot (blech) and squirrels (I mean really, why?), these two opted to shoot identifiable but non-obvious places and things on campus. Call them artistic. Actually, call one of them David.⁷

Unfortunately, somewhere between the camera and David's computer, the CF card somehow got corrupted! Both his Mac and his PC refuse to recognize the card as having any photos, even though he's sure they're on there. Both operating systems want to format the card, but, thus far, he's refused to let them, hoping instead someone can come to the rescue.

Write a program in `~/cs50/ps4/recover/` called `recover` that recovers David's photos.

OMG, not again, what?

Well, here's the thing. Even though JPEGs are more complicated than BMPs, JPEGs have "signatures," patterns of bytes that distinguish them from other file formats. In fact, most JPEGs begin with one of two sequences of bytes. Specifically, the first four bytes of most JPEGs are either

```
0xff 0xd8 0xff 0xe0
```

or

```
0xff 0xd8 0xff 0xe1
```

⁷ Call the other Daniel Armendariz of MIT, whose skills with a camera far exceed mine.

from first byte to fourth byte, left to right. Odds are, if you find one of these patterns of bytes on a disk known to store photos (*e.g.*, my CF card), they demark the start of a JPEG.⁸

Fortunately, digital cameras tend to store photographs contiguously on CF 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 generally initialize CF 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 $1048576 \div 512 = 2048$ “blocks” on a CF 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 CF 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 CF card, closing that file only once you encounter another signature. Moreover, rather than read my CF 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. Fortunately, I bought a brand-new CF card for my stroll about campus. Odds are, that CF card was “zeroed” (*i.e.*, filled with 0s) by the manufacturer. Because I didn’t outright delete any photos we took, the only bits on that CF card should belong to actual photos or be 0s. And it’s okay if some trailing 0s (*i.e.*, slack space) end up in the JPEGs your program spits out; they should still be viewable.

Know that each of the JPEGs we took was less than 2 MB in size and that there are 38 JPEGs on my CF card. Please recover them all! Since I’ve but one CF card, I’ve gone ahead and created a “forensic image” of the card, storing its contents, byte after byte, in a file called `card.raw` in `~cs50/pub/ps/shared/ps4/`. So that you don’t waste time iterating over millions of 0s unnecessarily, I’ve only imaged the first 39 MB of the CF card. Even though we’ve asked FAS Computer Services to increase your FAS account’s quota for CS 50 this semester, you don’t need a copy of that file in your own account. Simply open our copy with `fopen` via its full path: `/home/c/s/cs50/pub/ps/shared/ps4/card.raw`.

Notice that `~/cs50/ps4/recover/` is empty. It’s up to you to create, at least, a `Makefile` and `recover.c` for this program. You probably don’t need a `recover.h`, but you’re welcome to create one. For simplicity, you may hard-code the path to `card.raw` in your program; your program need not accept any command-line arguments. When executed, though, 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. (You may

⁸ To be sure, you might encounter these patterns on some disk purely by chance, so data recovery isn’t an exact science.

find `sprintf` of assistance.) You need not try to recover the JPEGs' original names. To check whether the JPEGs your program spit out are correct, simply SFTP them to your own hard drive, double-click, and take a look. If each photo appears intact, your operation was likely a success!

Odds are, though, the first JPEGs your program 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. Be sure to check in your code often with RCS, especially since you'll likely be executing `rm` quite a bit!

8. And now the proverbial icing on the cake. You are hereby challenged to find as many of the places and things that we photographed on campus as possible. To prove that you found some place or thing, you must email a picture of yourself (or of someone in your section) posing next to or near it to `cs50+pics@fas.harvard.edu`, also CCing your TF, by 7:00 P.M. on Friday, 9 November 2007. In the body of the email, state your FAS username, your TF's name, and the place or thing that you found.

The section whose students collectively identify the most photographs by way of these proofs will be awarded a prize: **free dinner in the Square** with their TF some night. The TFs did not accompany me on my stroll, so yours is welcome to join forces with you on this quest. In the event of a tie, the section that submits the most proofs first shall be declared the winner of dinner.

Submitting Your Work.

9. Ensure that your work is in `~/cs50/ps4/`. Submit your work by executing the command below.

```
cs50submit ps4
```

Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.