Table of Contents

Whodunit

tl;dr

Answer some questions and then implement a program that reveals a hidden message in a BMP, per the below.

```
$ ./whodunit clue.bmp verdict.bmp
```

Distribution

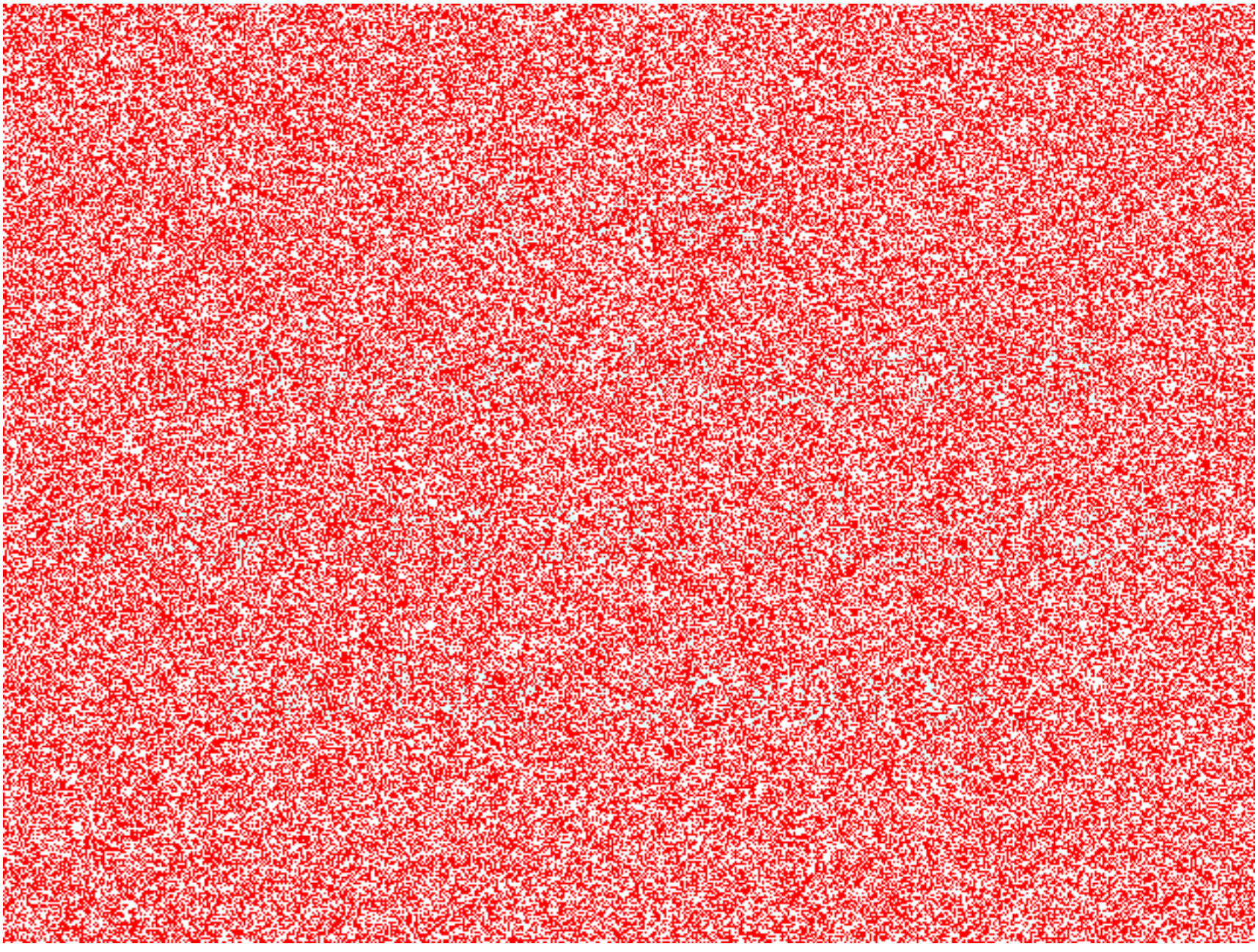
Downloading

```
$ wget https://github.com/cs50/problems/archive/whodunit.zip (https://git
$ unzip whodunit.zip
$ rm whodunit.zip
$ mv problems-whodunit whodunit
$ cd whodunit
$ ls
bmp.h  clue.bmp  copy.c  large.bmp  small.bmp  smiley.bmp
```

Background

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 whodunit.

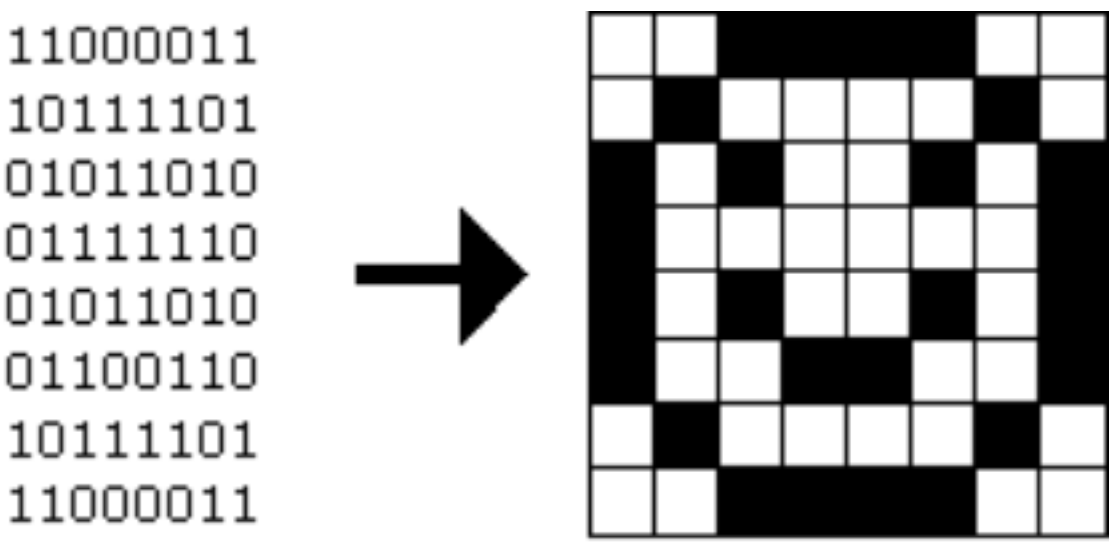
Unfortunately for you (though even more unfortunately for Mr. Boddy), the only evidence you have is a 24-bit BMP file called `clue.bmp`, pictured below, that Mr. Boddy whipped up on his computer in his final moments. Hidden among this file's red "noise" is a drawing of whodunit.



You long ago threw away that piece of red plastic from childhood (https://s-media-cache-ak0.pinimg.com/564x/a6/10/0c/a6100c96163cd9ec3e6df3621d5db6d5.jpg) that would solve this mystery for you, and so you must attack it as a computer scientist instead.

But, first, some background.

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. [1]



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 (https://en.wikipedia.org/wiki/GIF)) that supports "8-bit color" uses 8 bits per pixel. A file format (like BMP (https://en.wikipedia.org/wiki/BMP_file_format), JPEG (https://en.wikipedia.org/wiki/JPEG), or PNG (https://en.wikipedia.org/wiki/Portable_Network_Graphics)) that supports "24-bit color" uses 24 bits per pixel. (BMP actually supports 1-, 4-, 8-, 16-, 24-, and 32-bit color.)

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, HTML and CSS (languages in which webpages can be written) model colors in this same way. If curious, see http://en.wikipedia.org/wiki/Web_colors (http://en.wikipedia.org/wiki/Web_colors) for more details.

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, (almost) 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. (Incidentally, these headers have evolved over time. This problem only expects that you support the latest version of Microsoft's BMP format, 4.0, which debuted with Windows 95.) 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. (In 1-, 4-, and 16-bit BMPs, but not 24- or 32-, there's an additional header right after `BITMAPINFOHEADER` called `RGBQUAD`, an array that defines "intensity values" for each of the colors in a device's palette.) 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. (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.) In 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 store this bitmap as follows, where `0000ff` signifies red and `ffffff` signifies white; we've highlighted in red all instances of `0000ff`.

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

Because we've presented these bits from left to right, top to bottom, in 8 columns, you can actually see the red smiley if you take a step back.

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. Within CS50 IDE's file browser, double-click **smiley.bmp**, and you should see a tiny smiley face that's only 8 pixels by 8 pixels. Via the drop-down menu in that file's newly opened tab, change **100%** to **800%** to zoom in a bit, and you should see a larger version, a la the below. (If it seems blurry, be sure that **Smooth** atop the window isn't checked.) At this zoom level, you can really see the image's pixels (as big squares).

Okay, let's now look at the underlying bytes that compose `smiley.bmp` using `xxd`, a command-line "hex editor." Execute

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

in a terminal window, and you should see the below. (You might have to increase the terminal window's size.) As before, we've highlighted in red all instances of `0000ff`.

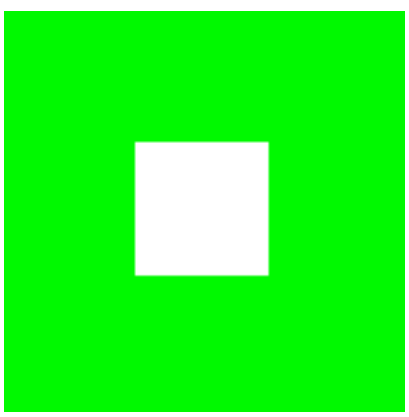
```
0000036: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff  .......
000004e: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff  .......
0000066: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff  .......
000007e: 0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff  .......
0000096: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

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 `00000036` in hexadecimal is `54` in decimal. You're thus looking at byte `54` onward of `smiley.bmp`. 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.

So, `smiley.bmp` is 8 pixels wide by 8 pixels tall, and it's a 24-bit BMP (each of whose pixels is represented with 24 ÷ 8 = 3 bytes). Each row (aka "scanline") thus takes up (8 pixels) × (3 bytes per pixel) = 24 bytes, which happens to be a multiple of 4. It turns out that BMPs are stored a bit differently if the number of bytes in a scanline is not, in fact, a multiple of 4. In `small.bmp`, for instance, is another 24-bit BMP, a green box that's 3 pixels wide by 3 pixels wide. If you view it (as by double-clicking it), you'll see that it resembles the below, albeit much smaller. (Indeed, you might need to zoom in again to see it.)



Each scanline in `small.bmp` thus takes up (3 pixels) × (3 bytes per pixel) = 9 bytes, which is not a multiple of 4. And so the scanline is "padded" with as many zeroes as it takes to extend the scanline's length to a multiple of 4. In other words, between 0 and 3 bytes of padding are needed for each scanline in a 24-bit BMP. (Understand why?) In the case of `small.bmp`, 3 bytes' worth of zeroes are needed, since (3 pixels) × (3 bytes per pixel) + (3 bytes of padding) = 12 bytes, which is indeed a multiple of 4.

To "see" this padding, go ahead and run the below.

```
xxd -c 12 -g 3 -s 54 small.bmp
```

Note that we're using a different value for `-c` than we did for `smiley.bmp` so that `xxd` outputs only 4 columns this time (3 for the green box and 1 for the padding). You should see output like the below; we've highlighted in green all instances of `00ff00`.

```
0000036: 00ff00 00ff00 00ff00 000000  ............
0000042: 00ff00 ffffff 00ff00 000000  ............
000004e: 00ff00 00ff00 00ff00 000000  ............
```

For contrast, let's use `xxd` on `large.bmp`, which looks identical to `small.bmp` but, at 12 pixels by 12 pixels, is four times as large. Go ahead and execute the below; you may need to widen your window to avoid wrapping.

```
xxd -c 36 -g 3 -s 54 large.bmp
```

You should see output like the below; we've again highlighted in green all instances of `00ff00`

```
0000036: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
000005a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
000007e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
00000a2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
00000c6: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 0(
00000ea: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 0(
000010e: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 0(
0000132: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff ffffff 00ff00 0(
0000156: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
000017a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
000019e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
00001c2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 0(
```

Worthy of note is that this BMP lacks padding! After all, (12 pixels) × (3 bytes per pixel) = 36 bytes is indeed a multiple of 4.

Knowing all this has got to be useful!

Okay, `xxd` only showed you the bytes in these BMPs. How do we actually get at them programmatically? Well, in `copy.c` is a program whose sole purpose in life is to create a copy of a BMP, piece by piece. Of course, you could just use `cp` for that. But `cp` isn't going to help Mr. Boddy. Let's hope that `copy.c` does!

Go ahead and compile `copy.c` into a program called `copy` using `make`. (Remember how?) Then execute a command like the below.

```
./copy smiley.bmp copy.bmp
```

If you then execute ls (with the appropriate switch), you should see that `smiley.bmp` and `copy.bmp` are indeed the same size. Let's double-check that they're actually the same! Execute the below.

```
diff smiley.bmp copy.bmp
```

If that command tells you nothing, the files are indeed identical. (Note that some programs, like Photoshop, include trailing zeroes at the ends of some BMPs. Our version of `copy` throws those away, so don't be too worried if you try to copy a BMP that you've downloaded or made only to find that the copy is actually a few bytes smaller than the original.) Feel free to open both files (as by double-clicking each) to confirm as much visually. But diff does a byte-by-byte comparison, so its eye is probably sharper than yours!

So how now did that copy get made? It turns out that `copy.c` relies on `bmp.h`. Let's take a look. Open up `bmp.h`, and you'll see actual definitions of those headers we've mentioned, adapted from Microsoft's own implementations thereof. In addition, that file defines `BYTE`, `DWORD`, `LONG`, and `WORD`, data types normally found in the world of Windows programming. Notice how they're just aliases for primitives with which you are (hopefully) already familiar. It appears that `BITMAPFILEHEADER` and `BITMAPINFOHEADER` make use of these types. This file also defines a `struct` called `RGBTRIPLE` that, quite simply, "encapsulates" three bytes: one blue, one green, and one red (the order, recall, in which we expect to find RGB triples actually on disk).

Why are these `struct`s useful? Well, recall that a file is just a sequence of bytes (or, ultimately, bits) on disk. But those bytes are generally ordered in such a way that the first few represent something, the next few represent something else, and so on. "File formats" exist because the world has standardized what bytes mean what. Now, we could just read a file from disk into RAM as one big array of bytes. And we could just remember that the byte at location `[i]` represents one thing, while the byte at location `[j]` represents another. But why not give some of those bytes names so that we can retrieve them from memory more easily? That's precisely what the `struct`s in `bmp.h` allow us to do. Rather than think of some file as one long sequence of bytes, we can instead think of it as a sequence of `struct`s.

Recall that `smiley.bmp` is 8 by 8 pixels, and so it should take up $14 + 40 + (8 \times 8) \times 3 = 246$ bytes on disk. (Confirm as much if you'd like using `ls`.) Here's what it thus looks like on disk according to Microsoft:

offset	type	name	
0	WORD	bfType	BITMAPFILEHEADER
2	DWORD	bfSize	
6	WORD	bfReserved1	
8	WORD	bfReserved2	
10	DWORD	bfOffBits	
14	DWORD	biSize	BITMAPINFOHEADER
18	LONG	biWidth	
22	LONG	biHeight	
26	WORD	biPlanes	
28	WORD	biBitCount	
30	DWORD	biCompression	
34	DWORD	biSizeImage	
38	LONG	biXPelsPerMeter	
42	LONG	biYPelsPerMeter	
46	DWORD	biClrUsed	
50	DWORD	biClrImportant	
54	BYTE	rgbtBlue	RGBTRIPLE
55	BYTE	rgbtGreen	
56	BYTE	rgbtRed	
57	BYTE	rgbtBlue	RGBTRIPLE
58	BYTE	rgbtGreen	
59	BYTE	rgbtRed	
...			
243	BYTE	rgbtBlue	RGBTRIPLE
244	BYTE	rgbtGreen	
245	BYTE	rgbtRed	

As this figure suggests, order does matter when it comes to `struct`s' members. Byte 57 is `rgbtBlue` (and not, say, `rgbtRed`), because `rgbtBlue` is defined first in `RGBTRIPLE`. Our use, incidentally, of the *`attribute`* called *`packed`* ensures that `clang` does not try to "word-align" members (whereby the address of each member's first byte is a multiple of 4), lest we end up with "gaps" in our `struct`s that don't actually exist on disk. No need to worry about that particular implementation detail, though.

Lastly, you may have noticed in `copy.c` that, whenever we output an error message, we use `fprintf` (the first argument to which is `stderr`) instead of the more-familiar `printf`. It turns out that `printf` prints messages to "standard output" (aka `stdout`), the destination of which is typically a user's terminal window. But "standard error (aka `stderr`) also exists, the destination of which is also typically (and perhaps confusingly!) a user's terminal window. But via `stdout` and `stderr` can a programmer keep error messages separated from non-error messages so that, if the user wants, one or the other (or both) can be "redirected" (with `>`) or "piped" (with `|`) somewhere other than the user's terminal window.

In other words,

```
printf("hello, world\n");
```

is equivalent to

```
fprintf(stdout, "hello, world\n");
```

but the former is more succinct. In order to print an error message to `stderr`, though, do use `fprintf` per the below.

```
fprintf(stderr, "Usage: ./whodunit infile outfile\n");
```

Questions

Go ahead and pull up the URLs to which `BITMAPFILEHEADER` and `BITMAPINFOHEADER` are attributed, per the comments in `bmp.h`.

Rather than hold your hand further on a stroll through `copy.c`, we're instead going to ask you some questions and let you teach yourself how the code therein works. As always, `man` is your friend, and so, now, is Microsoft Developer Network (aka MSDN). If not sure on first glance how to answer some question, do some quick research and figure it out! You might want to turn to https://reference.cs50.net/stdio (https://reference.cs50.net/stdio) as well.

In `questions.txt`, answer each of the following questions in a sentence or more.

1. What's `stdint.h`?
2. What's the point of using `uint8_t`, `uint32_t`, `int32_t`, and `uint16_t` in a program?
3. How many bytes is a `BYTE`, a `DWORD`, a `LONG`, and a `WORD`, respectively?
4. What (in ASCII, decimal, or hexadecimal) must the first two bytes of any BMP file be? Leading bytes used to identify file formats (with high probability) are generally called "magic numbers."
5. What's the difference between `bfSize` and `biSize`?
6. What does it mean if `biHeight` is negative?
7. What field in `BITMAPINFOHEADER` specifies the BMP's color depth (i.e., bits per pixel)?

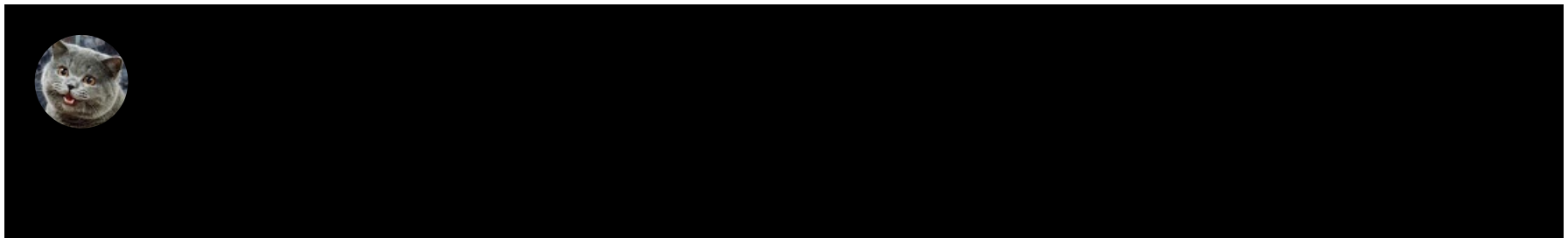
8. Why might `fopen` return `NULL` in lines 24 and 32 of `copy.c`?
9. Why is the third argument to `fread` always `1` in our code?
10. What value does line 65 of `copy.c` assign to `padding` if `bi.biWidth` is `3`?
11. What does `fseek` do?
12. What is `SEEK_CUR`?

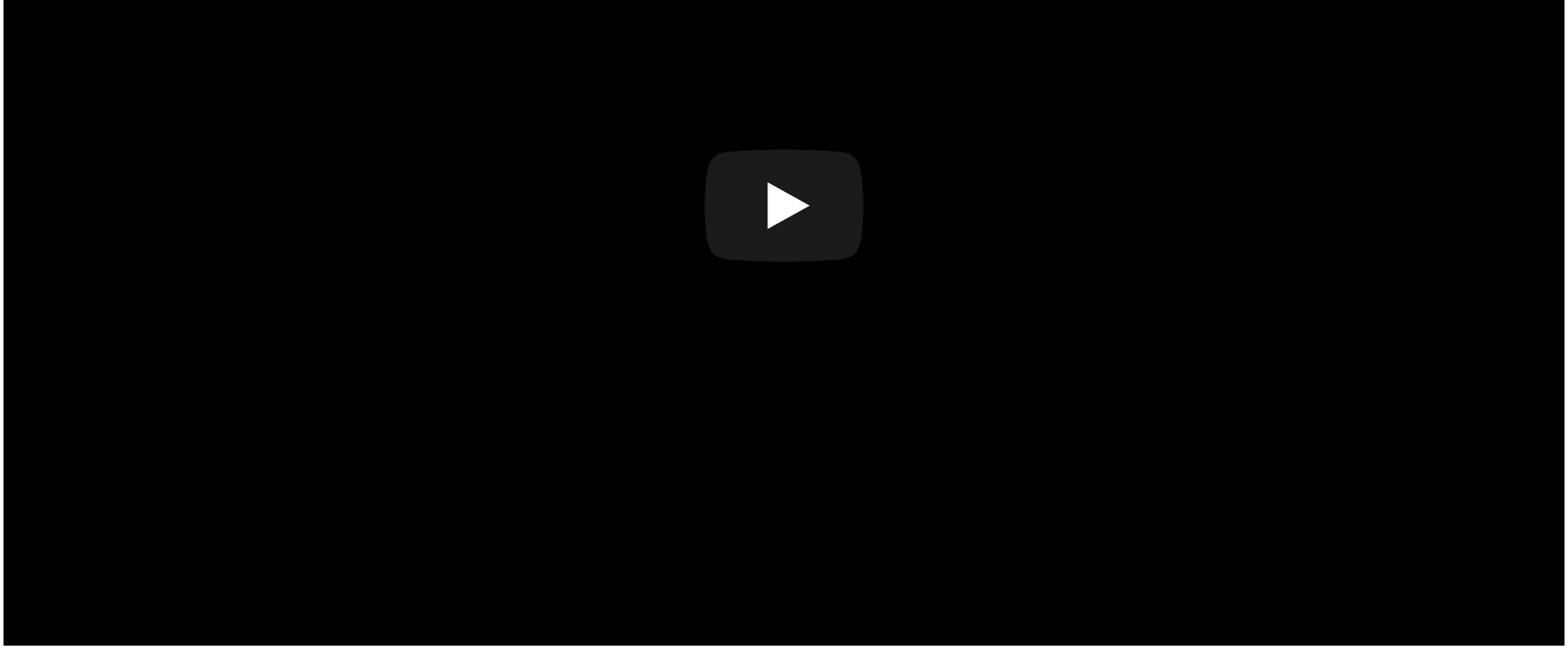
Specification

Implement a program called `whodunit` that reveals Mr. Boddy's drawing in such a way that you can recognize whodunit.

- Implement your program in a file called `whodunit.c` in a directory called `whodunit`.
- Your program should accept exactly two command-line arguments: the name of an input file to open for reading followed by the name of an output file to open for writing.
- If your program is executed with fewer or more than two command-line arguments, it should remind the user of correct usage, as with `fprintf` (to `stderr`), and `main` should return `1`.
- If the input file cannot be opened for reading, your program should inform the user as much, as with `fprintf` (to `stderr`), and `main` should return `2`.
- If the output file cannot be opened for writing, your program should inform the user as much, as with `fprintf` (to `stderr`), and `main` should return `3`.
- If the input file is not a 24-bit uncompressed BMP 4.0, your program should inform the user as much, as with `fprintf` (to `stderr`), and `main` should return `4`.
- Upon success, `main` should `0`.

Walkthrough





Usage

Your program should behave per the examples below. Assumed that the underlined text is what some user has typed.

```
$ ./whodunit
Usage: ./whodunit infile outfile
$ echo $?
1
```

```
$ ./whodunit clue.bmp verdict.bmp
$ echo $?
0
```

Hints

Think back to childhood when you held that piece of red plastic over similarly hidden messages. (If you remember no such piece of plastic, best to ask a classmate about his or her childhood.) Essentially, the plastic turned everything red but somehow revealed those messages. Implement that same idea in `whodunit`. Like `copy`, your program should accept exactly two command-line arguments. And if you execute a command like the below, stored in `verdict.bmp` should be a BMP in which Mr. Boddy's drawing is no longer covered with noise.

```
./whodunit clue.bmp verdict.bmp
```

Allow us to suggest that you begin tackling this mystery by executing the command below.

```
cp copy.c whodunit.c
```

Then add and/or change just a few lines of code.

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 Mr. Boddy's drawing is identifiable (by you), no matter its legibility, Mr. Boddy will rest in peace.

When submitting this problem, you'll be asked whodunit!

Testing

Because `whodunit` can be implemented in several ways, afraid you can't check your implementation's correctness with `check50`!

Staff's Solution

No solution from the staff, lest it spoil your fun!

FAQs

None so far! Reload this page periodically to check if any arise!

Changelog

- 2016-09-30
  - Added explanation of `fprintf`.
- 2016-09-27
  - Added a clarification regarding which file pointer to use for `fprintf`.
- 2016-09-23
  - Initial release.

---

1. Image adapted from http://www.brackeen.com/vga/bitmaps.html (http://www.brackeen.com/vga/bitmaps.html).