Problem 4-5: Grow

This is CS50. Harvard University. Fall 2014.

Table of Contents

Questions? Feel free to head to CS50 on Reddit[1], CS50 on StackExchange[2], the `#cs50ap` channel on CS50x Slack[3] (after signing up), or the CS50 Facebook group[4].

Objectives

- Acquaint you with file I/O.
- Get you more comfortable with data structures, hexadecimal
- Gently introduce pointers.

Recommended Reading*

- Chapters 18, 24, 25, 27, and 28 of *Absolute Beginner's Guide to C*
- Chapters 9, 11, 14, and 16 of *Programming in C*
- http://www.cprogramming.com/tutorial/cfileio.html
- http://en.wikipedia.org/wiki/BMP_file_format

[1] https://www.reddit.com/r/cs50
[2] http://cs50.stackexchange.com
[3] https://cs50x.slack.com
[4] https://www.facebook.com/groups/cs50

- http://en.wikipedia.org/wiki/Hexadecimal
- http://en.wikipedia.org/wiki/Jpg

* The Wikipedia articles are a bit dense; feel free to skim or skip!

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problems is not permitted (unless explicitly stated otherwise) except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code or writing to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes and tests is not permitted at all. Collaboration on the final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from your instructor. If a violation of this policy is suspected and confirmed, your instructor reserves the right to impose local sanctions on top of any disciplinary outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, such as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past years' quizzes, tests, and solutions thereto.

- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own solutions to problems online so that others might help you identify and fix a bug or other issue.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problems or your own final project.
- Whiteboarding solutions to problems with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problems.
- Failing to cite (as with comments) the origins of code, writing, or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz or test.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problems to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problems online or elsewhere.
- Splitting a problem's workload with another individual and combining your work (unless explicitly authorized by the problem itself).

- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem and basing your own solution on it.

Assessment

Your work on this problem set will be evaluated along four axes primarily.

Scope
: To what extent does your code implement the features required by our specification?

Correctness
: To what extent is your code consistent with our specifications and free of bugs?

Design
: To what extent is your code written well (i.e., clearly, efficiently, elegantly, and/or logically)?

Style
: To what extent is your code readable (i.e., commented and indented with variables aptly named)?

To obtain a passing grade in this course, all students must ordinarily submit all assigned problems unless granted an exception in writing by the instructor.

Getting Started

Welcome back!

As always, first open a terminal window and execute

```
update50
```

to make sure your workspace is up-to-date.

Next, navigate to your `~/workspace/unit4` directory. Instead of downloading a new distro for this assignment, we're going to *recursively copy* the distro we prepared for

Problem 4-4[5]. If you don't already have that distro, head to the link and download it and unzip the ZIP file according to the instructions there. Confirm you have a `whodunit` directory as with

```
ls
```

Then, from within your `~/workspace/unit4` directory, execute the following:

```
cp -r whodunit grow
```

This will copy the entire contents of the `whodunit` directory into a newly-created directory called `grow`. If you navigate inside the new `grow` directory, you should find that it is an exact duplicate of what was in your `whodunit` directory. Since we'll be using nearly all of the same files, this is good. We can delete a few files though, and we do so by way of the following command:

```
rm -f clue.bmp whodunit.c verdict.bmp
```

Make sure, though, that after executing that command your directory contains at least the following files:

```
bmp.h  copy.c  large.bmp  small.bmp  smiley.bmp
```

Off we go!

Refresher Course

Recall from Problem 4-4 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 set only expects that you support version 4.0 (the latest)

[5] http://cdn.cs50.net/ap/1516/problems/4/4/4-4.html

of Microsoft's BMP format, 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.

Let's 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
```

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

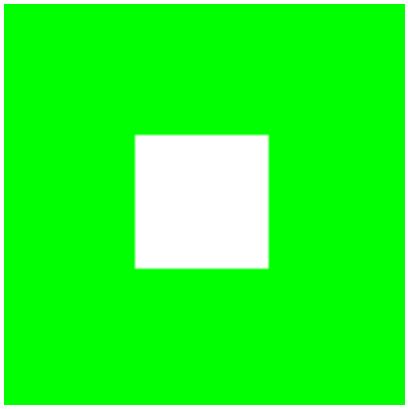
```
0000036: ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
  ........................
000004e: ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
  ........................
0000066: 0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
 ........................
000007e: 0000ff ffffff 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. (Interesting way to maybe hide some information in a file!)

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 with Image Viewer (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 00ff00 00ff
  ..................................
000005a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
000007e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
00000a2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
00000c6: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff
 ffffff 00ff00 00ff00 00ff00 00ff00  ..................................
00000ea: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff
 ffffff 00ff00 00ff00 00ff00 00ff00  ..................................
000010e: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff
 ffffff 00ff00 00ff00 00ff00 00ff00  ..................................
0000132: 00ff00 00ff00 00ff00 00ff00 ffffff ffffff ffffff
 ffffff 00ff00 00ff00 00ff00 00ff00  ..................................
0000156: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
000017a: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
000019e: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
00001c2: 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff00 00ff
  ..................................
```

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.

(Re)size Matters

Alright, next challenge! Implement now in `grow.c` a program called `grow` that resizes 24-bit uncompressed BMPs by a factor of `n`. Your program should accept exactly three command-line arguments, per the below usage, whereby the first (`n`) must be a positive integer less than or equal to 100, the second the name of the file to be resized, and the third the name of the resized version to be written.

```
Usage: ./grow n infile outfile
```

With a program like this, we could have created `large.bmp` out of `small.bmp` by resizing the latter by a factor of 4 (i.e., by multiplying both its width and its height by 4), per the below.

```
./grow 4 small.bmp large.bmp
```

You're welcome to get started by copying `copy.c` and naming the copy `grow.c` (remember how?). But spend some time thinking about what it means to resize a BMP. (You may assume that `n` times the size of `infile` will not exceed 2^{32} - 1.) Decide which of the fields in `BITMAPFILEHEADER` and `BITMAPINFOHEADER` you might need to modify. Consider whether or not you'll need to add or subtract padding to scanlines. And be thankful that we don't expect you to support fractional `n` between 0 and 1! (At least, not until and unless you tackle Problem 4-6[6]) But we do expect you to support a value of `1` for `n`, the result of which should be an `outfile` with dimensions identical to `infile`'s.

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 1516.unit4.grow bmp.h grow.c
```

If you'd like to play with the staff's own implementation of `grow`, you may execute the below.

```
~cs50/unit4/grow
```

If you'd like to peek at, e.g., `large.bmp`'s headers (in a more user-friendly way than `xxd` allows), you may execute the below.

```
~cs50/unit4/peek large.bmp
```

Better yet, if you'd like to compare your outfile's headers against the staff's, you might want to execute commands like the below while inside your `~/workspace/unit4/grow` directory. (Think about what each is doing.)

[6] http://cdn.cs50.net/ap/1516/problems/4/6/4-6.html

```
./grow 4 small.bmp student.bmp
~cs50/unit4/grow 4 small.bmp staff.bmp
~cs50/unit4/peek student.bmp staff.bmp
```

If you happen to use `malloc`, be sure to use `free` so as not to leak memory. Try using `valgrind` to check for any leaks!

Here's Zamyla again!

https://www.youtube.com/watch?v=g8LEbJapnj8
This was Problem 4-5.