David J. Malan
malan@harvard.edu

Filter

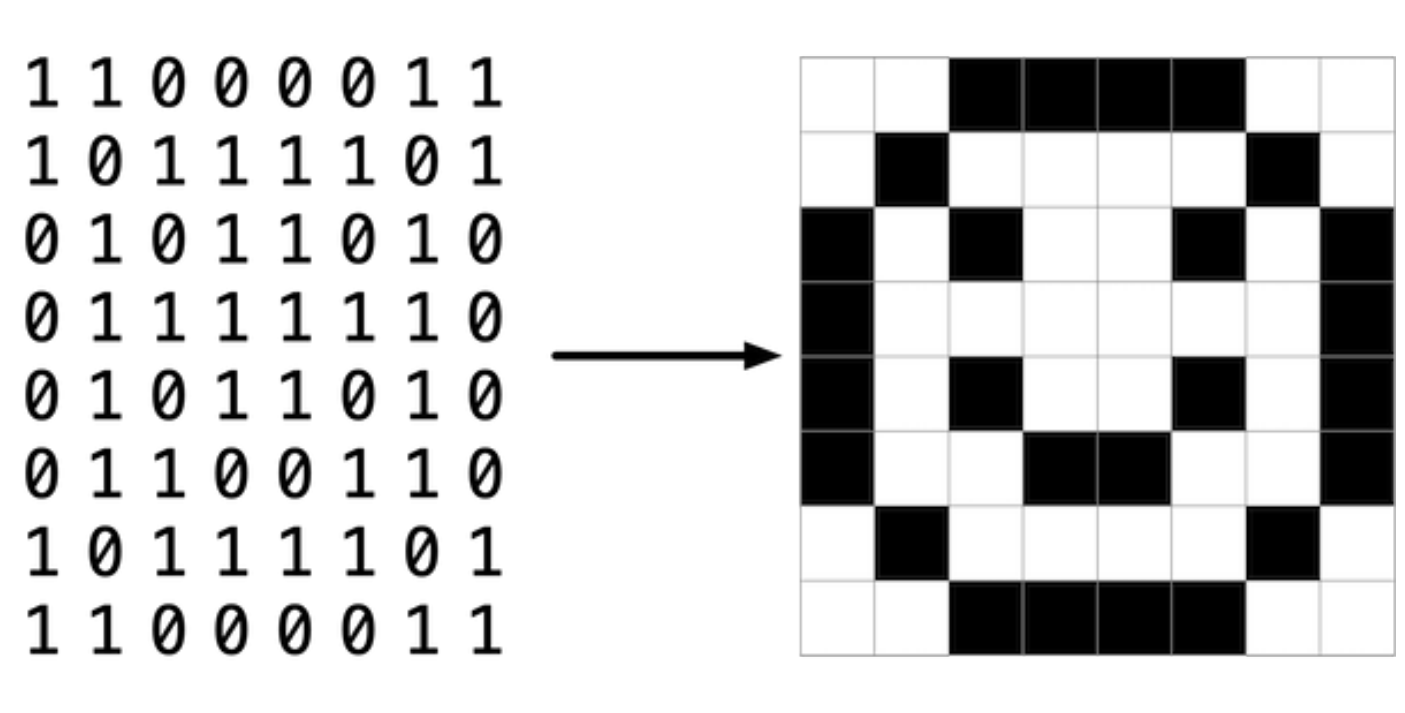
Implement a program that applies filters to BMPs, per the below.

```
$ ./filter -r image.bmp reflected.bmp
```

Background

Bitmaps

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.



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 BMP, JPEG, or PNG) 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 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.

A Bit(map) 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 uses 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. 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.

Notice that you could represent a bitmap as a 2-dimensional array of pixels: where the image is an array of rows, each row is an array of pixels. Indeed, that's how we've chosen to represent bitmap images in this problem.

Image Filtering

What does it even mean to filter an image? You can think of filtering an image as taking the pixels of some original image, and modifying each pixel in such a way that a particular effect is apparent in the resulting image.

Grayscale

One common filter is the "grayscale" filter, where we take an image and want to convert it to black-and-white. How does that work?

Recall that if the red, green, and blue values are all set to `0x00` (hexadecimal for `0`), then the pixel is black. And if all values are set to `0xff` (hexadecimal for `255`), then the pixel is white. So long as the red, green, and blue values are all equal, the result will be varying shades of gray along the black-white spectrum, with higher values meaning lighter shades (closer to white) and lower values meaning darker shades (closer to black).

So to convert a pixel to grayscale, we just need to make sure the red, green, and blue values are all the same value. But how do we know what value to make them? Well, it's probably reasonable to expect that if the original red, green, and blue values were all pretty high, then the new value should also be pretty high. And if the original values were all low, then the new value should also be low.

In fact, to ensure each pixel of the new image still has the same general brightness or darkness as the old image, we can take the average of the red, green, and blue values to determine what shade of grey to make the new pixel.

If you apply that to each pixel in the image, the result will be an image converted to grayscale.

Sepia

Most image editing programs support a "sepia" filter, which gives images an old-timey feel by making the whole image look a bit reddish-brown.

An image can be converted to sepia by taking each pixel, and computing new red, green, and blue values based on the original values of the three.

There are a number of algorithms for converting an image to sepia, but for this problem, we'll ask you to use the following algorithm. For each pixel, the sepia color values should be calculated based on the original color values per the below.

```
sepiaRed = .393 * originalRed + .769 * originalGreen + .189 * or
sepiaGreen = .349 * originalRed + .686 * originalGreen + .168 *
sepiaBlue = .272 * originalRed + .534 * originalGreen + .131 * c
```

Of course, the result of each of these formulas may not be an integer, but each value could be rounded to the nearest integer. It's also possible that the result of the formula is a number greater than 255, the maximum value for an 8-bit color value. In that case, the red, green, and blue values should be capped at 255. As a result, we can guarantee that the resulting red, green, and blue values will be whole numbers between 0 and 255, inclusive.

Reflection

Some filters might also move pixels around. Reflecting an image, for example, is a filter where the resulting image is what you would get by placing the original image in front of a mirror. So any pixels on the left side of the image should end up on the right, and vice versa.

Note that all of the original pixels of the original image will still be present in the reflected image, it's just that those pixels may have rearranged to be in a different place in the image.

Blur

There are a number of ways to create the effect of blurring or softening an image. For this problem, we'll use the "box blur," which works by taking each pixel and, for each color value, giving it a new value by averaging the color values of neighboring pixels.

Consider the following grid of pixels, where we've numbered each pixel.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

The new value of each pixel would be the average of the values of all of the pixels that are within 1 row and column of the original pixel (forming a 3x3 box). For example, each of the color values for pixel 6 would be obtained by averaging the original color values of pixels 1, 2, 3, 5, 6, 7, 9, 10, and 11 (note that pixel 6 itself is included in the average). Likewise, the color values for pixel 11 would be be obtained by averaging the color values of pixels 6, 7, 8, 10, 11, 12, 14, 15 and 16.

For a pixel along the edge or corner, like pixel 15, we would still look for all pixels within 1 row and column: in this case, pixels 10, 11, 12, 14, 15, and 16.

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.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `mkdir pset4` to make (i.e., create) a directory called `pset4` in your home directory.
- Execute `cd pset4` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/4/filter/less/filter.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip filter.zip` to uncompress that file.
- Execute `rm filter.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `filter`, which was inside of that ZIP file.
- Execute `cd filter` to change into that directory.
- Execute `ls`. You should see this problem's distribution, including `bmp.h`, `filter.c`, `helpers.h`, `helpers.c`, and `Makefile`. You'll also see a directory called `images`, with some sample Bitmap images.

Understanding

Let's now take a look at some of the files provided to you as distribution code to get an understanding for what's inside of them.

`bmp.h`

Open up `bmp.h` (as by double-clicking on it in the file browser) and have a look.

You'll see definitions of the headers we've mentioned (`BITMAPINFOHEADER` and `BITMAPFILEHEADER`). 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.

Perhaps most importantly for you, 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 `array[i]` represents one thing, while the byte at `array[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 structs 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.

`filter.c`

Now, let's open up `filter.c`. This file has been written already for you, but there are a couple important points worth noting here.

First, notice the definition of `filters` on line 11. That string tells the program what the allowable command-line arguments to the program are: `b`, `g`, `r`, and `s`. Each of them specifies a different filter that we might apply to our images: blur, grayscale, reflection, and sepia.

The next several lines open up an image file, make sure it's indeed a BMP file, and read all of the pixel information into a 2D array called `image`.

Scroll down to the `switch` statement that begins on line 102. Notice that, depending on what `filter` we've chosen, a different function is called: if the user chooses filter `b`, the program calls the `blur` function; if `g`, then `grayscale` is called; if `r`, then `reflect` is called; and if `s`, then `sepia` is called. Notice, too, that each of these functions take as arguments the height of the image, the width of the image, and the 2D array of pixels.

These are the functions you'll (soon!) implement. As you might imagine, the goal is for each of these functions to edit the 2D array of pixels in such a way that the desired filter is applied to the image.

The remaining lines of the program take the resulting `image` and write them out to a new image file.

`helpers.h`

Next, take a look at `helpers.h`. This file is quite short, and just provides the function prototypes for the functions you saw earlier.

Here, take note of the fact that each function takes a 2D array called `image` as an argument, where `image` is an array of `height` many rows, and each row is itself another array of `width` many `RGBTRIPLE`s. So if `image` represents the whole picture, then `image[0]` represents the first row, and `image[0][0]` represents the pixel in the upper-left corner of the image.

`helpers.c`

Now, open up `helpers.c`. Here's where the implementation of the functions declared in `helpers.h` belong. But note that, right now, the implementations are missing! This part is up to you.

`Makefile`

Finally, let's look at `Makefile`. This file specifies what should happen when we run a terminal command like `make filter`. Whereas programs you may have written before were confined to just one file, `filter` seems to use multiple files: `filter.c`, `bmp.h`, `helpers.h`, and `helpers.c`. So we'll need to tell `make` how to compile this file.

Try compiling `filter` for yourself by going to your terminal and running

```
$ make filter
```

Then, you can run the program by running:

```
$ ./filter -g images/yard.bmp out.bmp
```

which takes the image at `images/yard.bmp`, and generates a new image called `out.bmp` after running the pixels through the `grayscale` function. `grayscale` doesn't do anything just yet, though, so the output image should look the same as the original yard.

Specification

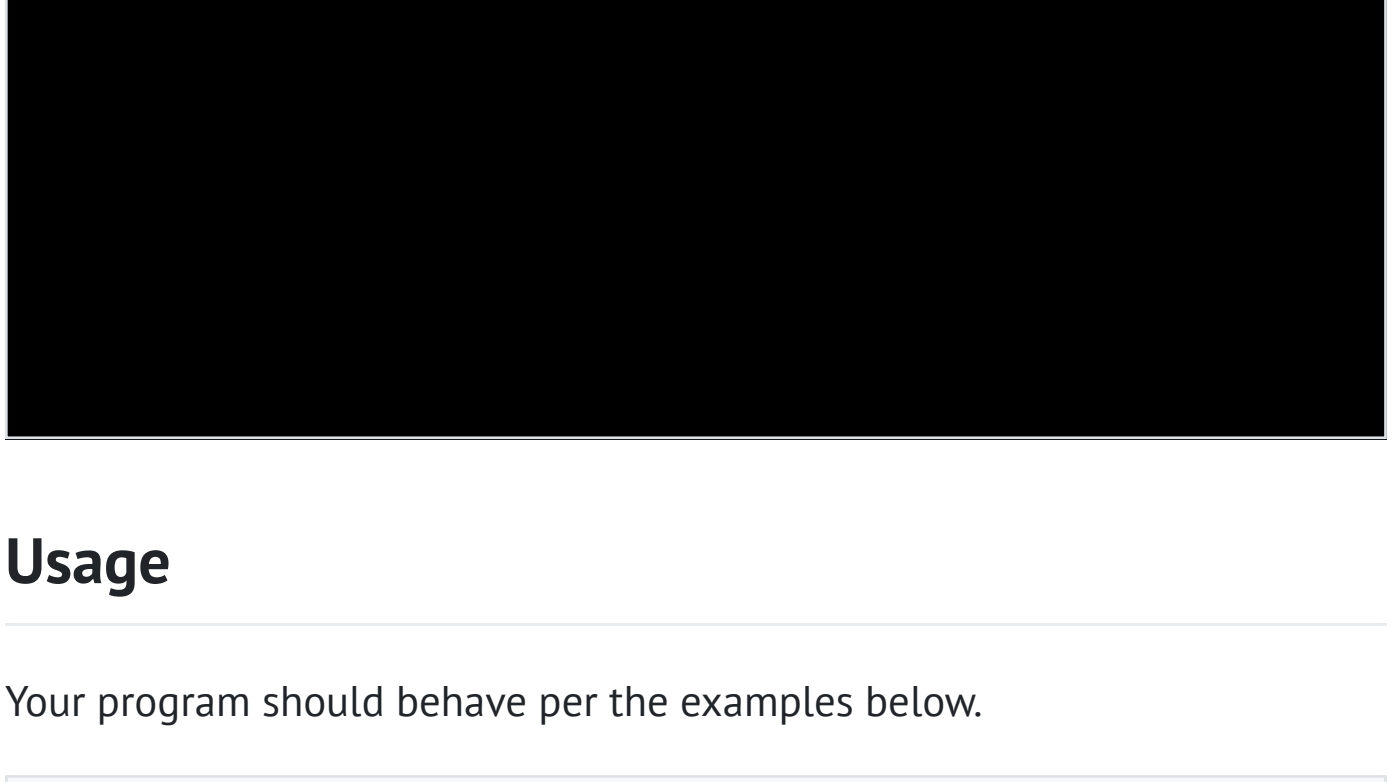
Implement the functions in `helpers.c` such that a user can apply grayscale, sepia, reflection, or blur filters to their images.

- The function `grayscale` should take an image and turn it into a black-and-white version of the same image.
- The function `sepia` should take an image and turn it into a sepia version of the same image.
- The `reflect` function should take an image and reflect it horizontally.
- Finally, the `blur` function should take an image and turn it into a box-blurred version of the same image.

You should not modify any of the function signatures, nor should you modify any other files other than `helpers.c`.

Walkthrough

Please note that there are 5 videos in this playlist.



Usage

Your program should behave per the examples below.

```
$ ./filter -g infile.bmp outfile.bmp
```

```
$ ./filter -s infile.bmp outfile.bmp
```

```
$ ./filter -r infile.bmp outfile.bmp
```

```
$ ./filter -b infile.bmp outfile.bmp
```

Hints

- The values of a pixel's `rgbtRed`, `rgbtGreen`, and `rgbtBlue` components are all integers, so be sure to round any floating-point numbers to the nearest integer when assigning them to a pixel value!

Testing

Be sure to test all of your filters on the sample bitmap files provided!

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/filter/less
```

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

```
style50 helpers.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/filter/less
```