CS 0449 – Project 1: RPS & Image Transformations

Due: Monday, October 2, 2017, at 11:59pm

Your first project is to write two programs in C that provide some experience with a wide range of the topics we have been discussing in class.

Rock, Paper, Scissors (30 points)

For the first part of this project, you will be implementing the game of Rock, Paper, Scissors. For those unfamiliar with the rules, typically the game is played with two people who use hand gestures to represent a rock (closed fist), paper (an open hand), or scissor (a vee made with your fingers.) Each person displays their choice at the same time and the winner is determined by (winner in bold):

Scissors cuts paper, **paper** covers rock, **rock** breaks scissors

Your job is to write a program where a human can play against the computer in a best-of-5 tournament. The first to win three games wins the match. Have the human player enter their choice, and then have the computer randomly pick its choice. If the two match, the game is a tie and doesn't count. Otherwise you will add one to the score of the winner. After the match is over, you should ask the user if they would like to play again.

Example:

```
Welcome to Rock, Paper, Scissors

Would you like to play? yes

What is your choice? scissors
The computer chooses rock. You lose this game!

The score is now you: 0 computer: 1

…
```

Hints

- Generating random numbers in C is a two-step part. First, we need to seed the random number generator *once* per program. The idiom to do this is:
  `srand((unsigned int)time(NULL));`
- When we need random numbers, we can use the rand() function. It returns an unsigned integer between 0 and RAND_MAX. We can use modulus to reduce it to the range we need:

```
int value = rand() % (high - low + 1) + low;
```

Image Transformations (70 points)

A Bitmap Image File (typical extension: .bmp) is a container format for a big array of pixels. There are a variety of ways that BMP files can encode image data, but we will focus on one particular form and write a program that performs two simple image transformations: Inverting the image and Converting a color image to grayscale.

Inverting an image means to take each pixel (a “picture element” – basically one discrete colored point in a larger image) and produce the “opposite” color, which we will define as being the bitwise-NOT of each pixel’s color value.

Converting a color image to grayscale is precisely what it says, we will take the various colors of the image and replace them by differing intensities of the color gray.

We will be assuming Windows Bitmap files whose contents are 24-bit RGB color. This means that each pixel is represented by a 24-bit number, split into three 8-bit parts. The first part is the intensity of the color blue, the second is the intensity of the color green, and the third is the intensity of the color red, each expressed as an integer value from 0-255. (Yes, that’d actually make it BGR and not RGB, but BMP is just weird that way…)

What to Do

For your project you will make a utility that can transform a BMP image in our assumed format using one of two operations: “invert” or “grayscale”.

Make a program called `bmp_edit` and make it so that it runs with the following command line options:

```
./bmp_edit –invert FILENAME
```

should destructively alter the BMP file named FILENAME to contain the inverse pixel colors of the originals in FILENAME. The command:

```
./bmp_edit –grayscale FILENAME
```

should destructively alter the BMP file named FILENAME to contain the pixel colors converted to grayscale as described below.

Note that we are requiring that the file specified already exist and be in the proper format. Your job is to read the pixels there and to simply overwrite the old ones with the altered ones, preserving the original file’s contents in all other locations.

Reading a BMP file

Our first step is to read and validate that the specified file is a BMP in a format we can handle. The first 14 bytes of all valid bitmap files begin with a “header” described in the table below:

Offset	length	description
0	2	Format identifier (magic number)
2	4	Size of the file in bytes
6	2	A two-byte reserved value
8	2	Another two-byte reserved value
10	4	Offset to the start of the pixel array

We can convert this table into a struct, create an instance of the struct, and read in the data from the file using fread().

Check that the format identifier is the characters “BM” (uppercase, no NUL terminator). If it is not, display an error message that we do not support the file format, and exit.

After this BMP header, there is a second header called a DIB header. DIB means a device-independent bitmap, and this header will tell us about the way the pixels are arranged in the file. This DIB header should also be made a struct with the following layout:

Offset	length	description
0	4	Size of this DIB header in bytes
4	4	Width of the image in pixels
8	4	Height of the image in pixels
12	2	Number of color planes (don’t worry)
14	2	Number of bits per pixel
16	4	Compression scheme used
20	4	Image size in bytes
24	4	Horizontal resolution
28	4	Vertical resolution
32	4	Number of colors in the palette
36	4	Number of important colors

Many of these fields we will read, but not use in our program, so don’t worry about understanding all of them. Just make the struct, create an instance, and fread() it in.

Check that the size of the DIB header (that first field in this struct) is 40. It could be other values and if so, once again, display an error message that we do not support the file format, and exit.

If our program gets to this point, we have a BMP that we can read. Let us ensure that the pixel data is encoded in 24-bit RGB by testing the bits per pixel field of our DIB header against 24. If it is any other value, display an error message that we don’t support the file format, and exit.

Now, we can be confident our file has the data we understand and can work with.

Use fseek() to move the file pointer to the location specified in the original BMP header as the offset of the start of the pixel array. We are now ready to process the individual pixels.

The Pixel Array

Each pixel is three 8-bit integers, representing blue, green, and red intensity on a scale of 0-255. Create a pixel struct with the appropriate fields:

Offset	length	description
0	1	Blue intensity
1	1	Green intensity
2	1	Red intensity

The pixels are laid out in the file row by row (in upside-down, bottom row of the image first, order, for reasons we don't care about nor will it affect our program).

The number of rows is the height as specified in the DIB header and the number of pixels per row is the width from the DIB header.

Use fread() to read in each pixel, transforming it as specified on the command line, and use fwrite() to write it back to the file. Note that in doing the fread(), the file pointer is advanced, so if you do an fwrite() you'll actually be overwriting the neighboring pixel. To fix this, you can rewind the file backwards using fseek() relative to the current position.

We do it this way so that we do not need to worry about creating a new, valid BMP file. We will just reuse the original one's headers without modification, since our transformations never alter the size of the image or anything else recorded in the header.

There is one major quirk with reading the pixel array. The BMP format requires that a single row of pixels be a multiple of 4 bytes. But each pixel is 3 bytes, and 3 * width is not guaranteed to be a multiple of 4. If that's the case, extra empty padding bytes are used to force the next pixel row to start at an offset that is a multiple of 4. At the end of reading a row, calculate how many padding bytes are present, and use fseek() to skip them, before looping back to process the next row.

Inverting an Image

To invert a pixel, simply perform the bitwise-NOT (~ in C) to each color in the pixel, and write the new one out. Black becomes white, and white becomes black. Other normal colors often end up getting a neon-like treatment.

This should be your first transformation as it is simple and allows you to do two sanity checks on your resultant image. First, the operation is its own inverse, so if you run it twice on the same program you must get the original image back. Second, there should be no weird pixels, so if you started with an all-white image, the result should be an all-black one, with no other colors.

Converting an Image to Grayscale

In the RGB model of color, the various shades of gray all have the exact same value for red, green, and blue. We need to take the color values we actually have and replace them with a color whose component values are equal. We could do this a variety of ways, such as by averaging the colors, or just picking one of the three and duplicating it to the other two, but people have researched the question to get a better conversion than any of these naïve approaches.

First, take the RGB values and "normalize" them to be a floating point value in the range [0,1] by dividing by the maximum intensity (255) being careful to watch your types.

Next, we use an equation that emphasizes how our eyes are much better at seeing green than blue to get a new value, we call Y built from the RGB values we computed above.

```
Y = 0.2126 * R + 0.7152 * G + 0.0722 * B
```

Where B, G, and R are the pixel values you read from the file.

We then do a transformation on it to make it work better for the RGB color space (again don't worry about the reasons):

$$B = G = R = \begin{cases} 12.92\ Y, & Y \le 0.0031308 \\ 1.055\ Y^{1/2.4} - 0.055, & Y > 0.0031308 \end{cases}$$

Where B, G, and R are the new pixel color values for you to overwrite in your file. These will still be in the range [0, 1] so multiply by 255 and cast to a char to get them in the form we need.

Hints and Requirements

- We need to treat these files as *binary files* rather than *text files*. Make sure to open the file correctly, and to use fread and fwrite for I/O.

- To make sure no padding (extra space inserted to keep data aligned in memory) in the structures is performed, add:

```
#pragma pack(1)
```

  To the top of your program, outside of any function

- Please use structures to represent the two headers and each pixel rather than a bunch of disjoint variables. Do your fread() and fwrite() with a whole structure at once. (That is, read or write an entire header or pixel in one file operation.)

- pow() from <math.h> performs exponentiation. To use math functions in our program we will have to tell the compiler we want to use the math library. Do so via the -lm option (that's a lowercase L followed by a lowercase M) like:

```
gcc bmp_edit.c -lm
```

- We provide you with two images on thoth you can use to test your program on. Obtain them by doing:

```
cp ~wahn/public/cs449/*.bmp .
```

  The trailing dot represents the current directory you are in.

- If you want to make your own files, mspaint in Windows can be used, just make sure that you save it as BMP and select 24-bit color from the dropdown menu. If you don't, our program won't be able to use it.
- If you're tired of copying the files back and forth to see them, you can copy your files to your public/html directory (make the html directory with "mkdir html" if it doesn't exist) and then you can access the files from a web browser by going to www.pitt.edu/~USERNAME/FILE.BMP where USERNAME is your pitt username and FILE.BMP is the name of the file. Please don't put any source code in that directory.
- If you really want to read more about BMP or Grayscale, Wikipedia has some decent articles, but I promise the above is all you need to complete the project.
    - https://en.wikipedia.org/wiki/BMP_file_format#File_structure
    - https://en.wikipedia.org/wiki/Grayscale#Colorimetric_.28luminance-preserving.29_conversion_to_grayscale

Environment

Ensure that your program builds and runs on thoth.cs.pitt.edu as that will be where we are testing.

Submission

When you're done, create a gzipped tarball (as we did in the first lab) of your commented source files and compiled executables.

Copy your archive to the submission directory:

```
~wahn/submit/449/RECITATION_NUMBER
```

*Make sure you name the file with your **username**, and that you have your name in the comments of your source file.*

Note that this directory is insert-only, you may not delete or modify your submissions once in the directory. If you've made a mistake before the deadline, resubmit with a number suffix like `USERNAME_project1_2.tar.gz`

The highest numbered file before the deadline will be the one that is graded, however for simplicity, please make sure you've done all the work and included all necessary files before you submit