Lab 6 – Arrays, Pointers, and Structures

Objectives

In this lab, each student is to write a program called **`prog6.c`** that allows the user to operate on bitmap images. The student should gain and understanding of:

- **Use of Dynamic Memory Allocation**
- **Manipulation of Two-Dimensional Arrays**
- **Manipulation of Structures**
- **Use of Pointers to Structures**
- **Manipulation of Files**
- **Use of Command Line Arguments**

Input

The program should receive as command line arguments the name of an input file, the name of an output file (which must be different), and three numbers from **-200** to **+200**, e.g.

```
prog6 Tiger.bmp TigerOut.bmp +63 -17 +0
```

The input file is to be a 24-bit color bitmap image of the windows **`.bmp`** format.

A 24-bit color .bmp file has two parts to its header describing the image file. The first header has the following structure:

```
struct HEADER
{  unsigned short int Type;     /* Magic identifier       */
   unsigned int Size;           /* File size in bytes     */
   unsigned short int Reserved1, Reserved2;
   unsigned int Offset;         /* Offset to data (in B)  */
} Header;                       /* -- 14 Bytes --         */
```

The second part of the header has the following structure:

```
struct INFOHEADER
{  unsigned int Size;           /* Header size in bytes    */
   int Width, Height;           /* Width / Height of image */
   unsigned short int Planes;   /* Number of colour planes */
   unsigned short int Bits;     /* Bits per pixel          */
   unsigned int Compression;    /* Compression type        */
   unsigned int ImageSize;      /* Image size in bytes     */
   int xResolution, yResolution;/* Pixels per meter        */
   unsigned int Colors;         /* Number of colors        */
   unsigned int ImportantColors;/* Important colors        */
} InfoHeader;                   /* -- 40 Bytes --          */
```

The actual image information follows as groups of three bytes representing the color of each pixel in RGB format. The pixels are stored as rows of columns just as a two dimensional matrix is stored in C. RGB format specifies the intensity of Red as the first byte, Green as the second byte, and Blue as the third byte. For example, the three bytes **0x00FF00** would represent Red = 0, Green = 255, and Blue = 0 which would be Green. A group of **0x802080** would be Red = 128, Green = 32, and Blue = 128 which is a purple. (White is **0xFFFFFF** and black is **0x000000**.)

You are to read in the pixel data of the image and store it in a dynamically allocated two-dimensional array of pixels where each pixel is stored as the following structure:

```
struct PIXEL  { unsigned char Red, Green, Blue; };
```

Functionality

Your program should create two separate output files. The first output file should use a second derivative filter as shown below which will serve as a kind of “edge detector” for the image.

```
char Matrix[3][3] =
{ {  0, -1,  0 },
  { -1,  4, -1 },
  {  0, -1,  0 }
};
```

This matrix is to be used as a filter on each pixel of the image to produce new pixels.

For example, consider an image that has the following data:

	Col 0	Col 1	Col 2	Col 3	Col 4 ...
Row 0	(0,0,0)	(0,2,0)	(3,2,1)	(1,0,0)	(0,0,1)
Row 1	(0,0,0)	(0,2,0)	(1,2,3)	(1,0,0)	(0,0,1)
Row 2	(5,0,0)	(0,4,0)	(7,2,4)	(9,0,2)	(8,0,1)
Row 3	(0,3,0)	(0,1,0)	(8,2,5)	(1,0,1)	(9,0,1)

...

The **Row 2 - Col 3** pixel **(9,0,2)** would be transformed by summing the product of the overlaid filter at pixel **[2][3]**. That is, you would calculate the new value for pixel **[2][3]** as the sum of **Matrix[0][0]** multiplied by **pixel[1][2]** and **Matrix[0][1]** multiplied by **pixel[1][3]** and **Matrix[0][2]** multiplied by **pixel[1][4]** and **Matrix[1][0]** multiplied by **pixel[2][2]** and **Matrix[1][1]** multiplied by **pixel [2][3]**, etc...

	Col 0	Col 1	Col 2	Col 3	Col 4 ...
Row 0	(0,0,0)	(0,2,0)	(3,2,1)	(1,0,0)	(0,0,1)
Row 1	(0,0,0)	(0,2,0)	(1,2,3)	(1,0,0)	(0,0,1)
Row 2	(5,0,0)	(0,4,0)	(7,2,4)	(9,0,2)	(8,0,1)
Row 3	(0,3,0)	(0,1,0)	(8,2,5)	(1,0,1)	(9,0,1)

The matrix factor is to be multiplied by each of the RGB components. So the new **[2][3]** pixel would be:

```
   [(0) x(1,2,3)]   + [(-1) x (1,0,0)] + [(0) x (0,0,1)]
+ [(-1) x (7,2,4)]  + [(4) x (9,0,2)]  + [(-1) x (8,0,1)]
+ [(0) x (8,2,5)]   + [(-1) x (1,0,1)] + [(0) x (9,0,1)]
   = (-1 -7 + 36 -8 -1, -2 + 0, -4 + 8 -1 -1)
   = (19, -2, 2)
```

The second output file should take the same input image and lighten or darken each primary color using the three numbers given (for R, G, and B). For example, in the given command line example, the program should increase the Red shade of each pixel by 63. (***Note, the new color must not be less than 0 or greater than 255***.)

Output

The program should create two new files. Both files should have the exact same header information along with the new pixel values. The name of the two files should be the output filename appended with either **`(edge)`** or **`(shade)`** along with the **`.bmp`** extension. For the given example, **`Tiger(edge).bmp`** and **`Tiger(shade).bmp`**.

Further Considerations

The program should be structured neatly, easily readable, and well commented. Code should be modularized with functions, logically structured, and written to perform efficiently as possible. Furthermore, variable and function names should be such that the software is as "self-commenting" as possible.

The **`main`** function should mainly contain only functions called to complete the separate tasks given.

Creation and Submission

Each individual student must complete their own program. Copying other students' code will be tested for and will not be tolerated.

Use the following line to compile your program

```
gcc -Wall -g prog6.c -o prog6
```

The code you submit must compile using the **`–Wall`** flag and no compiler errors or warnings should be printed. To receive credit for this assignment the code must compile and at a minimum perform some required function.

Code that does not compile or crashes before performing some required function will not be accepted or graded. **All students must do a final check on one of the CES Ubuntu machines to verify that `gcc` using Ubuntu shows no warning messages before submitting the project.**

The project must be submitted to Canvas before midnight on **Monday, April 15th**.