ECE 2220: System Programming Concepts **Fall 2017**
Machine Problem 5 **Due: 11:59 pm, Thursday, October 26**

The goal of this machine problem is to learn to use dynamic memory, two-dimensional arrays, structures, and bitmap image files. In this lab, each student is to write a program called `lab5.c` that allows the user to process a bitmap image file with a transformation matrix to detect edges.

Input

The program must accept three command line arguments: the conversion method, the name of the input file, and the output file.

`./lab5 command input.bmp output.bmp`

Your program must verify that the name of the output file is different than the name of the input file. The command must be one of the following strings: {**`trunc`**, **`center`**, **`mag`**, **`scale`**}. Your program must verify the command is valid. If the input is invalid for any reason, your program must print a message that includes the input format and the possible commands. The input file is 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)  */
};                              /* -- 14 Bytes --         */
```

The **`Type`** must be **`0x4D42`** to indicate the file is a bitmap file (i.e., the letters MB for BitMap). 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        */
};                              /* -- 40 Bytes --          */
```

While there are multiple different formats for the **`InfoHeader`**, we will consider only the one for which the **`Size`** is exactly 40. We also will be able to handle only files in which **`Planes`** is 1, **`Bits`** is 24, and **`Compression`** is 0. An example program, **`parsebmp.c`**, is provided which reads a bitmap file and verifies that the critical fields of the structures are correct.

The 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 **Blue** as the first byte, **Green** as the second byte, and **Red** as the third byte. For example, the three bytes **0x00FF00** represent **Red** = 0, **Green** = 255, and **Blue** = 0 which is just the color green. A group of **0x802080** represents **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 Blue, Green, Red;
};
```

Your program is to 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 a new pixel. (See below for an exception for the pixels in the first and last rows and columns.) For example, consider an image that has the following data stored in a 2D matrix called **pixel[][]**:

	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 pixel in row 2 and column 3 has value **(9,0,2)**. This pixel is transformed by overlaying the **Matrix[1][1]** at position **pixel[2][3]**, multiplying each pixel color by the corresponding value in the matrix, and summing the results, handing each color independently. The figure below shows the nine pixel values that are utilized in transforming the pixel at position **pixel[2][3]**. In this example, the calculation for the new value for **pixel_transform[2][3]** is the sum of **Matrix[0][0]** multiplied by **pixel[1][2]** plus **Matrix[0][1]** multiplied by **pixel[1][3]** plus **Matrix[0][2]** multiplied by **pixel[1][4]** plus **Matrix[1][0]** multiplied by **pixel[2][2]** plus **Matrix[1][1]** multiplied by **pixel[2][3]**, etc. for the remaining four positions. Each of the three RGB components is handled independently by multiplying the matrix value with each component.

	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 new value of the pixel for location **[2][3]** is stored in a separate dynamically allocated two-dimensional array as **pixel_transform[2][3]** and is equal to:

```
   [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, -4 + 8 -1 -1)
= (19, -2, 2)
```

The pixels in the first and last rows and columns are not modified since the matrix cannot correctly overlay the pixels in these locations. Make the output values for these pixels to be equal to the input values.

As an alternative example, consider the same operation above expressed as an equation. Let the function $M(i,j)$for $i,j = -1, 0, 1$ represent the C **char Matrix[][]**, where $M(-1,-1)$ is the same as **Matrix[0][0]**, and $M(1,1)$ is the same as **Matrix[2][2]**. That is, the index values have been shifted from 0, 1, 2 to -1, 0, and 1 to make writing the equation easier. Likewise, consider one of the colors at a pixel location $P(x,y)$. That is, the function P corresponds to one of the three members in the structure **struct Pixel** for the location **pixel[x][y]**. Here, *x* and *y* are the row and column number of the color in the pixel. Then we can write the transformation for *one* of the colors as:

$$Q(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} M(i,j) \times P(x+i, y+j)$$

There is a separate calculation for each of the three colors that make up a pixel. Notice that $Q(x,y)$ represents the extent to which the color value at position (x,y) differs from the values near position (x,y). If $Q(x,y)$ is zero (or close to zero) it means that the nearby colors are about the same and there is no edge. However, if the magnitude of $Q(x,y)$ is large then the nearby colors are very different and it is likely that the pixel at (x,y) is on an edge. Thus, an edge is detected at a pixel location when one or more of the colors have a large magnitude with this *differentiator* calculation. To identify the edges, we create the final new pixel value to show the intensity of the edge. The value of $Q(x,y)$ must be converted back into an 8-bit unsigned char.

One problem with this calculation is the type conversion that occurs because the matrix is defined as a signed 8-bit number (i.e., **char Matrix[3][3]**) while each member in a pixel is an intensity defined as an 8-bit unsigned number (i.e., **struct Pixel** contains three **unsigned char**). The C language converts all the types to 32-bit signed integers, performs the calculation, and then makes a final conversion back to the type of the variable that holds the result. Notice that $P(x,y)$ has a value from 0 to 255 (since it is an unsigned char and the number represents the intensity of that color). With our definition of $M(i,j)$, the value of $Q(x,y)$ is an integer with a range from –1022 to 1020.

Conversion Method

You are to implement four options for calculating the final value for a color. The first command line argument corresponds to one of the four methods in the following list. For all the approaches $Q(x,y)$ is calculated with the C-type **int**. For example, you might use

```
int result;
unsigned char new_pixel_color;
```

where **result** holds the calculation of the new color for a pixel, and **new_pixel_color** is the conversion from the signed integer result back to an 8-bit unsigned pixel value (with value 0 to 255).

1. **Truncation**: simply use the lowest 8 bits of the result. Because **`result`** is a 2s-complement number, if the answer is a negative number, the value of **`new_pixel_color`** has an unpredictable value since C simply takes the least significant 8 bits and does not do any conversion with the sign bit.

```
new_pixel_color = result;
```

2. **Center and clip**: Notice that the result of our matrix calculation is to approximate a differentiation of the colors near a pixel. If all of the colors near a pixel are equal, then the value of result is 0. The larger the value of the result, the more "difference" there is between the pixel at the center of the calculation and those pixels that are nearby. Center the result at 128 (i.e., add 128 to the result), and clip any values still less than 0 to 0 and greater than 255 to 255.

```
result += 128;
if (result < 0) result = 0;
if (result > 255) result = 255;
new_pixel_color = result;
```

3. **Magnitude and clip**: Since the sign of the result in not as important as its magnitude, first take the absolute value before clipping.

```
result = abs(result);
if (result > 255) result = 255;
new_pixel_color = result;
```

4. **Scale and center**: By dividing the result by 8 we ensure that the answer is within the range of an 8-bit signed integer with values between -128 and 127. Now if we add 128, the value is between 0 and 255.

```
result /= 8;
result += 128;
new_pixel_color = result;
```

Output

The program should write the two headers and then each new pixel to the output file to create a valid **`.bmp`** file. None of the information in the headers should change. The **`parsebmp.c`** program can be used to check the output file, and any graphics program that can display **`.bmp`** files must be able to display your output file. The standard I/O library **`<stdio.h>`** contains the **`fread()`** and **`fwrite()`** functions which are used to read from and write to binary files.

Notes

1. You compile your code using:

```
gcc -Wall -g lab5.c -o lab5
```

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 your code must load in a bmp file and at a minimum create the output file with the first transformation. Code that does not compile or fails to pass the minimum tests will not be accepted or graded.

2. Use **`valgrind`** to verify that your code is correctly using **`malloc`** and **`free`** and the all memory accesses are to valid locations.

3. Submit your file **`lab5.c`** to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE222-1,#5. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes.

See the ECE 222 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student. See the course syllabus for additional policies.