15-123 Effective Programming in C and UNIX
Lab 6 – Image Manipulation with BMP Images
Due Date: Sunday April 3rd, 2011 by 11:59pm

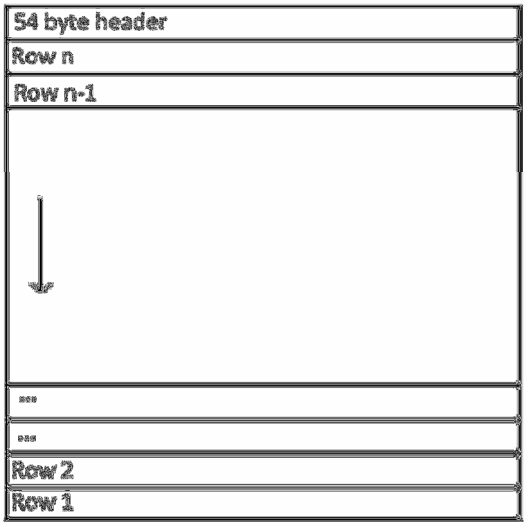
The Assignment Summary:

In this assignment we are planning to manipulate BMP images using some widely used image manipulation techniques. In particular, we will be using techniques such as quantization, rotation and blurring of a given image using image bit manipulation techniques. First we need to understand the bitmap file format.

The BMP file format:

BMP file format is one of the most popular uncompressed file formats for images. BMP file format developed by Microsoft stores an n by m image as a file of size *3·n·m + 54+ some extra bytes* bytes. The 54 bytes is allocated for header information and each pixel takes 3 bytes for Red, Green and Blue (RGB) colors and if the image width is not a multiple of 4, image may also have some padding bytes (why?). Each pixel in BMP image uses 3 bytes (RGB) or 24-bits and therefore 2^{24} possible colors can be represented in a BMP image.
Each R, G, or B byte can store unsigned numbers in the range of 0-255. Hence the color red is (255, 0, 0) and color green is (0, 255, 0). The color (128, 0, 0) makes dark red, and (255, 0, 255) makes purple since purple is red + blue.

Typical view of a bit map image as follows where last image row is represented first.



Although we think of an image as a rectangle, the actual bytes are stored in a 1D array as follows

Header – 54 bytes	R G B R G B	some extra bytes may be added at end of each row	R G B

The left most RGB is the lower right pixel of the image and last RGB is the upper left pixel of the image. As we read the image, we will first store the header information (54 bytes) in a buffer and then read the image bytes (3 for each pixel) into an array.

Part I

In this part of the assignment you are simply trying to read, store and write a BMP image file. The attributes of the BMP file can be stored in the following struct.

```
typedef struct {
  char* imagebytes;        /* an array of bytes. 3 bytes (RGB) for each pixel  */
  char header[54];         /* header information buffer */
  int height, width, size, padding;  /* other image attributes */
} bmp;
```

You are to complete the following functions. This part gives a good exercise on reading and writing an image. You will have to make sure, you write the header back, all image bytes (RGB) and padding bytes (when applicable) to the output file. You should be able to open the output file using any image viewer.

```
//Read the image into mybmp. You must allocate just enough memory to hold the
image bytes. This can be done by first reading the header and figuring out
what the size of the image is. Note that we pass a FILE* since file was
already opened in calling program.
void readImage(FILE* fp, bmp* mybmp);

// writes the image as is to outfile. The name of the file is passed to the
function
void writeImage(char* outfile, bmp* mybmp);
```

You also need to complete the main program so that when the command

```
$ ./a.out file1 file2
```

is given, it will simply read BMP file1 and write to BMP file2. When you open file2 using an image viewer, you should see the same image as in file 1. This will allow us to make sure; we can read and write BMP images w/o losing any data.

Part II

Quantization of an Image

In this problem, you will implement a function that achieves a quantization on an image. Quantization reduces the total number of colors used in an image. You can see an example in Figure 1.

Figure 1: A sporty coupe with quantization level 0 (left) and level 7 (right).

Given an ordinary image of size w x h and a quantization level q between 0 and 7, inclusive, for each pixel in the image, take each color component (red, green and blue) and clear the lowest q bits. For example, suppose the color components for a pixel are given by the bytes

RED	GREEN	BLUE
01101011	10111110	11010111

If the quantization level is 5, then the resulting pixel should have the following color components (note how the lower 5 bits are all cleared to 0):

RED	GREEN	BLUE
01100000	10100000	11000000

Note that an image processed with a quantization level of 0 should not change.

In this part you will complete the function

int quantize(char* infile, bmp* mybmp, char* outfile, int qlevel);

This function will take a valid BMP image, given by infile, read the file into mybmp using readImage developed in Part I, apply the quantization level and write a valid BMP image back to file given by outfile.

You also need to complete the main program so that when the command (for example)

```
$ ./a.out -q 7 file1 file2
```

(The level can be anywhere from 0 to 7)

is given, it will simply read BMP file1, apply quantization level 7 and write the BMP to file2. When you open file2 using an image viewer, you should see the quantized image.

Part III

Blurring an Image

In this problem, you will write a function to blur an image. This is done by using a "mask". A mask is an n by n array of non-negative integers representing weights. (Note: although weights can be 0, the weight in the center position of the mask cannot be zero.) For our purposes, n must be odd. The origin of the mask is its center position. For each pixel in the input image, think of the mask as being placed on top of the image so its origin is on the pixel we wish to alter. The original intensity value of each pixel under the mask is multiplied by the corresponding value in the mask that covers it. These products are added together and then we divide by the total of the weights in the mask to get the new intensity of the mask. Always use the original values for each pixel for each mask calculation, not the new values you compute as you process the image.
For example, refer to Figure 3, which shows a 3 by 3 mask and an image that we want to blur.

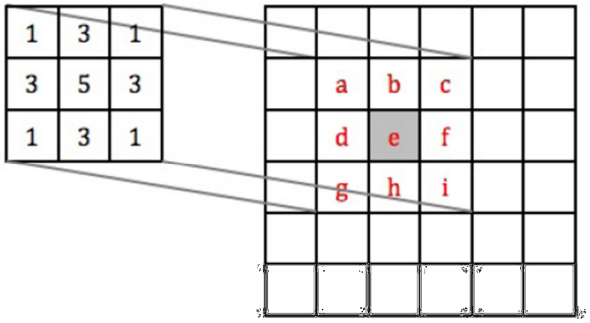


Figure 3: Overlay the 3 X 3 mask over the image so it is centered on pixel e to compute the new value for pixel e.

Suppose we want to compute the new intensity value for pixel e. Imagine overlaying the mask so its center position is on e. We would compute the new intensity for the pixel e as:
(a + 3b + c + 3d + 5e + 3f + g + 3h + i) / 21

This calculation is done three times for each pixel, once for its red channel, once for its green channel, and once for its blue channel.

Note that sometimes when you center the mask over a pixel you want to blur, the mask will hang over the edge of the image. In this case, compute the weighted sum of only those pixels the mask covers. Remember that you must divide by the sum of only those weights that you use from the mask. For the example shown in Figure 4, the new intensity for the pixel e is given by:
(3b + c + 5e + 3f + 3h + i) / 16

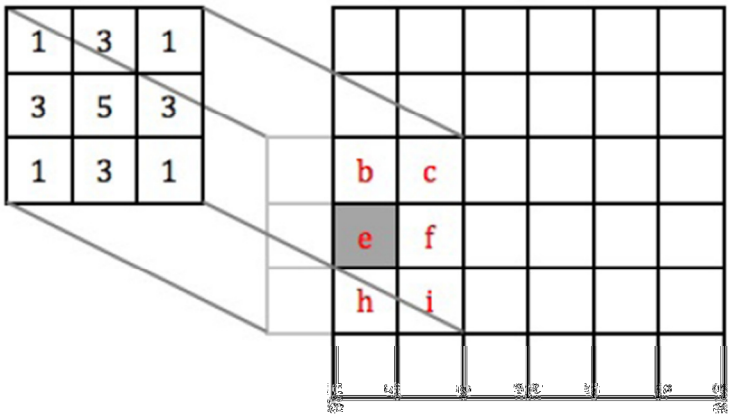


Figure 4: If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

Figure 5 shows a sample image blurred using the following masks:

$$\begin{matrix} 1 & 3 & 1 \\ 3 & 5 & 3 \\ 1 & 3 & 1 \end{matrix} \qquad \begin{matrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 4 & 3 & 2 \\ 3 & 4 & 5 & 4 & 3 \\ 2 & 3 & 4 & 3 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{matrix}$$

Figure 5: Andrew Carnegie: original image (left), blurred with a 3×3 mask (middle), and a 5×5 mask (right). See text for mask values.

In this part you will complete the function

int blurring(char* imagein, char* imageout, int masklevel);

Where mask level is either 1 or 3 or 5. If it is 1, no changes to the image are made, if it is 3, then the 3 by 3 mask is applied and if it is 5, then 5 by 5 mask is applied.

You also need to complete the main program so that when the command

```
$ ./a.out -b <1/3/5> file1 file2
```

is given, it will simply read BMP file1 and apply mask 1 or 3 or 5 and write to BMP file2. When you open file2 using an image viewer, you should see images as shown in Figure 5.

Part IV

Rotation Effect

In this problem, you will create a rotation effect on an image. Your task here is to implement a function that takes as input an image of size wx h and creates a "Rotation" image of size 2w x 2h that contains the same image repeated four times, the top right image containing the original image, the top left containing the original image rotated 90 degrees counterclockwise, the bottom left containing the original image rotated 180 degrees, and the bottom right containing the original image rotated 90 degrees clockwise. **Note that the original image must have the same width and height (square image) in order to do the "rotation" effect.** A sample image is shown in Figure 2.

Figure 2: Original image (left); Image after "rotation effect" (right)

In this part you will complete the function

int rotation(char* infile, bmp*mybmp, char* outfile);

This function will take a valid BMP image, given by infile, create the rotation as described and shown above and write a valid BMP image back to file outfile.

You also need to complete the main program so that when the command

```
$ ./a.out -r file1 file2
```

is given, it will simply read BMP file1 and rotate as defined above and write to BMP file2. When you open file2 using an image viewer, you should see images as shown in Figure 2.

Suggested Data Structure:

We will read the image file and maintain image attributes in a struct as follows

```
typedef struct {
  char** rows;        /* an array of rows */
  char header[54];   /* header information buffer */
  int height, width, size, padding;  /* other image attributes */
} BITMAP;
```

The BITMAP struct will help us perform the necessary operations on the image. First, you need to allocate memory for rows (an array that holds rows of the image)

The structure of the 54 byte header of a BMP file is as follows. The first 14 bytes are allocated for

```
typedef struct {
        unsigned short int type; /* BMP type identifier */
        unsigned int size; /* size of the file in bytes*/
        unsigned short int reserved1, reserved2;
        unsigned int offset; /* starting address of the byte */
} HEADER;
```

The next 40 bytes are reserved for a structure as follows.

```
typedef struct {
        unsigned int size; /* Header size in bytes */
        int width,height; /* Width and height in pixels */
        unsigned short int planes; /* Number of color 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 ncolors; /* Number of colors */
        unsigned int importantcolors; /* Important colors */
} INFOHEADER;
```

Part V (Optional) – In this part of the assignment you are to implement a known algorithm that allows us to hide a message inside an image. ***Steganography*** *is the art and science of writing hidden messages in such a way that no-one, apart from the sender and intended recipient, suspects the existence of the message, a form of security through obscurity[source: Wikipedia]*

In this part we will only test the following two images.

```
//Apply the following algorithm to an image. Remove all but last two bits of
each color component. Make the resulting image 85 times brighter
void coolAlgorithm(BITMAP* mybmp);
```

This algorithm will take this image

and convert to this image when you apply the above algorithm. That is, remove all but last 2 bits of each color component. Then multiply (or bit shift) the color component by 85.

[source: Wikipedia]

You also need to complete the main program to handle "-r" flags

```
$ ./lab6 -c file1 file2 // tree is converted to a cat -really cool!!
```

Style Points

Proper use of functions (passing/returning arguments etc), commenting, style, handin, compile, indenting, and efficiency of the solution are worth an additional 10 points. Also 3 points will be allocated for selflog.txt file.

Extra Credit: There is some opportunity to get extra credit in this assignment. TA's will consider giving up to +5 points for really cool algorithms. You should write why it is a really cool algorithm in a readme.txt file in the handin folder. For more advanced users, you can also think of how to implement an "edge detection" algorithm. Edges can be detected, in theory, by observing changes in color intensity, variations in scene illumination, change in material properties etc. Your algorithm can generalize to images say, passport images of people, so you can mark the outline. If you are completing this part, give a link to an image folder you tested with so TA's can try things out. Mark the outline with red pixels.

Commenting your code: Please comment your code to show what you are doing. The least amount of comments is a short description at the beginning of each function that explains what it does, what parameters it takes, and what the expected output or return value is.

Downloading your code: You can download some starter code (although you have to complete most of it) and sample images from **/afs/andrew.cmu.edu/course/15/123/downloads/lab6**

Submitting your code: submit **main.c to /afs/andrew/course/15/123/handin/lab6/yourid**. DO NOT submit any image files you are using to test the program.