EECS402, Winter 2013, Project 3

Overview:
Working with, and modifying pictures on a computer is big business. Images might be modified by performing image sharpening algorithms in order to better make out a criminal's face from a fuzzy surveillance camera photo. A company might charge a small fee for removing the "red eye" problem that flash photography suffers from, so that you can make better prints from your photos. Team members located in different cities might "mark up" an image during teleconferences in order to share their thoughts on images or graphs. And sometimes, you just want to be able to put conversation bubbles on a photo to add some humor.

Each of these situations requires knowledge of how computers deal with imagery. This project will introduce you to a straight forward image format, and allow you to modify an image in a few specific ways.

Due Date and Submitting:
This project is due on **Friday, March 1 at 5:30pm**. Late submissions will not be accepted for any reason.

For this project, you must submit several files. You must submit each header file (with extension .h) and each source file (with extension .cpp) that you create in implementing the project. In addition, you must submit a valid UNIX Makefile, that will allow your project to be built using the command "make" resulting in an executable file named "proj3.exe". When submitting your project, all these files must be in the single tar file emailed. Also, your Makefile must have a target named clean that removes all your .o files and your executable.

Background - .ppm Images:
Since you will be reading and writing images, you need some background on how images work. For this project, we will use a relatively simple image format, called PPM imagery. These images, unlike most other formats, are stored in an ASCII text file, which you are already familiar with. More complicated image formats (like .gif and .jpg) are stored in a binary file and use sophisticated compression algorithms to make the file size smaller. A .ppm image can contain the exact same image as a .gif or .jpg, but it would likely be significantly larger in file size. Since you already know how to read and write text files, the only additional information you need is the format of the .ppm file.

In general, an image is made up of many individual pixels, in a two-dimensional array with a certain number of rows and columns. Every pixel in the image simply contains a specific color. When these pixels are displayed on the screen you see them as an image. Images can be either a grayscale image, or a color image. In a grayscale image, each pixel simply contains a "brightness" value, or shade of gray. In a color image, each pixel must be made up of multiple values, since there is no way to store different colors in one single value.

Different colors can be defined by three integer values, which contain the amounts of red, green, and blue that make up the color. Any color can be made by altering these three values. For example, a value of "10 0 0" will result in a fully red color. Similary, "0 10 0" results in fully green, and "0 0 10" results in fully blue. "3 3 3" would be a dark gray since all three colors are represented evenly. "8 8 8" would be a much brighter gray, since still all three colors are represented evenly, but at a much higher intensity. Since red and blue can be combined to generate purple, the value "10 0 10" would result in this mixture.

Most image types start with two special characters, which are referred to as that image type's "magic number." A computer can determine which type of image it is based on the value of these first two characters. For a .ppm image, the magic number is "P3".

Since a 100 pixel image may be an image of 25 rows and 4 columns, or 10 rows and 10 columns (or any other such combination) you need to know the specific size of the image. Therefore, after the magic number, the next two elements of the .ppm file are the width of the image, followed by the height of the image. Obviously, both of these values should be integers, since they both are in units of "number of pixels".

The next value is also an integer, and is simply the maximum value in the color descriptions. For example, in the above examples, 10 is the maximum value. For this project, you will use 255 as the maximum number. With a maximum of 10, you are only allowed 10 shades of gray, and 10^3 unique colors which would not allow you to generate a very photographic looking image, but if your maximum value is 255, you could get a much wider range of colors (255^3).

The only thing left is a description of every pixel in the image. The pixel in the upper left corner of the image comes first. The rest of the first row follows, and then the first pixel of the second row comes after that. This pattern continues until every pixel has been described (in other words, there should be rows*cols color values in the .ppm file). As mentioned above, each pixel is described with three integers (red, green, blue).

A very very small image of a red square on a blue background would be stored in a .ppm file as follows:

```
P3
4 4
255
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   255 0 0   255 0 0   0 0 255
0 0 255   255 0 0   255 0 0   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
```

Once you create these images, you can view them many ways. There are many freely available programs that will display PPM images directly (I often use one called "IrfanView" on Windows (should be able to download this from download.com) and either "xv" or ImageMagick's "display" on Linux).

Another alternative is to convert the image to a JPEG image, which will allow you to display the image via a web browser. One way to convert a PPM to JPG is to use the Linux command "cjpeg" like this:

```
% cjpeg inFile.ppm > outFile.jpg
```

The "%" character shown is just meant to be the Linux prompt – it is not part of the command.

Detailed Description:
For this project, you are only required to implement a few algorithms to modify an image. However, after completing the project, you will be able to add any number of your own algorithms to modify imagery in any number of ways.

Following are descriptions of the algorithms you are required to implement. First, you will need to allow for an image to be annotated with rectangles. Rectangle outlines may be placed on an image to draw attention to a specific area, or filled rectangles may be placed on an image to block out a specific area. Both of these operations will be supported in this project.

Second, and more interestingly, an image may be annotated with a pattern. A pattern, while rectangular overall, contains a description of a shape that is to be used to annotate an image. A pattern consists of a rectangle of zeros and ones. When a pattern is placed over an image, the values in the pattern each fall over a specific pixel. A one in a pattern indicates that the pixel under it should be modified to be a certain color, that is specified by the user. A zero in a pattern indicates that the pixel under it should NOT be affected by the pattern. Its original value is left intact. Patterns are contained in text files of the following format: The first value is an integer representing the number of columns in the rectangular pattern. The second value is an integer representing the number of rows in the rectangular pattern. What follows is a collection of zeros and ones that is (rows * columns) in length. For example, here is the contents of a pattern file that defines a pattern of the letter 'T':

```
6 8
1 1 1 1 1 1
1 1 1 1 1 1
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
```

The placement of such patterns on an image will be supported in this project. This capability allows you to annotate an image with any shape you wish, regardless of what it looks like.

The final image modification algorithm you will implement is the insertion of another (presumably smaller) PPM image at a specified location within the image being modified. This insertion simply reads another PPM image from a file and inserts the image contents where the user desires. PPM images are, by definition, rectangular. Since oftentimes, the image you want to insert is not rectangular, you must support a transparency color, such that any pixel in the image to be inserted which is the transparency color does not get inserted into the image. Note that this is very similar to the use of a pattern, described above, except that patterns can only be one color, which inserted images can have as many colors as the PPM allows.

At any stage, you must allow the user to output a PPM image in its current state from the main menu. The user may want to output an image after each annotation made, or just once when all annotations have been performed. Since the option of outputting an image is available on the main menu, this functionality will be supported in this project.

There are examples of all required functionality available in the sample output of the project.

Implementation and Design Information:
All of your global constants must be declared and initialized in a file named "constants.h". This file will not have a corresponding .cpp file, since it will not contain any functions or class definitions. Make sure you put

all your global constants in this file, and avoid magic numbers. Since you now know about dynamic allocation, the image pixels will be allocated using the new operator, using exactly the amount of space required for the image (for example, a smaller image will use less memory than a larger image). Therefore, there is no practical limit to the size of the image allowed.

I would recommend the following classes be developed: ColorClass, ImageClass, PixelLocationClass, and RectangleClass. I'll leave the majority of the design up to you, and remember I will be looking at your design during grading. If you find you want some global functions, you may use them. Each individual class will be contained in a .h and a .cpp file (named with the class name). ALL class member variables MUST be private. Your member functions may be public. Each global function will be contained in a .h and a .cpp file named the same as the function. Do not put multiple global functions in a single file. Remember, when submitting, you must make a tar file containing ALL .h files, .cpp files, and your Makefile. Do not include your .o files or your executable.

I'll give a brief overview of the classes I recommended above, to clarify the purpose of each.

ColorClass is a class to describe the color of a single pixel of an image. As described above, you'll need a value for the amount of red, green, and blue combined to define the color of the pixel. The ImageClass, then, is a collection of pixels (ColorClass objects) that describe an image. The ImageClass will need to have functionality to write and read images to/from files. When developing this functionality, remember that an ImageClass object should write/read image-related fields to/from files, its really each pixel's responsibility to write/read its own color to/from the file. In other words, the ImageClass write/read methods should not write/read *color* values. Instead, the ImageClass should call a member function of the ColorClass to write those values. Always think about this type of thing when designing your project. The PixelLocationClass is just a way of encapsulating a specific row/column and any functionality to operate on that data (that is, its attributes simply include a row value and a column value, and you can refer to a location in the image via *one* PixelLocationClass object, rather than specifying two integers (row and col)). Finally, a RectangleClass needs little description - a class that describes a rectangle to be drawn on an image.

While you have more flexibility in your project design, your menu and the way your project operates should match that shown in the sample output. Do not change the actions associated with menu options, orderings, etc.

A Quick Detail:
The "open" member function of the file stream classes (ifstream, ofstream) take the name of a file in as a parameter of type "c-string", NOT C++ string. You'll have filenames stored as C++ strings, though, so you'll need to convert it to a C-string so the compiler will be happy. Do this using a member function of the string class called "c_str()". For example, your code would look something like this:

```
ifstream inFile;
string fname;
...
inFile.open(fname.c_str());
```

Error Handling:

Since we've talked about error handling for stream input/output, you'll need to ensure you handle potential issues when dealing with input. There are several things that can go wrong during the input/output, and you should consider *all* of those cases.

Here's how to handle errors that come up. For the initial prompt for the main image, if the image can't be loaded, print an error message and allow the program to end. If other files can't be read or written during the program, output a ***descriptive*** error message and continue the program. Do not keep re-prompting as in prior projects. In any case, make sure you print a descriptive error message. Saying "Error found when trying to read magic number - expected P3 but found P5" is far better than just saying "Error reading image" which doesn't describe the error that occurred at all. Make sure you consider all the different things that could go wrong when reading a PPM file, which may or may not be in a proper format.