CS 202
Assignment #8

Purpose:	Learn class inheritance, class implementation, and make utility.
Due:	Tuesday (3/12)
Points:	150

Assignment:

Computer images are stored as a large collections 1's and 0's. More specifically, an image can be thought of as a two-dimensional grid where each cell stores the value of a color for the specific cell or pixel. Larger grid's represent larger or higher resolution images. By adjusting the values at the locations and/or moving the values around the image can be manipulated in various ways such as rotating the image, re-sizing the image, or removing color thus making it a gray-scale (black-and-white) image.

Design and implement a C++ class to perform a series of image manipulations.

- Image Type Class
  The *imageType* class will provide a series of image manipulation functions. The *imageType* class should inherit the bitmap image functions from the *bitmapImage* class (which includes functions for the reading and writing of the formatted BMP image files).

imageType
`-fileName: string`
`-title: string`
`-H_MAX = 10000: static const int`
`-W_MAX = 10000: static const int`
`+imageType(string="", int=0, int=0, string="")`
`+colorToBW(): void`
`+resize(double): void`
`+resize(int, int): void`
`+smooth(): void`
`+brightness(double): void`
`+RGBshift(color): void`
`+rotate(double, int=0): void`
`+crop(int, int, int, int): void`

The header file should include: **`enum color {RED, GREEN, BLUE};`**
You may add additional private functions as needed.

Function Descriptions

The following are more detailed descriptions of the required functions.

- The *imageType()* constructor should also accept a file name string, width, height, and image title string (in that order). The constructor uses default parameters as noted in the UML. The constructor must use the base class constructor (which will verify the images sizes). If a file name is provided, the constructor should ensure that a valid file exists and provide and error if not.

- The *colorToBW()* function should convert a color image to gray-scale (black-and-white). Each pixel, (*red*, *green*, *blue*) should set based on the following formula:

  $$newRed = newGreen = newBlue = \frac{oldRed + oldGreen + oldBlue}{3}$$

  The function should provide an error message if the image is invalid (width or height =0).

- The *resize(int, int)* should resize the bitmap image based on the passed width and height (in that order). This will require a temporary image of the new size. Pixels from the original image will be interpolated and placed into the new temporary image. The ratios can be calculated as follows:

  $$xRatio = \frac{oldWidth}{newWidth} \qquad yRatio = \frac{oldHeight}{newHeight}$$

  For each (***x***,***y***) pixel in the new image the new value should be set from the old image from the location as follows:

  $$\begin{aligned} oldX &= \lfloor newX * xRatio \rfloor \\ oldY &= \lfloor newY * yRatio \rfloor \\ (newX, newY) &= (oldX, oldY) \end{aligned}$$

  Where $\lfloor \, \rfloor$ the is the mathematical floor function (i.e., rounding down). Pay very close attention to the data types since the (***x***,***y***) values are integers and the ratios are doubles. When the temporary image is completed, it should be used to replace the original image and the temporary image deleted. The function should provide an error message if the image is invalid (i.e., width or height are 0). The function should provided an error message if the new width or new height are < 0 or if are > W_MAX or H_MAX. If there is an error, the current image, if any, should not be modified.

- The *resize(double)* should resize the bitmap image based on the passed percentage. The current width and height can be adjusted by the percentage which can then be passed to the *resize(int,int)* function. This function will maintain the aspect ration of the image. Pay close attention to the data types as the calculations will require some static casting.

- The *brightness(double)* function should increase or decrease the brightness of the image by the passed percentage. For each pixel, the red, green, and blue color values should be multiplied by the percentage as follows:

  $$newColorValue = colorValue\left(1.0 + \frac{pct}{100.0}\right)$$

  If the calculated new color value is > 255 it should be set to 255. If the calculated color value is < 0 it should be set to 0. The function should provide an error message if the image is invalid (i.e., width or height are 0). The function should provided an error message if the percentage value is < -100.0 or > +100.0. If there is an error, the current image, if any, should not be modified.

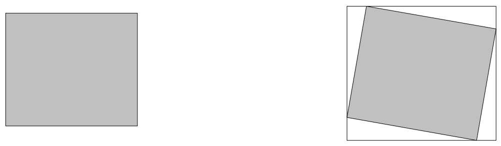
- The *RGBshift(color)* function should shift the image to the passed color (RED, GREEN, or BLUE). This will require increasing the target color value while setting the other color values to zero.

The formula for a red shift is as follows:

$$newRed = (oldRed - oldBlue) + (oldRed - oldGreen)$$

The formula will will need to be adjusted for blue and green. The function should provide an error message if the image is invalid (i.e., width or height are 0). If there is an error, the current image, if any, should not be modified.

- The *rotate(double, int)* function should rotate the image by the number of degrees passed (double). The degrees must range between -180.0 and +180.0 (inclusive). Additionally, a background color may be passed to specific any exposed background (the default value is black). The background color must be between 0 (black) and 16777215 (white). For example, given the below original image (gray) rotated 10°, the new image will expose some background (shown in white for this example).



The calculations will require that the degrees be converted to radians which base be done as follows:

$$radians = \frac{2\ \pi\ degrees}{360}$$

As shown above, the new image size is larger. Thus, the size of the new image must be determined prior to actual rotation. The new size is calculated as follows:

$$\begin{aligned}
x1 &= -oldHeight * \sin(radians) \\
y1 &= oldHeight * \cos(radians) \\
x2 &= oldWidth * \cos(radians) - oldHeight * \sin(radians) \\
y2 &= oldHeight * \cos(radians) + oldWidth * \sin(radians) \\
x3 &= oldWidth * \cos(radians) \\
y3 &= oldWidth * \sin(radians) \\
minx &= min(0, min(x1, min(x2, x3))) \\
miny &= min(0, min(y1, min(y2, y3))) \\
maxx &= max(x1, max(x2, x3)) \\
maxy &= max(y1, max(y2, y3)) \\
newWidth &= maxx - minx \\
newHeight &= maxy - miny
\end{aligned}$$

Pay close attention to the data types as the calculations will require some static casting. Once the new size has been determine, a new temporary image must be created with the provided background color. Then, for every pixel in the new image, check the old image for a value based on the following formulas:

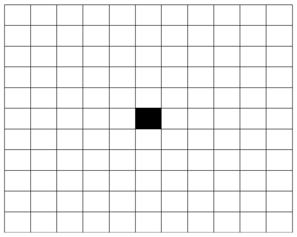
$$\begin{aligned}
sourceX &= (x + minx) * \cos(radians) + (y + miny) * \sin(radians) \\
sourceY &= (y + miny) * \cos(radians) - (x + minx) * \sin(radians)
\end{aligned}$$

Again, pay close attention to the data types as the calculations will require static casting. The *sourceX* and *sourceY* are the coordinates in the source image (original image) for the color value for the (***x,y***) location in the new image. The calculations may generate a point that is not in the original source image. As such, before accessing the (***sourceX, sourceY***) location in the original source image, you must ensure that is it in range. When the temporary new image is completed, it should be used to replace the original image and the temporary image deleted. The function should provide an error message if the image is invalid (i.e., width or height are 0), the rotation percentage is invalid, or the background color is invalid. If there is an error, the current image, if any, should not be modified. Refer to the example execution for the error messages.

- The *smooth()* function will smooth or blur the image. The following steps can be used to smooth the image. The radius should be set to 5 initially.

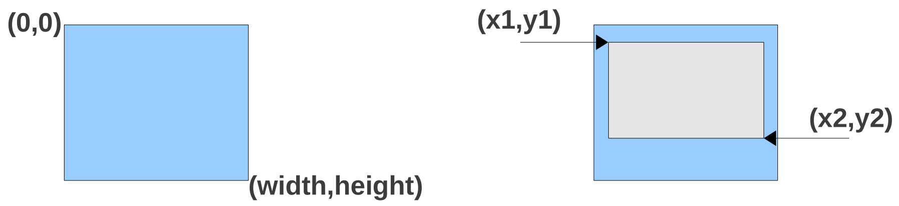
  1. Take the ***radius*** pixels surrounding your pixel and your pixel.
  2. Average the RGB values of all radius ***n*** pixels and stick them in your current pixel location. Note, $n = (2\,radius + 1)^2$
  3. Repeat for all pixels.

  For example, the value for the pixel shown in black should be



  For the image edge values you can keep them the same. Ideally, you would average only the available/accessible pixels. The new, averaged pixels can not be placed into the same picture as the non-averaged picture, or it won't blur correctly. When the temporary new image is completed, it should be used to replace the original image and the temporary image deleted. The function should provide an error message if the image is invalid (i.e., width or height are 0). If there is an error, the current image, if any, should not be modified.

- The *crop(int,int,int,int)* function should crop the image. The first two passed arguments represent the new (***x1,y1***) position for the new upper left corner. The next two passed arguments represent the new (***x2,y2***) position for the lower right corner. For example, given the below image:



  The new, cropped version of the original image is shown in gray. The routine must verify that (***x1,y1***) is greater than (***x2,y2***) and that all points are within range. The function should provide an error message if the image is invalid (i.e., width or height are 0). If there is an error, the current image, if any, should not be modified.

- Bit Map Image Class
  There are many programs that can write information to a simple, device and OS independent bitmap image format. The *bitmapImage* class (**`bitmapImage.h`** and **`bitmapImage.so`**) provides a simple interface for the creation of an uncompressed BMP format bitmap file.

`bitmapImage`
`-width: int`
`-height: int`
`-**pixelData: int`
`-freePixelData()`
`+bitmapImage()`
`+bitmapImage(const char*)`
`+bitmapImage(int, int)`
`+~bitmapImage(): virtual`
`+setSize(int, int): void`
`+loadFromBitmapFile(const string): bool`
`+saveToBitmapFile(const string) const: void`
`+saveToBitmapFile(string, int) const: void`
`+getHeight() const: int`
`+getWidth() const: int`
`+getRed(int color) const: int`
`+getGreen(int color) const: int`
`+getBlue(int color) const: int`
`+getColorForRGB(int, int, int) const: int`
`+getPixel(int, int) const: int`
`+setPixel(int x, int y,int color): void`
`+fillWith(int color): void`
`+fillRect(int, int, int, int, int): void`
`+setPixelRGB(int, int, int, int, int): void`

  The class header and object files will be provided on the class web site.
  Note, this class may be used for future assignments.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 202 for additional information.

Make File:

You will need to develop a make file for future assignments. With the provided make file, you can type

```
make
```

Which will create the *imageType* executable. The makefile is similar to the previous assignments.

Submission:

- Submit a zip file of the program source files, header files, and makefile via the on-line submission by 23:55.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

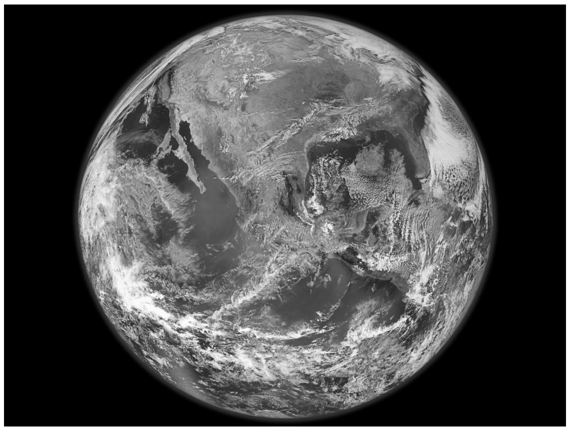
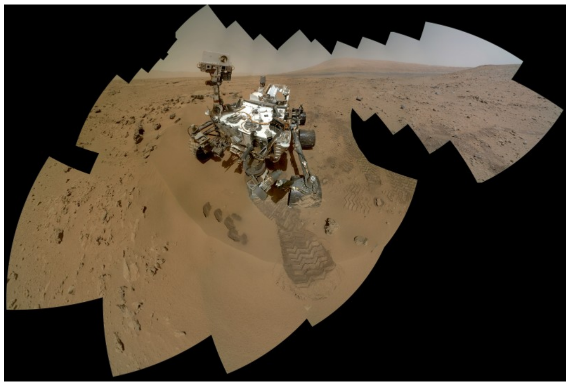
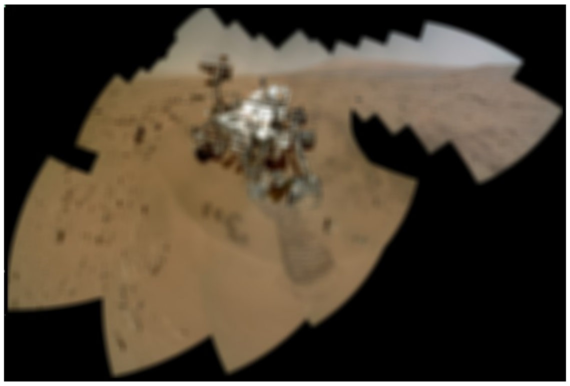
Example Execution:

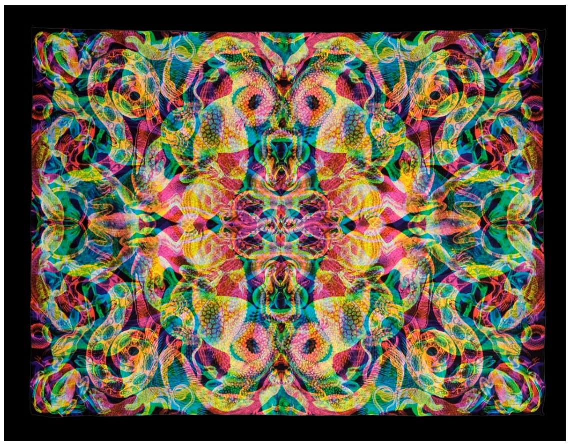
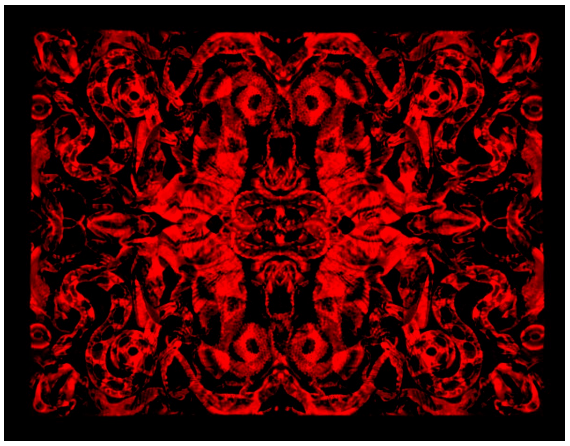
Below is an example program execution for the main.

```
ed-vm% ./main
************************************************************
CS 202 - Assignment #8.
Image Manipulation Program.

Image Test #0, file: image0.bmp
Invalid Bitmap File Format: No such file or directory
Error opening null.txt

Image Test #1 A, file: image1a.bmp
Image Test #1 B, file: image1b.bmp

Image Test #2 A, file: image2.bmp
Error, no image to convert to resize.
Error, invalid re-size parameters.
Error, invalid re-size parameters.
Error, invalid re-size parameters.
Image Test #2 B, file: image2.bmp
Image Test #2 C, file: image2.bmp
Image Test #2 D, file: image2.bmp

Image Test #3, file: image3.bmp
Error, invalid brightness percentage.
Error, invalid brightness percentage.

Image Test #4, file: image4.bmp
Error, no image to convert to color shift.

Image Test #5, file: image5.bmp

Image Test #6, file: image6.bmp

Image Test #7, file: image7.bmp
Error, no image to convert to color shift.
Error, invalid background color.
Error, invalid background color.
Error, invalid rotation percentage.
Error, invalid rotation percentage.

Image Test #8, file: image8.bmp
Error, invalid crop points.
Error, invalid crop parameters.
Error, invalid crop points.
Error, invalid crop points.
Error, invalid crop parameters.
Error, invalid crop parameters.
Image Test #9, file: image9.bmp

ed-vm%
```

Example Output:

Below are some examples the original input and final output images (using the Ubuntu default image viewer).

Original Image		**New Image**
Function: colorToBW()		
`image0.bmp`		**`T0newimage0.bmp`**
		
Function: smooth()		
`image1a.bmp`		**`T1newimage1a.bmp`**
		

Function: RGBshift()		
`image6.bmp`		`T6newRimage6.bmp`
		
Function: colorToBW()		
`image7.bmp`		`T7newAimage7.bmp`