EECS402, Fall 2019, Project 2

Overview:

Images are everywhere, and being able to work with images and colors is not uncommon in programming. For this project, you will use object-oriented programming to develop some simple classes to represent and utilize colors and images. Once completed, this project could be expanded to do more operations, including writing to files for viewing, etc.

There is no user interaction required for this project. Please note: while these specifications are fairly lengthy, it is because I am fully specifying this project as it is our first object-oriented project. The actual implementation of the project should not be especially long or complex. The complete specifications for this project are given below.

Due Date and Submitting:

This project is due on **Monday, October 7, 2019 at 4:00pm**. Early submissions are allowed, with corresponding bonus points according to the policy described in detail in the course syllabus.

For this project, your solution must be fully contained in exactly one file. Your project source code must in one file named exactly "project2.cpp".

Detailed Description:

At this point in the course, you have been given an overview of basic object-oriented principles, and this project will be using the object-oriented property of encapsulation using C++ classes.

Colors are typically described as a combination of some amount of red, green, and blue (referred to as "RGB" values). With different amounts of red, green, and blue, you can create any color. For example, the color bright red would be represented as a full amount of red and no green and no blue, and a dimmer red would be represented as less than a full amount of red with no green and no blue. Similarly, black is no red, no green, and no blue, while white is full red, full green, and full blue. Other colors are obtained by mixing the component colors, so yellow is obtained via full red, full green, and no blue, etc. The component color values must be within a specified range, and the term "full red" (or green or blue) means the maximum allowed color value, which for this project will be 1000, while the minimum allowed color value is 0.

An image is just a bunch of colors arranged in a rectangle. Because each individual color element (called a pixel) is so small, your eyes and brain can interpret the rectangle of tiny pixel colors as an image. Since an image is made up some number of rows and columns, the pixel color values are often stored as a two-dimensional array (rows being the first dimension, and columns being the second dimension).

Finally, we may want to refer to a specific pixel within the collection of pixels that make up an image. We can do this with a row index and a column index to uniquely identify a specific pixel via its location.

For this project, you will implement the beginnings of three classes to represent colors, images, and locations within an image. The classes are defined next.

ColorClass

The first class you are required to implement for this project will contain information describing a color, and the name of this class must be **ColorClass**. Required attributes (data members) include: an integer representing the amount of red in a color, an integer representing the amount of green in a color, and an integer representing the amount of blue in a color. In addition to the required functionality below, you may choose to implement additional member functions that are appropriate for your design, but any additional member functions must be a good design and kept private.

Required functionality within the **ColorClass** include the following **member functions**, which must be implemented using exactly the names and parameters as shown:

These constructors allow an initial value of a color object's RGB components to be initialized appropriately. The default ctor will set the color's initial RGB values to the color full white, and the value ctor will set the color's initial RGB values to the values provided. For the value ctor, if a specified color value is outside the valid range, it will be "clipped" to ensure all ColorClass attributes are always set to be within the valid range. Clipping is done such that a requested color value less than the allowed minimum is clipped to the minimum, while color values higher than the allowed maximum are clipped to the maximum.

```
ColorClass()
ColorClass(
     int inRed,
     int inGreen,
     int inBlue
     )
```

The following ColorClass member functions simply set the color's component color values to the appropriate values to form the color indicated. In all cases, these function will result in "full" colors as defined above.

```
void setToBlack()
void setToRed()
void setToGreen()
void setToBlue()
void setToWhite()
```

This function sets the color object's RGB values to the provided values. If any input value is outside the allowed color value range, then the assigned value is "clipped" in order to keep the RGB color values within the valid range. If any clipping was necessary, the function returns true, otherwise the function returns false.

```
bool setTo(
     int inRed,
     int inGreen,
     int inBlue
     )
```

This function sets the color's component color values to the same as those in the "inColor" input parameter. If any clipping was necessary in order to keep the color values in the allowed range, the function returns true, otherwise it returns false. Note that, since a ColorClass object is used as input, and since we are making sure ColorClass attributes are always within the valid range, this function should never find a need to clipping, and therefore, would be expected to always return false.

```
bool setTo(
     ColorClass &inColor
     )
```

This function causes each RGB value to have the corresponding value from the input parameter color added to it. At the completion of this function, the RGB values will still be in the allowed color value range. If any resulting color value would end up outside the valid color value range, the value is "clipped" in order to keep the RGB color values within the valid range. If any clipping was necessary, the function returns true, otherwise the function returns false.

```
bool addColor(
     ColorClass &rhs
     )
```

This function causes each RGB value to have the corresponding value from the input parameter subtracted from it. At the completion of this function, the RGB values will still be in the allowed color value range. If any resulting color value would end up outside the valid color value range, the value is "clipped" in order to keep the RGB color values within the valid range. If any clipping was necessary, the function returns true, otherwise the function returns false.

```
bool subtractColor(
     ColorClass &rhs
     )
```

This function performs a simplified brightness adjustment which multiplies each RGB value by the adjustment factor provided. If adjFactor is greater than 1, the color gets brighter, if adjFactor is less than 1, the color gets dimmer. At the completion of this function, the RGB values will still be in the allowed color value range. If any resulting color value would end up outside the valid color value range, the value is "clipped" in order to keep the RGB color values within the valid range. If any clipping was necessary, the function returns true, otherwise the function returns false. Since our color values are integers and the adjustment factor is a double, the product will be a double – to assign back to an int for the updated color value, just type cast to an int, as opposed to trying to implement a rounding scheme.

```
bool adjustBrightness(
     double adjFactor
     )
```

Prints the component color values to the console using the following format: "R: <red> G: <green> B: <blue>" where <red>, <green>, and <blue> are all replaced with their corresponding component color values. Note: NO newline character is printed!

```
void printComponentValues()
```

RowColumnClass

The second class you will implement is a very simple class to uniquely identify a specific pixel within an image. Instead of writing this class, we could have chosen to always refer to a pixel by providing both index values separately, but this class will allow us to refer to a pixel location with a single object, and thanks to encapsulation, will also allow us to develop functionality involving a RowColumnClass, like the ability to add two locations together, etc. This class only requires two attributes: an integer representing the row index of the location, and an integer representing the column index of the location. Important note: this RowColumnClass should be implemented as a class that can stand alone and must *not* be developed in a way that ties it specifically to this project. In other words, the RowColumnClass should not know anything about ColorClass objects (defined above) or ColorImageClass objects (defined later). Do not put limits on the row or column index values that can be stored. Even if *this project's* image size is specified, that should

be a property of the image itself. This RowColumnClass should be able to be used to support *any* image of any image type and/or size, so make sure you implement it in this generic way. Don't make this class difficult or long – it should be very easy and very short.

Required functionality within the **RowColumnClass** include the following **member functions**, which must be implemented using exactly the names and parameters as shown below. You may add additional member functions if they are appropriate in the design, but any additional member functions must be kept private.

This default constructor simply sets both the row and column value of the newly created RowColumnClass object to -99999.

```
RowColumnClass()
```

This value constructor simply sets the row and column values to the corresponding values passed into the constructor.

```
RowColumnClass(
     int inRow,
     int inCol
     )
```

These functions are simple "setter functions" that just directly set the appropriate attribute(s) to the value(s) provided.

```
void setRowCol(
     int inRow,
     int inCol
     )

void setRow(
     int inRow
     )

void setCol(
     int inCol
     )
```

These functions are simple "getter functions" that just return the appropriate attribute value to the caller.

```
int getRow()

int getCol()
```

This function adds the row and column index values in the input parameter to the row and column index of the object the function is called on

```
void addRowColTo(
     RowColumnClass &inRowCol
     )
```

This function prints this object's attribute's in the format "[<row>,<col>]" where <row> and <col> are replaced with the value of the corresponding attribute values. Note: NO newline character is printed!

```
void printRowCol(
     )
```

ColorImageClass

Finally, the last class you will implement will be named ColorImageClass and will represent a small image. As described earlier, a color image will simply be defined as a collection of color pixel values arranged in a rectangle. Therefore, the only required attribute for ColorImageClass is a 2D array of ColorClass objects. The array must be organized such that the first dimension is the row dimension and the second dimension is the column dimension, such that indexing into the 2D array like this "[2][6]" would mean row 2 (i.e. the 3rd row since we are 0-based indexing) and column 6 (i.e. the 7th column). Since we have not yet talked about dynamic memory allocation, our ColorImageClass will only support images of a set size. For this project, your 2D array in ColorImageClass must be set to 10 rows by 18 columns. Obviously, that would be a very small image, but you should develop your class such that the size can be very easily changed if needed.

Required functionality within the **ColorImageClass** include the following **member functions**, which must be implemented using exactly the names and parameters as shown below. You may add additional member functions if they are appropriate in the design, but any additional member functions must be a good design and kept private.

This default constructor simply sets all pixels in the image to full black.

```
ColorImageClass(
     )
```

This function initializes all image pixels to the color provided via input

```
void initializeTo(
     ColorClass &inColor
     )
```

This function performs a pixel-wise addition, such that each pixel in the image has the pixel in the corresponding location in the right hand side input image added to it. If the result of one or more of the pixel additions required color value clipping, this function returns true, otherwise it returns false.

```
bool addImageTo(
     ColorImageClass &rhsImg
     )
```

This function causes the image's pixel values to be set to the sum of the corresponding pixels in each image in the imagesToAdd input parameter. In other words, this image's pixel at row 0 column 0 will be assigned the sum of each input image's pixel value at row 0, column 0, etc. Note: this function does not "add TO" this image - it simply adds the input images and assigns the result to this image. If the result of one or more of the pixel additions required color value clipping, this function returns true, otherwise it returns false.

```
bool addImages(
     int numImgsToAdd,
     ColorImageClass imagesToAdd []
     )
```

This function attempts to set the pixel at the location specified by the "inRowCol" parameter to the color specified via the "inColor" parameter. If the location specified is a valid location for the image, the pixel value is changed and the function returns true, otherwise the image is not modified in ANY way, and the function returns false.

```
bool setColorAtLocation(
```

```
    RowColumnClass &inRowCol,
    ColorClass &inColor
    )
```

If the row/column provided is a valid row/column for the image, this function returns true and the output parameter "outColor" is assigned to the color of the image pixel at that location. If the row/column is invalid (i.e. outside the image bounds) then the function returns false, and the output parameter "outColor" is not modified in any way.

```
bool getColorAtLocation(
    RowColumnClass &inRowCol,
    ColorClass &outColor
    )
```

This function prints the contents of the image to the screen. Each pixel is printed using the format described for the ColorClass above. The image is printed one row of pixels per line (i.e. a newline is printed after the last pixel of each row). Between each pixel in a row, two dashes are printed (i.e. "--") in order to separate the values visually for easier understanding by the end user.

```
void printImage()
```

Additional Specifications / Information:

For this project, you are really implementing a "framework" of useful classes, as opposed to developing a solution to a specific problem. In other words, the classes that you implement in this project could be the basis for additional work to perform image-based processing and analysis. Since we are just developing a framework, the only `main()` function you need to write is for your own testing. The contents of the main function that you submit should simply be the main function provided in the sample, but formatted to be consistent with your programming style. Clearly, though, you will want to write your own main function to fully test the functions developed for this project.

Special Testing Requirement:

In order to allow me to easily test your program using my own main() function in place of yours, some special preprocessor directives will be required. Immediately before the definition of the main() function, (but after ***all*** other prior source code), include the following lines EXACTLY:

```
#ifdef ANDREW_TEST
#include "andrewTest.h"
#else
```

and immediately following the main() function, include the following line EXACTLY:

```
#endif
```

Therefore, your source code should look as follows:

```
library includes
program header
constant declarations and initializations
global function prototypes (if needed)
class definitions
#ifdef ANDREW_TEST
```

```
#include "andrewTest.h"
#else
int main()
{
  implementation of main function
}
#endif
global function definitions (if needed)
class member function definitions
```

Lines above in red are to be used exactly as shown. Other lines simply represent the location of those items within the source code file.

Design and Implementation Details:

Remember that you are allowed to implement additional private member functions as appropriate for your design. In my solution, I implemented a few additional member functions that came in handy because I needed to perform the same functionality multiple times, so these additional functions minimized the amount of duplicated code in places. Your design is up to you – but it should be a logical and quality design without significant duplicated code.

I suggest you implement this program (and all programs, really) in a piece-wise fashion starting with the basics and adding functionality as you go. You can certainly implement the entire RowColumnClass (one function at a time, or a few statements at a time) without ever even beginning the color or image classes, and I would recommend doing so. Once you've implemented and thoroughly tested your RowColumnClass, begin working on ColorClass. When it is fully implemented and thoroughly tested, then move on to the ColorImageClass. This will allow you to make continual progress and mostly avoid cases where you get frustrated by a zillion compile errors because you've implemented a ton of code without ever compiling it.

"Specific Specifications"

These "specific specifications" are meant to state whether or not something is allowed. A "no" means you definitely may NOT use that item. In general, you can assume that you should not be using anything that has not yet been covered in lecture (as of the first posting of the project).

- Use of Goto: No
- Global Variables / Objects: No
- Global Functions: Yes (as necessary)
- Use of Friend Functions / Classes: No
- Use of Structs: No
- Use of Classes: Yes – required!
- Public Data In Classes: **No** (all data members must be private)
- Use of Inheritance / Polymorphism: No
- Use of Arrays: Yes – required!
- Use of C++ "string" Type: No
- Use of C-Strings: No
- Use of Pointers: No
- Use of STL Containers: No
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: No
- Use of exit(): No
- Use of overloaded operators: No
- Use of float type: **No** (That is, all floating point values should be type double, not float)

Sample Program Output:

I will provide a simple `main()` function that will provide some basic test cases to get your started. I will also provide the output that results from the instructor's solution. Sample output is provided on the project web site and will allow you to see what is expected and to compare your program's output.
Important note: The sample output shows you what is expected for a very limited number of cases. If your program produces the exact same output, that does NOT mean that you are finished. There are many cases that were not tested in the provided output, and it is expected that you will thoroughly test every functionality specified above, whether it is explicitly included in the sample main or not.