EECS402, Fall 2025, 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 **Tuesday, November 4, 2025 at 4:30pm**. Early submissions are allowed, with bonus points added according to the policy described in detail in the course syllabus.

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. Your program's "main" function must be in a file named exactly "imageMods402.cpp". 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 exactly "imageMods402.exe". Also, your Makefile must have a target named "clean" that removes all your .o files and your executable (but not your source code!). Finally, you must submit a "typescript" file as you have for prior projects that shows you building your program and running valgrind on a reasonable test case with no memory leaks.

When submitting your project, be sure that **every** file (both .h and .cpp files, and the Makefile and typescript file!!!) are attached to the submission email (do not zip, tar, or otherwise compress or combine your submission files – attach each file separately). The submission system will respond with the number of files accepted as part of your submission, and a list of all the files accepted – it is ***<u>your responsibility</u>*** to ensure all files were attached to the email and were accepted by the system. If you forget to submit a file on accident, we will not allow you to add the file after the deadline, so please take the time to carefully check the submission response email to be completely sure every single file you intended to submit was accepted by the system.

Detailed Description:

In the previous project, you developed classes for representing a Color, a Color Image, and a Row/Column Location. This project will use those same concepts but will focus on the use of dynamic allocation of arrays and file input/output, as well as separating your implementation into multiple files. We've also talked about detecting and overcoming stream issues, and you'll be expected to manage that as well.

Background: .ppm Imagery

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.

Most image types start with two special characters, which are referred to as that image type's "magic number" (not to be confused with the magic numbers we've talked about as being bad style in programming). A computer program 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", which simply allows an image viewing program to determine that this file is a PPM image and should be read using the PPM format. If a file doesn't start with the very specific magic number "P3", it is to be considered an invalid PPM file.

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". (note: width comes first, and height comes second! People always get this mixed up, so take care with the order…)

The next value is also an integer, and is simply the maximum value in the color descriptions. 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). For this project, a value of 255 is the only valid "maximum color value" allowed, and any value that is NOT 255 will be considered an "invalid PPM image" and should be reported as an error. Note that, generally, PPMs can have different max values, but to keep things simpler, we will consider any max color value other than 255 as invalid.

The only thing left is a description of each and 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 descriptions). As mentioned above, each pixel is described with three integers (red, green, blue), so a 4 row by 4 column color image requires 4*4*3=48 integers to describe the pixels. If a PPM file doesn't have *exactly* the number of RGB triples to match the number of rows and columns specified, then the file is considered an invalid PPM file.

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
```

Please note that I have provided this example in a nicely formatted way for ease of understanding, but a PPM file is not required to be “line-based”. In other words, there is nothing in the PPM format that requires one row of pixels to be contained on a single line, or that pixel values are separated with more spaces then the RGB values, etc. For example, the following PPM file should be considered identical to the nicely formatted one above:

```
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
```

Since this is the case, you should NOT try to use line-based inputs (like getline) for this project, and instead, just use the extraction operator (>>) when reading the contents of a PPM file.

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 which should be able to be downloaded for free from www.irfanview.com).

File Naming For PPM Images

Most often, PPM files will end with a “.ppm” extension. However, this is not a requirement, and your program should not perform any checking for a specific filename extension. While it might be very confusing, there could be a file named “definitelyNotAPPM.exe” with contents that were a valid PPM – if we were to provide such a file to your implementation, you should read the contents of the file just as if it were named “myPPM.ppm”. Typically, file extensions are used as a convention for general understanding of what a file contains, but (especially in Linux) this is strictly a convention as opposed to a requirement.

Required Functionality

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 rectangles to be drawn on an image. 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 placed on an image. A pattern consists of a rectangle of only zeros and ones. When a pattern is placed over an image, values in the pattern will likely fall over a specific pixel in the original image. A value of 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, resulting in a sort of transparency.

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 change the original image, but pixels that are not the transparency color will be used to replace the pixel value in the original image. Note that this is very similar to the use of a pattern, described above, except that patterns can only be one color, while inserted images can have as many colors as the PPM allows.

At any stage, you must allow the user to output a PPM image file in its current state from the main menu. The user may want to output an image after each change made, or just once when all updates 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.

Out of Bounds Patterns and Inserted Images

When inserting a pattern or image, it is possible that part, or all, of the inserted item will extend beyond the bounds of the image being inserted into. This should not be considered an error for this project, but rather any pixels that are within bounds should be inserted as usual, and the ones that are out of bounds should just not have any effect. This approach allows a user to purposely insert a pattern or image at a location that will result in only part of the inserted item being included. For example, if a user specifies an image insertion location as row -1 and column -1, then the first row and column of the inserted image will not be seen in the result. Similarly, if the image being inserted into is 100x100 in size, and a pattern being inserted is 10x10, if the user specifies an insertion location of row 95 column 30, then the full top 5 rows of the pattern will be inserted at the bottom of the image, but the bottom 5 rows of the pattern will be "cut off". Inserting a pattern or image will never cause the size of the original image to be changed in any way.

Implementation and Design

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).

While PPM images and patterns are not really limited in size for any particular reason, for the purposes of this project, we are limiting "valid" PPMs and patterns to be between 1 and 2000 (inclusive) pixels in any dimension. Sizes outside of that range will be considered an invalid PPM file, even if the rest of the file is formatted properly.

I'll leave the majority of the design up to you, but remember I will be looking at your design during grading. If you find you want some global functions, you may implement them. However, remember that this project is required to be implemented in an object-oriented way, so if you find yourself writing global functions, you should stop and consider whether they ought to be methods of one of your classes instead.

Each individual class will be contained in a .h and a .cpp file (named with the class name before the dot). ALL class member variables MUST be private. Your member functions may be public. ALL method implementations must be in the .cpp file for the class – do not implement any methods in your .h files! For example, if you have a class called "FooClass" then the class definition and documented prototypes will be in a file named FooClass.h and all method implementations will be in FooClass.cpp.

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 (unless they are overloaded using the same name, and therefore belong in the same file). Global functions must be implemented in a .cpp file as opposed to the .h file (which will only contain the documented prototype). For example, if you have a global function called "clipIntToRange", then the documented function prototype will be in a file named clipIntToRange.h and the function will be implemented in a file named clipIntToRange.cpp.

Remember, when submitting, you must submit ALL .h files, .cpp files, your Makefile, and the typescript file. Do not include your .o files or your executable in your submission.

While you might want to make use of your framework from the previous project, there are some important changes to note:

1) The maximum color value will now be 255 (instead of 1000). If you didn't use magic numbers, it should be rather straightforward to update your code to use a max color value of 255 in place of the previous value of 1000.
2) The ColorImageClass developed in the previous project had a matrix of pixels that was statically allocated with a specified size – for this project, the size will not be known at compile time, and you must use dynamic allocation to allocate exactly the number of pixels needed – no more and no less.
3) There is no need for clipping in this project. In the previous project, clipping was a big deal, but it is not part of this project, so you can remove the clipping-related functionality if you want to. If you choose to keep the functionality from the previous project (like "add images" and "adjust brightness"), that functionality should clip to the max value of 255 instead of 1000. That functionality is not part of this project and will not be tested for this project, but you may choose to leave it in as long as it doesn't cause any issues.
4) You'll have to add functionality as required for this project.

"Class Responsibility" will be an important design aspect of this project. Thinking about the functionality to write and read images to/from files - when developing this functionality, remember that a ColorImageClass object should write/read image-related attributes to/from files. It is really each individual pixel's responsibility to write/read its own color to/from the file. In other words, the ColorImageClass write/read methods should not write/read color RGB values or do anything with "red", "green", or "blue" valus at all (the ColorImageClass shouldn't even know the details of what a ColorClass has as attributes, etc). Instead, the ColorImageClass should call a member function of the ColorClass to write those values. Always think about this type of thing when designing your project.

You'll see that when choosing to annotate an image with a rectangle, there are three different methods you must support – specifying the rectangle via: 1) the upper-left and lower-right locations directly; 2) specifying the upper-left corner and a width and height; and 3) specifying the center of a rectangle and a width extent and height extent from the center (i.e. half-width and half-height). At first glance, this seems like tedious "make work", but the reason for requiring three methods to do the same thing is to make you think about your design of your Rectangle class. One thing to remember is that, regardless of which method I use to specify the rectangle, the resulting rectangle can be described using a pre-defined set of attributes. For example, even if the user uses method 3 to specify a rectangle, internally in your program, it can be stored, described, and used via an upper-left corner and a lower-right corner. In other words, there is no need to have attributes in your rectangle class to support each input method – simply convert the values the user input to the attributes you will store all rectangles as. This is another type of thing we will be looking at in your design, so its worth understanding and doing correctly.

While you have more flexibility in your project design, your menu and the way your project operates must match that shown in the sample output. Do not change the actions associated with menu options, orderings, expected user inputs, etc. I must be able to input the exact same values I would to my solution, in the exact same order, and have the program act accordingly. Do not add additional prompts that the user has to respond to, re-order menu options, change the number of items requested for input, 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;
//get the name of the file stored in fname somehow
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. In addition to being an object-oriented program using dynamic allocation and file I/O, this project will focus on error checking, and many of our test cases during grading will be “nitpicky” to check that you detected and handled errors that might come up appropriately.

If no fatal errors occur (see the next two paragraphs below), your program must return a value of 0.

The “main image” (i.e. the background image) will be specified on the command line as the only command line argument after your executable name. If the user does not provide exactly one argument after the executable name, then a proper usage statement should be printed and the program should immediately exit with an exit value of 2. Note: to utilize the exit function in this project, you will need to add:

```
#include <cstdlib>
```

When reading of the main image specified via command line argument, if the image is invalid in any way, print a ***descriptive*** error message and the program should immediate exit with an exit value of 3.

If other files can't be read or written during the program (pattern files, other images, etc.), output a ***descriptive*** error message and continue the program. The program should ***not*** exit in these cases – the reasoning is that the user may have spent hours annotating an image, etc., and if they make a simple typo when trying to type the name of a pattern file (for example) you don’t want the user to have to start over. 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, and doesn’t provide the user any insight as to how they can fix the problem. 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 (there’s quite a few things to consider that could go wrong when reading an image file).

If the user specifies an invalid PPM or invalid pattern file, then no modifications should be made to main image. For example, if the user requests to insert a pattern, but the pattern file they specify is found to be invalid, no elements of the pattern should be inserted on the image, even if some of the pattern file seemed valid.

For integer-type individual inputs or “row column” paired inputs, you should ensure the user provided a valid value and if not, re-prompt the user to enter the data again.

Testing and Autograder

You do NOT need to include the “ANDREW_TEST” stuff for this project as in previous projects. Since you’ll be designing your own classes, we will not be able to write our own main function to test your functionality. Therefore, we’ll be relying on your main and your menu system when grading. This means it is very important that your menu and prompts for inputs, etc., are setup exactly like those shown in the sample outputs. If you ask for input in a different order, or request an extra input, etc., your project will fail the automated tests, even if your functionality of the image processing is correct.

We will use the autograder to check standard cases (i.e. non-error cases) and it will also check that you recognized error situations for many different error cases. During autograder testing, we will only be checking that your program recognized an error and, in cases where the program is supposed to exit due to the error, that the program exits with the expected return value. However, we will NOT be providing a list of all the error situations that you need to deal with, and we will not be providing you with the expected output format for those error cases. Therefore, we will do further error case checking for correctness after the final project deadline to ensure that not only does your program produce the correct exit value, but also that the output error messages are detailed and informative. We may also introduce some additional error cases to be considered after the final deadline that are not part of the test suite performed by the autograder. You need to make sure to consider, and properly handle, different error situations that may come up.

I’ll reiterate since this may be different than the way earlier projects were handled. For this project, the results from the autograder will not necessarily be your final correctness grade. In addition to fairly standard test cases, the autograder will likely give you a good idea if you’ve thought of and recognized most of the error cases, but the autograder will not be checking the quality and specificity of your error messages. As described above, we will be manually running the error checking cases after the final deadline, and you may end up getting deductions on test cases that the autograder reported you passed, if your output messages are deemed insufficient. Finally, we will also likely be running a few extra error-checking-related test cases after the final deadline that were not included in the autograder tests. Please remember/realize that your correctness grade may change from what the autograder reports for this project!

"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. We have not necessarily covered all the topics listed, so if you don’t know what each of these is, it’s not likely you would “accidentally” use them in your solution. Those types of restrictions are put in place mainly for students who know some of the more advanced topics and might try to use them when they’re not expected or allowed. 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
- Use of C++ "string" Type: Yes
- Use of C-Strings: No (except as noted to satisfy the “open” method)
- Use of Pointers: Yes – required! All matrices for images/patterns must use dynamic allocation
- Use of STL Containers: **No**
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes – required!
- Use of exit(): **Yes** (as specified)
- Use of overloaded operators: No
- Use of float type: **No** (That is, all floating point values should be type double, not float)