

EECS402, Winter 2022, Project 3

Overview:

Going over boring lectures uses energy! What?! Did you notice that if you take the first letter of each word in the first sentence, it spells “Go Blue!”? Probably not, as unless you are looking for that pattern, it doesn’t stand out as anything special. However, if the recipient knew what to look for, it is incredibly simple to obtain the intended message. This is called steganography, which means hiding data or information in the midst of otherwise unsuspicious data or information. Steganography can be used to transmit encoded messages to a recipient without the need for ciphers, and this can be a benefit because an onlooker might not even recognize it as a secret message at all, meaning they wouldn’t know to try to “decode” it.

Digital images happen to be a fantastic means to support the use of steganography, as most humans just look at the colors represented by the large array of pixel values and their brain interprets an image. As we learned in the previous project, though, that digital image is made up of many small pieces of data, and when managed in specific ways, can easily have information encoded within that data that virtually makes no difference to the image that the human eye sees.

In this project, we will utilize our knowledge of file input/output and object-oriented programming in order to implement a steganography technique allowing us to “encode” messages into an image, such that the modified image could be sent to a friend, or posted on a social media site, and would likely only be able to be decoded by someone who knows it contains a hidden message.

Due Date and Submitting:

This project is due on **Thursday, March 24, 2022 at 4:30pm**. Early submissions are allowed, with corresponding bonus points included, 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. 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". Also, your Makefile must have a target named “clean” that removes all of your .o files and your executable (but not your source code!). Do not zip/tar/rar/etc. your files in any way, please just attach them to your submission email as individual attachments.

When submitting your project, be sure that **every** source file (both .h and .cpp files!!!) are attached to the submission email. 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 **your responsibility** to ensure all source 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. Of course, we’ve also

talked about detecting and overcoming stream Input/Output issues, and you'll be expected to manage that as well.

Please take note: In previous projects, we allowed you to neglect error checking in order to keep things simpler. For this project, though, error checking is one of the primary things we will be looking for, so you should expect more "nitpicky" test cases where we purposely try to provide inputs that could cause a program to fail or act in an unexpected way. Be sure to consider error checking every step of the way in this project.

Background: .ppm Imagery

Since you will be reading and writing images, you need some background on how image files 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 quality of image as a .gif or .jpg, but it would likely be significantly larger in file size since no compression is used. 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.

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 (i.e. number of columns) comes first, and height (i.e. number of rows) 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 allowed value in the color descriptions**. For the purposes of this project, we'll require that this maximum allowed value be the integer value 255 – a file with any other "maximum allowed value" than 255 must be considered an "invalid PPM file" by your project. 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 allowed value is 255, you could get a much wider range of colors (255^3).

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 of pixels 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 $\text{rows} \times \text{cols}$ pixel descriptions in the .ppm file). Each pixel is described with *three* integers (red, green, blue), so a 4 row by 4 column color image requires $4 \times 4 \times 3 = 48$ integers to describe the pixels. The integer values are separated with an arbitrary amount of whitespace. While it is fairly common to separate individual RGB values with a single space and to have a newline character after one "row's worth" of RGB values, this is NOT a requirement of the PPM format. In the above example of a 4x4 image, we needed 48 integers for the RGB

values, and those 48 values will be separated by arbitrary whitespace (spaces, tabs, and/or newlines), there may be 12 RGB values on each line of pixel data, or there may be 3 RGB values on each line, there may be a single value on each line, or all 48 might all be on one line. A final note is that there may or may not be a newline character after the last value for the last pixel's RGB triple – either will be considered acceptable.

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
```

As noted above, this image could also be stored, as a valid PPM file, using the following format:

```
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 both of these would be considered valid PPMs, you must read the image files using “line-based” inputs (i.e. don’t use the getline function).

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 free 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. Just remember that JPEG compression is “lossy”, so converting to JPEG will allow you to easily view the image as an image, but you won’t be able to consider individual pixel RGB values specifically in a JPEG image. One way to convert a PPM to JPG is to use the Linux command "cjpeg" like this:

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

Or using Linux’s ImageMagick to convert like this:

```
% convert inFile.ppm outFile.jpg
```

Note: The "%" character shown in the commands is just meant to be the Linux prompt – it is not part of the command you would type in.

Encoding and Decoding Messages

As mentioned earlier, our steganography approach in this project will encode secret data within the pixel data of an otherwise uninteresting image. Humans are not very good at detecting tiny differences in color – for example, the full red RGB of (255, 0, 0) is essentially not distinguishable from the almost-full red RGB of (254, 1, 1). Therefore, we can make tiny modifications to colors and a human will not be able to detect the differences.

We will exploit this fact by modifying a portion of an image such that whether the individual RGB components are even or odd implies different colors. This limits us to being able to encode only 8 different colors, as described here:

- Encoding value: 0, Color description: Black, Encoding RGB: (Even, Even, Even)
- Encoding value: 1, Color description: Red, Encoding RGB: (Odd, Even, Even)
- Encoding value: 2, Color description: Green, Encoding RGB: (Even, Odd, Even)
- Encoding value: 3, Color description: Blue, Encoding RGB: (Even, Even, Odd)
- Encoding value: 4, Color description: White, Encoding RGB: (Odd, Odd, Odd)
- Encoding value: 5, Color description: Yellow, Encoding RGB: (Odd, Odd, Even)
- Encoding value: 6, Color description: Magenta, Encoding RGB: (Odd, Even, Odd)
- Encoding value: 7, Color description: Cyan, Encoding RGB: (Even, Odd, Odd)

It's important to realize that we aren't just changing pixel values to the specified "encoding value", because that would likely be very noticeable in the image. Instead, we are going to **tweak the RGB components of a pixel by at most 1**, which will be essentially undetectable. Here's how it will work – let's say the current pixel value we're going to encode is made up of RGB values of (108, 217, 62). If we want to encode the color yellow into the pixel, we need (Odd, Odd, Even), so we modify the RGB values to be (109, 217, 62). Similarly, if we had wanted to encode the color green, we need (Even, Odd, Even) – if the original pixel value was (108, 217, 62), then no changes are needed, as it already represents the encoded color green. As a final example, if we wanted to encode the color magenta, we need (Odd, Even, Odd), and again starting with (108, 217, 62), we can see that all three color components need to be modified, resulting in (109, 216, 63).

Algorithmically, here is the description: If a color attribute is even and we need to encode even, or it is odd and we need to encode odd, then no change to the value is made. **If a color is odd and we need to encode even, then we simply subtract one from its value.** Finally, **if a color is even, and we need to encode odd, then we add one to its value.** Notice that we don't need to worry about "special cases" at the boundaries with this algorithm, because we never subtract from an even number, so we'd never subtract from 0 (which would result in an invalid -1 color attribute), and we never add to an odd number, so we'd never add to 255 (which would result in an invalid 256 color attribute).

Decoding the message simply works in reverse. If an encoded pixel's value is (Even, Odd, Even), then it would be decoded to green, meaning the RGB components would be set to (0, 255, 0). All of our decoded colors should be "full colors", meaning they are made up of some combinations of only 0s and 255s. **Something to note: there is no way for us to "undo" a decode operation.** Once a pixel has been decoded, its original RGB values have been replaced with the decoded color RGBs.

Figure 1 provides a full, albeit tiny, example. The images that are shown in the figure have been zoomed to 2000% of their actual size for easier viewing in this document. If you were to make these images and view them without zooming, they would be so small, that they'd be almost impossible to interpret. Beneath the images is the contents of the corresponding files. The images are formatted as small PPM files as described earlier, and the message file is formatted not as a PPM, but in a separate message format described below.

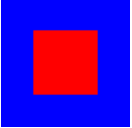
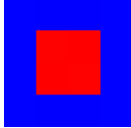

<u>Original Image</u>	<u>Message File</u>	<u>Encoded Image</u>	<u>Decoded Image</u>
	<not an image>		
P3 4 4 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 255 0 0 255 0 0 0 255 0 0 255 255 0 0 255 0 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255	4 4 7 1 1 7 7 7 1 7 7 1 7 7 7 1 1 7	P3 4 4 255 0 1 255 1 0 254 1 0 254 0 1 255 0 1 255 254 1 1 255 0 0 0 1 255 0 1 255 255 0 0 254 1 1 0 1 255 0 1 255 1 0 254 1 0 254 0 1 255	P3 4 4 255 0 255 255 255 0 0 255 0 0 0 255 255 0 255 255 0 255 255 255 0 0 0 255 255 0 255 255 255 0 0 0 255 255 0 255 255 0 255 255 255 0 0 255 0 0 0 255 255

Figure 1: A (tiny) end-to-end example. The images shown have been zoomed to 2000% because a 4x4 image is very tiny.

The message file describes the message that is to be encoded in an image. Its format is similar to the PPM in some ways, but since it is not actually a viewable image, there is no “magic number” in the file. The first field is the message’s width (again, number of columns first!) and the second field is the message’s height (number of rows). Following that is a description of which color should be encoded for every “pixel” in the message. The values will be separated by arbitrary whitespace (spaces, tabs, and/or newlines), and there is NO requirement that one “row’s worth” of values be on a single line in the file, etc. In the above example, we see the message is made up of 7s (cyan) and 1s (red). If you look at the message file, you should recognize that the message I want to encode is a cyan-colored “N” over a red background.

Using the Program and Menu System

Using the program will consist of providing the name of a file representing an initial PPM formatted image as command line input, and then using a minimal menu-driven system to request encodings, perform decoding, and write resulting images out to PPM files.

As with any programs that require the user to provide command line argument(s), you must check that the user provided the correct number of arguments, and if not, provide a usage statement that informs the user of the correct way to use your program. Since this represents an unrecoverable error, you may call the exit(...) function to abruptly end the program – in this case, **provide an exit value of 2** to indicate this particular issue occurred. To use the exit function, you must #include <cstdlib>.

If the filename the user provides on the command line is not able to be read successfully as a properly formatted PPM image, you must provide an *informative* error message so the user knows exactly what went wrong, and the program should exit. Since this represents an unrecoverable error, you may call the exit(...) function to abruptly end the program – in this case, **provide an exit value of 3** to indicate this particular issue occurred.

Once the initial image has been read in successfully, any errors occurring afterwards are recoverable errors, and your program **must not use the exit function** for any reason. When a recoverable error occurs, print an *informative* error message so the user knows exactly what went wrong, and continue running the program normally. Errors must not cause your program to seg fault, infinite loop, produce crazy results, etc.

The menu will look like this:

```
1. Encode a message from file
2. Perform decode
3. Write current image to file
4. Exit the program
Enter your choice:
```

When encoding a message, the user will be prompted for which pixel in the original image the encoded message's upper left corner should be placed at. Performing the decode operation does not require any additional inputs. When requesting to write the image to file in its current state, the user will be prompted to enter a file name to write the image to. Finally, when exiting the program, a "thank you" message is printed and the program ends (remember, you cannot use exit to accomplish this!). See the sample outputs posted on the project page for what the console output must look like for this project.

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). Therefore, there is no practical limit to the size of the image allowed. However, to enforce "reasonable" sizes (and induce some error checking!), your solution must enforce the use of images between 1 and 2000 pixels (inclusive) in any dimension.

I'll leave the design up to you, but remember I will be looking at your design during grading, and am expecting good and proper object-oriented design. Your design should primarily be object-oriented, but if you find you want some global functions and they shouldn't be member functions, you may implement them. Each individual class will be contained in a .h and a .cpp file (named with the exact class name before the dot – i.e. if you had a class called FooClass, it would have the class definition in FooClass.h and the FooClass method implementations would be in FooClass.cpp). 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 (i.e. if you have a global function called "doBar", then the doBar prototype will be in doBar.h and the doBar implementation will be in doBar.cpp). 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). Remember, when submitting, you must submit ALL .h files, .cpp files, and your Makefile. 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). All "max color values" should use 255 instead of 1000. If you didn't use magic numbers, this should be rather straightforward. 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) The “clipping” isn’t really necessary for this project. You can leave that functionality in if you want to – it shouldn’t hurt anything, but if you want to remove it, that’s ok too. 4) You’ll have to add functionality as required for this project.

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.

Class Responsibility

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 attributes to/from the file*. In other words, the `ColorImageClass` write/read methods should *not* write/read color RGB values (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/read those color-related attribute values. Always think about this idea of “class responsibility” when designing your project.

Memory Management

As of when this project is posted, we will not have yet discussed certain “memory management” issues you may read about or have heard about. “Memory management” is an important part of programming, especially once we start using dynamic allocation. Since this topic hasn’t been taught yet, however, for the purposes of this project only, we won’t require you to deal with the following memory management situations:

- You don’t need to worry about memory leaks
- You don’t need to worry about perform a “deep copy” or “full copy” of an object
- You don’t need to worry about overriding a copy constructor
- You don’t need to worry about implementing a destructor

We’ll be talking about these topics during the project development period, so if you want to include them once we’ve discussed them, that is ok with me, but it is not required for this project. Please note that for future projects, these things will be a required part of those projects.

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 fileName;
//get the name of the file stored in fileName somehow
inFile.open(fileName.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.

I've already discussed handling a user's improper execution via the command line and handling the situation when the initially provided image is not able to be read in successfully.

If the user requests to encode a message that is not in a properly formatted message file, then your program must print an *informative* error message, and **no modifications** are made to the image.

When encoding a message, the user may request a position of the message within the image that causes some or all of the encoded message to be "out of bounds" in the original image. If this happens, you should not consider this an error – instead, **simply encode the message for the pixels that are "in bounds" and do nothing for the ones that are "out of bounds"**. This allows the end-user to partially encode messages on the edge(s) of the image.

There are likely other error conditions that may come up – handle them appropriately. In other words, it's up to you to figure out how you want to handle any error that I have not specifically described. In general, you'll want to provide an *informative* error message to the user and allow the program to continue in an otherwise normal fashion.

"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)
- Global Constants: Yes
- 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! Variable-sized arrays will be dynamically allocated.
- Use of STL Containers: **No**
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes – required!
- Use of exit(): Yes

- Use of overloaded operators: No
- Use of float type: **No** (That is, all floating point values should be type double, not float)