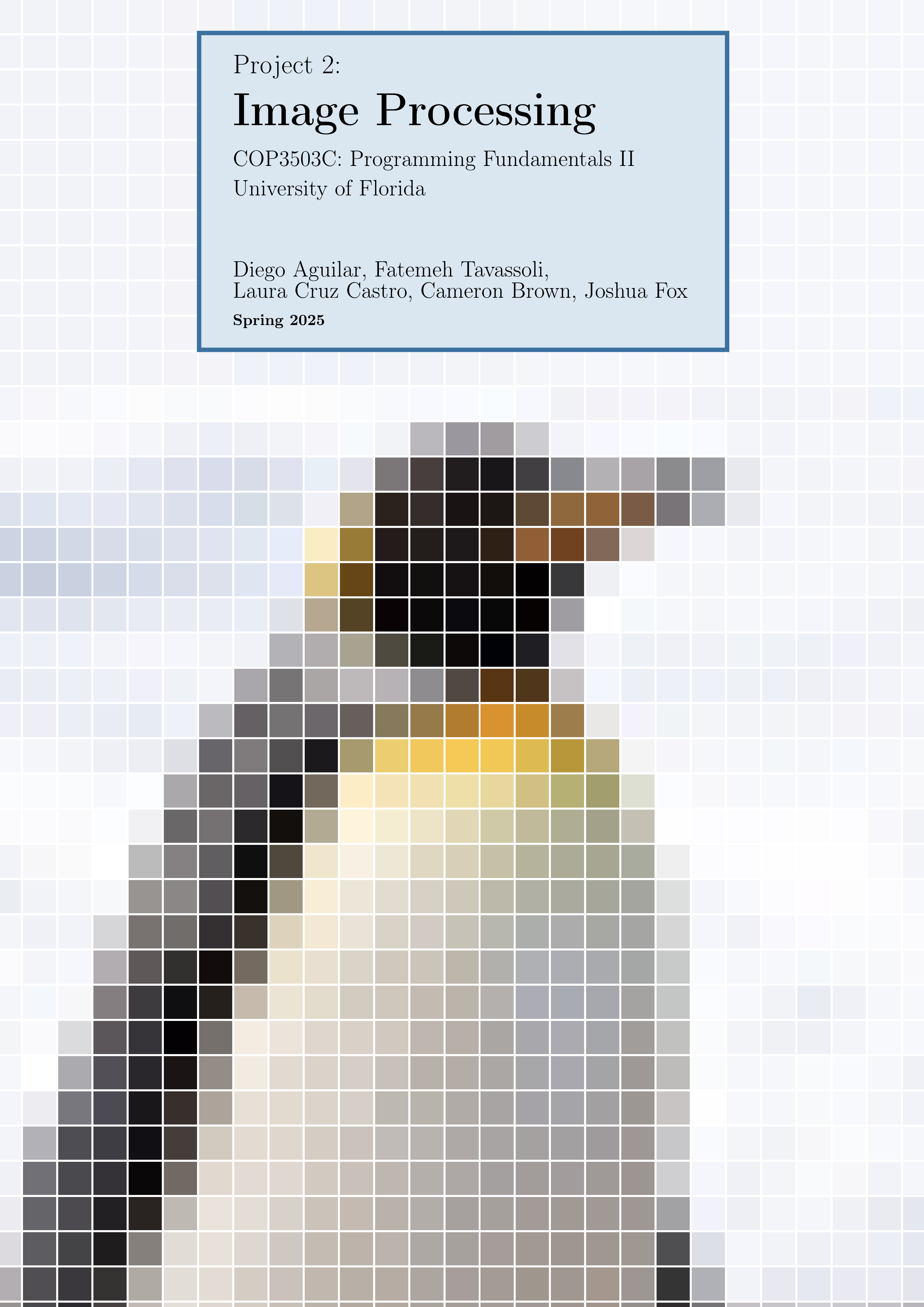

Project 2:

Image Processing

COP3503C: Programming Fundamentals II
University of Florida

Diego Aguilar, Fatemeh Tavassoli,
Laura Cruz Castro, Cameron Brown, Joshua Fox

Spring 2025

Welcome to the second project in Programming Fundamentals II! In this project, you will use many of the various concepts you've learned in order to read, manipulate, and write binary image files. You will design a series of algorithms that can be applied to images in order to change their appearance. Then, you'll make a interface to this program using your computer's command line, allowing you to modify images on the fly. Along the way, you will grow and flex your knowledge of various C++ concepts!

If you wish to skip to just the project requirements:

- For Milestone 1:
  - **Tasks:** subsection 3.4
  - **Makefiles:** subsection 3.5
  - **Submission:** subsection 3.6
- For Milestone 2:
  - **Tasks:** subsubsection 4.2.2
  - **Errors:** subsubsection 4.2.3
  - **Submission:** subsection 4.3

Contents

1 Overview

Lots of applications need to process images in some way. Load them, store them, write them back out to files, scale them, rotate them, adjust the color in part (or all) of the image, etc. The purpose of this assignment is to show you how you can perform some of these operations on a particular type of file. In this assignment, you will:

- Read data from binary `.tga` files
- Process the image data stored within files in a variety of ways
- Write out the new `.tga` files in the same binary format
- Develop a command-line interface to handle user input
- Develop a Makefile to allow others to compile and run your project quickly and easily

2 TGA File Specification

It's recommended to install at least one of the following programs if you do not already have a TGA viewer installed:

- **Visual Studio**: If you have installed Visual Studio, you should be able to click on a .tga file and have the file open up in a new Visual Studio tab.
- **Photoshop**: If you use the popular Photoshop image manipulation tool, you can use this program to view your outputted .tga files.
- **GNU Image Manipulation Program (GIMP)**: A popular free and open-source image editor, GIMP will be able to open your .tga files and is completely free.
- **TGAViewer**: A small, cute little program whose job is just to view .tga files. How nice!

2.1 File Format

Since binary files are all about bytes, they are typically an unreadable mess to any program (or person) that doesn't know exactly how the data is structured. In order to read them properly, you must have some sort of blueprint, schematic, or breakdown of how the information is stored. Without this description of the file format, you would just be reading random combinations of bytes attempting to get some useful information out of it—not the most productive process.

The TGA file format is a relatively simple format, though it has some options which can get a bit complex in some cases. The purpose of this assignment is not make you a master of this particular image format, so a few shortcuts will be taken (more on those later). First, let's take a quick look at the file format, showcased in Table 2.1.

Field	Size	Length	Description
Header	ID Length	1 byte	Denotes the length of the Image ID.
	Color Map Type	1 byte	Specifies if a color map is present.
	Image Type	1 byte	Indicates the type (e.g., grayscale, true color, compressed).
	Color Map Origin	2 bytes	Starting index of the color map, usually 0.
	Color Map Length	2 bytes	Length of the color map, usually 0.
	Color Map Depth	1 byte	Bit depth of the color map, usually 0.
	X Origin	2 bytes	Horizontal image origin, typically 0.
	Y Origin	2 bytes	Vertical image origin, usually 0.
	Image Width	2 bytes	Width of the image in pixels.
	Image Height	2 bytes	Height of the image in pixels.
	Pixel Depth	1 byte	Bit depth of each pixel, typically 24 for RGB.
	Image Descriptor	1 byte	Provides additional details about the image.
Data	Image Data	Variable	Contains pixel data in BGR format. Starting from bottom-left to upper-right. Number of pixels is the product of width and height.

Table 2.1: File specification for TGA files.

So to start, there is a header. Every file format is potentially different, but in a TGA file the header data takes up 18 bytes total, across a number of variables, and this information describes the rest of the file. A possible structure for the header is shown in Listing 2.1.

```
struct Header {
    char idLength;
    char colorMapType;
    char dataTypeCode;
    short colorMapOrigin;
    short colorMapLength;
    char colorMapDepth;
    short xOrigin;
    short yOrigin;
    short width;
    short height;
    char bitsPerPixel;
    char imageDescriptor;
};
```

Listing 2.1: Example of using a C++ `struct` to hold header information.

Note that if you used `char` in your program, when you print out the `char` or `unsigned char` variable, you get a symbol that corresponds to its numeric value, instead of the number itself. **If you want to see the numeric value of a char variable instead of its symbol, you would have to cast it to an integer, as shown in Listing 2.2**.

```
char someVariable = 65;
cout << someVariable; // Prints out 'A' instead of 65
cout << (int)someVariable; // Prints out 65 instead of 'A'
```

Listing 2.2: Showcasing various methods of printing information about a character in C++.

2.2 TGA Image Data

After the header comes the really important part, the image data itself. In a TGA file, the image data is stored in a contiguous block of pixels. The number of pixels in the block is equal to the image's length multiplied by the image's width. The contents of a single pixel can vary depending on the properties of the file, but for this assignment we are using images with 24-bit color. This means that each pixel contains:

- 1 byte (8 bits) for blue data
- 1 byte (8 bits) for green data
- 1 byte (8 bits) for red data

Notice that the order of the color data is BGR, not RGB!

3 Milestone 1: Image Manipulations

! It is recommended to make functions to read and write the data from these files into a format your code can work with (Like a class/struct to represent a TGA image)! It is also recommended to make each task for milestone 1 into a function you can use on these images to make milestone 2 easier.

3.1 Algorithms

These algorithms are described in the Table 3.1. You will be implementing the formulas in the table in your own program. Each formula contains two values, P_1 and P_2. These represent the individual channel values of the two pixels used in the formula. Note that the order of the pixels matters for some algorithms - in these cases, the pixels in the "top layer" become P_1, and the pixels in the "bottom layer" become P_2. NP_1 and NP_2 refer to the normalized values of P_1 and P_2, as explained in subsection 3.2.

Note that these formulas should be calculated for **each channel** of the input images, and then used in **each channel** of the resulting output image. You will need to use the equations in autoreftable:formulas three times per pixel, one time per channel.

! Notice that although these correspond to the tasks denoted later this does not include all of them!

Method	**Channel Formula**
Multiply	$NP_1 \cdot NP_2$
Screen	$1 - [(1 - NP_1) \cdot (1 - NP_2)]$
Subtract	$P_1 - P_2$
Addition	$P_1 + P_2$
Overlay	$\begin{cases} NP_2 \leq 0.5: & 2 \cdot NP_1 \cdot NP_2 \\ NP_2 > 0.5: & 1 - [2 \cdot (1 - NP_1) \cdot (1 - NP_2)] \end{cases}$

Table 3.1: Formulas for common manipulation algorithms.

Note that the "Overlay" method shown in Table 3.1 is a conditional algorithm, meaning the specific algorithm you'll need to run varies based on the input pixel value. You can implement this in code using an `if/else` statement.

3.2 Clamping and Overflow

If you attempt to implement some of the above algorithms using the suggested `unsigned char` data type, you will experience **overflow**. This occurs when one of the above calculations results in a number below zero or greater than 255 - values which can not be stored by that data type.

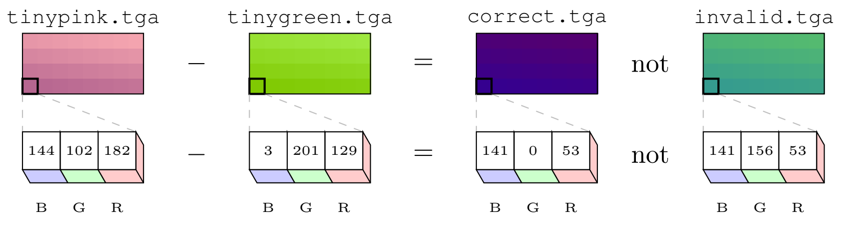

Figure 3.1: When clamping is not done, an entirely different image can arise. Notice how the green value is appropriately clamped to 0 in the correct image, while it overflows to $102 - 201 = -99 + 255 = 156$ in the incorrect image.

3.3 Rounding

For operations involving multiplication, you will need to use normalized values. These values are the original values of the pixel divided by 255. Therefore, the range of a normalized pixel is 0-1. In these operations, you will likely need to use `float` to store the intermediate values. After the operation, you just need to multiply the resulting `float` by 255, and voilà, you have a new pixel value!

Keep in mind that operations involving `float` values will likely result in precision errors. Therefore, you should add `0.5f` to the normalized value multiplied by 255, before converting to an `unsigned char`. Why `0.5f`? This value is perfect for rounding numbers. Numbers closer to the next highest value (such as 79.7) will be rounded up (to 80, in our example). Numbers closer to the lower value (such as 79.2) will be rounded down (to 79).

3.4 Tasks

In this project, you are expected to complete ten different tasks, each of which use different input images and image manipulation algorithms. Your program should be able to complete all ten tasks and produce images that match the images given in the `examples/` folder. The output names for all files you generate should be `output/partX.tga`, where `X` is the task number. For now, all tasks should complete whenever your project is built and ran. The user should not need to enter any input.

1. Use the **Multiply** blending mode to combine `layer1.tga` (top layer) with `pattern1.tga` (bottom layer).
2. Use the **Subtract** blending mode to combine `layer2.tga` (bottom layer) with `car.tga` (top layer).
3. Use the **Multiply** blending mode to combine `layer1.tga` with `pattern2.tga`, and store the results temporarily, in memory (aka, don't write this to a file somewhere, just store the pixel values somewhere in your program). Load the image `text.tga` and, using that as the bottom layer, combine it with the previous results of layer1/pattern2 using the **Screen** blending mode.
4. **Multiply** `layer2.tga` with `circles.tga`, and store it. Load `pattern2.tga` and, using that as the bottom layer, combine it with the previous result using the **Subtract** blending mode.
5. Combine `layer1.tga` (as the top layer) with `pattern1.tga` using the **Overlay** blending mode.
6. Load `car.tga` and **add** 200 to the green channel.
7. Load `car.tga` and scale (**multiply**) the red channel by 4, and the blue channel by 0. This will increase the intensity of any red in the image, while negating any blue it may have.
8. Load `car.tga` and write each channel to a separate file: the red channel should be `part8_r.tga`, the green channel should be `part8_g.tga`, and the blue channel should be `part8_b.tga`. *(Hint: If your red channel image appears all red, try writing [Red, Red, Red] instead of [Red, 0, 0] to the file—ditto for green and blue!)*
9. Load `layer_red.tga`, `layer_green.tga` and `layer_blue.tga`, and combine the three files into one file. The data from `layer_red.tga` is the red channel of the new image, layer_green is green, and layer_blue is blue.
10. Load `text2.tga`, and rotate it 180 degrees, flipping it upside down. This is easier than you think! Try diagramming the data of an image (such as earlier in this document). What would the data look like if you flipped it? Now, how to write some code to accomplish that...?

3.5 Makefile

In your submission for this project, you will need to create a Makefile. This is a small file that allows us to build your project instantly, without needing to run any of our own commands. For this project, you will write a Makefile with just one rule to start. This rule should generate an executable named `project2.out`.

When you run `make`, the program looks for a Makefile in your current directory. If it finds one, it executes the top **rule**. A rule is a series of steps that executes based on a keyword. Here is an example of a Makefile with two rules:

```makefile
build:
    g++ -o MyProgram main.cpp FileReader.cpp FileDataBase.cpp UI.cpp UIControl.cpp

clean:
    rm a.out
    rm output
```

Listing 3.1: Makefile with two rules.

3.6 Submission

To submit your first milestone, submit your project zip file to the Milestone 1 assignment in Gradescope. For more information about how to package your project and submit to the Gradescope platform, see Appendix C. For details on grading, see Appendix B.

4 Milestone 2: Command-Line Interface

! This is the second part of the project, and is due **after** the first part is due. You should not start this part until you have completed the first part.

4.1 Goals

The first step in developing your command-line interface is removing the code that automatically runs the ten tasks listed in part two of the assignment. You don't have to delete the code for these tasks from the project, but they should no longer execute automatically. Before building the command-line interface, build your project and run it with no arguments to make sure none of the ten tasks execute automatically.

The interface should not do anything other than what the user asks. If the user supplies only the `--help` argument, then simply print the help message and exit. If the user asks to multiply two images using the `multiply` method, then proceed with that method, and then exit.

4.2 Developing The Interface

Now that you have access to your command-line, we can begin work on building the actual command-line interface. Your interface should use the `argc` and `argv` features of C++ to handle command-line arguments, as discussed in class.

This section will discuss what methods you will need to implement in your interface, how arguments are passed to those methods, and the actual specification itself.

4.2.1 Tracking Image

Just like the tasks you completed in Part 2, a user can request multiple image operations to be done one after the other, using the output of the previous step as an input to the next step. In order to support this, your command-line interface will need to keep track of a **"tracking image."** This is what will be written to the output filename once all the image manipulations have been done on this image.

The initial source for this tracking image will be given to you as the second argument (explained more later in **??**). It is recommended to load in the data from the source file into your tracking image first. Then read all the image manipulation algorithms requested by the user, in order. These algorithms will take in the tracking image as an input, and then the tracking image will be set to the output of the algorithm. In other words, each image manipulation algorithm will "act" on the tracking image.

4.2.2 Methods

You will need to implement several methods in your CLI that the user can call. The methods are discussed below.

- `multiply`: This method takes one additional argument, the second image to use in the multiplication process, alongside your tracking image.
- `subtract`: This method takes one additional argument, the second image to use in the subtract algorithm, alongside your tracking image. The first image, the tracking image, will be the top layer. The additional image argument constitutes the bottom layer.
- `overlay`: This method takes one additional argument, the second image to use in the overlay algorithm, alongside your tracking image. The first image, the tracking image, will be the top layer. The additional image argument constitutes the bottom layer.
- `screen`: This method takes one additional argument, the second image to use in the screen algorithm, alongside your tracking image. The first image, the tracking image, will be the bottom layer. The additional image argument constitutes the **top** layer.
- `combine`: This method is similar to what you did in task 9 of Part 2, where three individual files each provide one channel of the resulting image. This method takes two additional arguments, the source for the green layer (the first additional argument), and the source for the blue layer (the second additional argument). The source for the red layer is the tracking image.
- `flip`: This method takes no additional argument, and simply flips the tracking image.

- `onlyred`: This method takes no additional arguments, and simply retrieves the red channel from the image, similar to how you did in task 8.
- `onlygreen`: This method takes no additional arguments, and simply retrieves the green channel from the image, similar to how you did in task 8.
- `onlyblue`: This method takes no additional arguments, and simply retrieves the blue channel from the image, similar to how you did in task 8.
- `addred`: This method adds a certain value to the red channel of an image. This method takes one additional argument, the amount to add to the red channel. This will need to be converted to an integer.
- `addgreen`: This method adds a certain value to the green channel of an image. This method takes one additional argument, the amount to add to the green channel. This will need to be converted to an integer.
- `addblue`: This method adds a certain value to the blue channel of an image. This method takes one additional argument, the amount to add to the blue channel. This will need to be converted to an integer.
- `scalered`: This method scales the red channel of an image. This method takes one additional argument, the amount to scale the red channel. This will need to be converted to an integer.
- `scalegreen`: This method scales the green channel of an image. This method takes one additional argument, the amount to scale the green channel. This will need to be converted to an integer.
- `scaleblue`: This method scales the blue channel of an image. This method takes one additional argument, the amount to scale the blue channel. This will need to be converted to an integer.

Each of the final six operations listed accept an integer as an additional argument. For `addred`, `addgreen`, and `addblue`, you should be able to accept negative integers. You should ensure that the pixel value never falls out of the 0-255 range, clamping as appropriately, as described in subsection 3.2.

4.2.3 Error Handling

Your program should be able to handle commands in the following format:

- If no arguments are provided, or if the first and only argument is `--help`, print the help message.

```
Project 2: Image Processing, Fall 2023

Usage:
	./project2.out [output] [firstImage] [method] [...]
```

  Your help message should be **exactly** the same for the tests to pass. Note that the final line in the help message is indented with a **tab character**, not several spaces or another character. You are expected to use this character as well.

- The first argument will be the name of the output file. If the argument does not end with `.tga`, print "Invalid file name." If this argument is missing, you should proceed with the bullet point above.
- The second argument will be the name of the source file for your tracking image. If this argument is not provided, or if the argument does not end with `.tga`, print "Invalid file name." If the filename is not a real file, then print "File does not exist."
- The next few messages will be for any errors with the arguments of any method:
  - Nonexistent method name: "`Invalid method name.`"
  - Missing an argument: "`Missing argument.`"
  - Invalid file input: "`Invalid argument, invalid file name.`"
  - Nonexistent file input: "`Invalid argument, file does not exist.`"
  - Non-integer argument input: "`Invalid argument, expected a number.`"

For commands that are not invalid (commands where some image manipulation is successfully done), the output of the command does not matter. You can print whatever you like!

4.3 Submission

At this point, you should have a completed command-line interface to your image manipulation program. It should be able to handle variable file names and variable arguments, and should not run anything automatically. This should allow to pass several of the tasks you will be tested on.

Again, Gradescope will be used to test out your command-line interface. There are three areas you will be tested on. Like Milestone 1, there are ten individual tasks, some of which you will add to your Makefile using your new command line arguments, and others will be compiled by Gradescope after you submit your submission. You won't know some of the arguments your program is tested with.

- **Help Message and Error Messages**: Your program should produce the correct help message when requested, and your program should produce the correct error messages when invalid arguments are supplied.
- **Tasks 11-13**: As described below, you will add three new tasks into your Makefile under a new rule named `tasks`. Gradescope will expect that after this rule is called, the images for tasks 11-13 will be generated.
- **Tasks 14-20**: Gradescope will call your command-line interface with variable arguments. Tasks 14-16 will have their arguments shown, while tasks 17-20 will have their arguments hidden. Your program should be able to handle these arguments and produce the correct images.

11. Multiply `circles.tga` with `layer1.tga`.
12. Flip (or, rotate 180 degrees) `layer1.tga`.
13. Subtract `layer2.tga` (bottom layer) from `layer1.tga` (top layer).

Tasks 14-16 will be executed by Gradescope calling your executable with the necessary command-line arguments. For debug purposes, they are listed below:

14. Flip `car.tga` four times.
15. Subtract `layer1.tga` from `car.tga`, then multiply the output with `circles.tga`, then flip it.
16. Given `car.tga`, scale its blue channel by 3, its red channel by 2, and its green channel by 0.

Tasks 17-20 have been hidden from you. Gradescope will call your executable with the necessary command-line arguments.

A Compatibility

Once you believe you have completed the project, ensure that the following compatibility requirements are met. Failure to follow these principles could break your project when it is being graded, and will cause you to lose 10 points.

A.1 Paths

For compatibility purposes, you adhere to the following principles about paths. This applies for all aspects of the project, including the source code itself, and your Makefile.

- **Use forward slashes, not backward slashes**: On some operating systems, especially Windows, backslashes can be used to specify folder names in paths. While this format is supported on some operating systems, this is not supported on all operating systems. Therefore, please use forward slashes.
  - **Bad**: `folder\filename.txt`
  - **Good**: `folder/filename.txt`
- **Use relative paths, not absolute paths**: Absolute paths link to a file on *your* computer, and *your* computer only. Frequently, absolute file paths will include specific details about your computer that will break your program on other people's computers. Therefore, use relative paths instead, which specify a filename relative to your current directory. Your paths should start with `input/`, `output/`, or `examples/`.
  - **Bad**: `C:/Users/rubberduck/College/COP3503C/Project 2/input/car.tga`
  - **Good**: `input/car.tga`

B Grading

Table B.1 explains how the entire project will be graded.

Task / Milestone	Points	Description
Milestone 1 (80 points)		
Tasks 1-10	70	7 points possible per task. Complete task with matching pixels and header. Proportional points for pixel accuracy.
Makefile	10	Makefile development: 5 points for correct executable, 5 points for `make run` completion.
Deductions	0	Any deductions found from the Deductions section deduct 10 points.
Milestone 2 (70 points)		
Command-Line Tests	6	Tests the help message and error messages.
Tasks 11-13	15	5 points per task. Command-line tasks generated by your Makefile.
Tasks 14-20	49	7 points per task. Command-line tasks generated by Gradescope.
Total	150	

Table B.1: Rubric for the entire project.

B.1 Deductions

10 points will be deducted from the final project score if any of the following are true:

- Your submission contains any other files than the `src/` folder and your Makefile. (Explained more in the next section...)

- Your Makefile build command does not use the C++11 standard, builds code from the wrong location, or includes header files.
- Your Makefile is incorrectly named, or uses an improper encoding. Your Makefile cannot be named `Makefile.txt`, `Makefile.rst`, or anything similar, and it must be a plain-text file (ie, not created in Microsoft Word or macOS Rich Text Editor).

No points will be deducted for the following:

- Printing output when running a successful image manipulation (ie, where the user's command input does not fail).
- Warnings appearing when compiling the executable. Be careful, however, as these warnings might indicate that your program will fail under specific circumstances which you might want to be aware of.
- Writing code without comments. You may add helpful comments if you'd like, but commenting or explaining your code is not required.

C Submitting to Gradescope

If you've made it to this part, you might almost be done with your project! That's exciting!! Let's talk about how to actually submit your project to Gradescope. Gradescope is a machine grader that is used to test and grade your submission automatically. The grader has over fifty tests, and conforms to the rubric specified above.

Your submission should be one zip file, containing:

- Any source and header files, placed in a folder named `src/`. There should be only C++ files in here. Delete any object files or other files.
- The Makefile you created to build your project.

You do not need to include `output/`, `input/`, nor `examples/` in your submission. These will be provided in the grading environment for you. An example structure is shown in Figure C.1. `File.h` and `File.cpp` are example alternate source files.

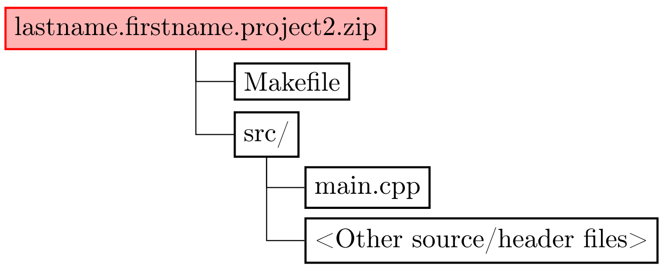

Figure C.1: The appropriate tree structure for your submission.

Do not include any other files in your submission. This is not limited to the following: project images, executables, IDE-specific configurations, PDF files, etc.

C.1 Zipping with mac!

! On mac you may need to zip it via terminal. To do this:

- Open terminal and navigate to your project directory.
- Within your project folder run:
  - zip lastname.firstname.project2.zip Makefile src/

C.2 Gradescope Environment

The Gradescope testing environment consists of four parts:

1. **Deductions**: Testing to see if your submission contains deductions.
2. **Building**: Using your Makefile to build your submission and generate tasks 1-10. If you do not provide a Makefile, your submission will be supplied one, but you will receive a zero for the Makefile in the rubric. This allows your submission to build even if you have not yet completed the part 4.
3. **Tests**: Testing tasks 1-10 for Milestone 1, or tasks 11-20 for Milestone 2. After each test is run, an accuracy comparison is run on the output image.

The testing environment has been created to help ensure each student receives a fair amount of points. Here are some further tips on the testing environment:

- If you submit your Makefile and `src/` folder in an enclosing folder, your submission may not be graded correctly. Ensure that your submission appears as shown above before submitting.
- When testing your command-line interface, Gradescope may (or may not) reference files in an enclosing folder. For example, `./project2.out out.tga input/test.tga flip` is a valid command, as is `./project2.out out.tga test.tga flip`.
- Some TGA files tested by Gradescope are not provided to you.
- Many of the commands tested in Gradescope are shown to the student. However, some of the commands tested are hidden and not accessible by the student.
- Some TGA files tested by Gradescope have lengths and widths other than 512x512.
- If `make` or an individual test takes a significant amount of time, the command will be aborted, and that test will receive zero points. This helps prevent infinite loops.