Project 2:

# Image Processing

COP3503C: Programming Fundamentals II
University of Florida

Laura Cruz Castro, Cameron Brown, Joshua Fox

**Revision 9** — Last updated: March 16, 2023

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!

# 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

This assignment is split into four parts.

- **Part 1** covers reading and writing TGA files. You should be able to write a program to read a TGA file, and write its contents to a new file with a different name.
- **Part 2** covers manipulating image data. You will implement ten different manipulation tasks, each of which use different manipulation methods.
- **Part 3** covers creating a command-line interface to your program. Your program will no longer run the same ten tasks repeatedly, rather, the user will be able to specify the tasks they want to run themselves.
- **Part 4** covers Makefiles, and how they speed up development and use of your program. Here, you will write repeatable commands that will be used to build, and then run/grade your project.

# 2   TGA Files

What is a TGA, or `.tga`, file? It's simply an image file - just like your standard PNG, JPEG, or GIF files. However, the file format supporting TGA files is much simpler and easier to work with, especially when modifying the images through code. In this assignment, you will learn this specification in order to read and create your own TGA files in C++! How awesome is that? It's like your own little Photoshop!

Some operating systems won't let you open TGA files natively (looking at you, Windows), so you will need to install some sort of viewer for them. If you already have a tool installed that lets you open and view these, great. You can tell if you have a tool installed already if you try clicking on one of the .tga files provided and observing if a program is able to show you the file.

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!

Note that not having a viewer installed on your computer will not prevent you from completing the project. However, it will prevent you from seeing the contents of your output images, which may make it more difficult to catch possible errors in your code.

Although it is helpful to generate the images and check whether the output is correct visually, we acknowledge that not all people can see pictures correctly (ie, have an inability ability to perceive or distinguish certain color, or students with visual disabilities). This assignment has been developed with visually impaired students in mind, and can be completed entirely through code.

As you move through the document and begin to work with TGA files, we recommend accomplishing your tasks using images with only a few pixels. For example, rather than testing your code using a 512x512 image, you could use an image of size 2x2 to check if your methods are implemented correctly at the channel level. This setup is more accommodating for screen readers and other assistive devices.

# 3 TGA File Specification

## 3.1 RGB and Pixels

Before jumping into the format, we need to discuss how colors can be represented in computers. Without having a way to represent colors, you can't represent pictures. For example, imagine trying to store an image of a sunflower on your computer. How do you store the golden color of the flower, or the pastel blue color of the sky, or the olive green color of the flower stem?

One of the most common format for representing colors numerically is known as the RGB format. This format uses three numbers (0-255) to store the amount of red, green, and blue in the color. Figure 3.1 showcases some examples of this representation.

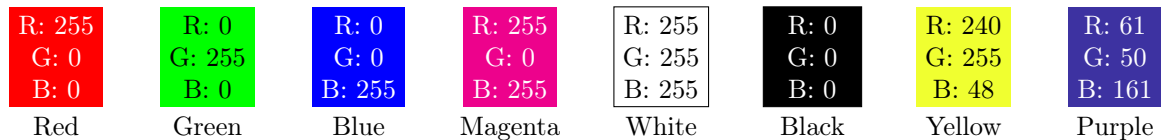| R: 255<br>G: 0<br>B: 0 | R: 0<br>G: 255<br>B: 0 | R: 0<br>G: 0<br>B: 255 | R: 255<br>G: 0<br>B: 255 | R: 255<br>G: 255<br>B: 255 | R: 0<br>G: 0<br>B: 0 | R: 240<br>G: 255<br>B: 48 | R: 61<br>G: 50<br>B: 161 |
|---|---|---|---|---|---|---|---|
| Red | Green | Blue | Magenta | White | Black | Yellow | Purple |

Figure 3.1: Various colors represented in RGB format.

You can see that colors high in red, green, and blue, but devoid of other colors, are unsurprisingly red, green, and blue, respectively. White is made by setting red, green, and blue to the max. Black is made by setting all values to zero. Other colors are made by combining various values of these three elements. If you would like to create your own colors, you can use a color picker tool, found on your computer or online. An example of such tool can be found at https://colorpicker.me/.

In computers, these three numbers are commonly stored as 3 **bytes**, each made up of 8 **bits**. Each byte is able to represent a number between 0 and 255. Note that some file formats store the bytes in the reverse order of blue, green, red, rather than the conventional red, green, blue. This includes TGA - but more on that later!

You should now feel comfortable understanding how a single color can be represented numerically. But, a picture is made up of many colors - how do we represent all of these various colors? This is the concept of **pixel**. Digital photos are made up of a series of these pixels which create a unique image. The **height** and **width** of an image are determined by the number of these pixels in a column and row, respectively. For example, a *1920x1080* image, or *1080p* image, has 1,920 pixels in each row, and 1,080 pixels in each column. The image has a height of 1080 pixels and a width of 1920 pixels. Each pixel has three color attributes: red, green, and blue.

## 3.2 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 3.1.

| File Header | | |
|---|---|---|
| ID Length | 1 byte | Size of the Image ID field |
| Color Map Type | 1 byte | Is a color map included? |
| Image Type | 1 byte | Compressed? True Color? Grayscale? |
| Color Map | 2 bytes | Color Map Origin - Always 0 |
| | 2 bytes | Color Map Length - Always 0 |
| | 1 byte | Color Map Depth - Always 0 |
| Image Specification | 2 bytes | X Origin - Always 0 |
| | 2 bytes | Y Origin - Always 0 |
| | 2 bytes | Image Width |
| | 2 bytes | Image Height |
| | 1 byte | Pixel Depth - Always 24 (for RGB, each 8 bits) |
| | 1 byte | Image Descriptor |
| File Data | | |
| Image Data | Variable | The data of the image, ie, each pixel. Every pixel is made of 3 bytes, representing BGR data. Note that the first pixel in this series represents the pixel in the bottom left corner. The final pixel is the pixel in the upper right pixel. |

Table 3.1: 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. Depending on the specifics of the file (or your scenario), some of those variables may have a value of zero, or they may be non-zero. (In the case of the TGA format, some values in the header were once very important, but nowadays are not used—the format still has them for compatibility reasons.)

**From the header description, the image width and image height are at a 12 byte offset and a 14 byte offset, respectively, from the beginning of the file.** You may find it helpful to read each piece of data in the header into a structure. Then, once the header has been completely read, you can go about using it for whatever purposes you have in mind. A possible structure for the header is shown in Listing 3.1.

```cpp
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 3.1: 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 3.2.**

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

Listing 3.2: Showcasing various methods of printing information about a character in C++.

We've covered reading headers into memory, but how do we construct and write a header for an entirely new image? Here's the great news: **as long as the images are the same width and height, you can simply steal the header from either image and use that in the resulting image.** In this project, all source images that will be used have headers that are compatible with each other. Note that if you need to change the width and height of the resulting image, you will need to modify this property in the resulting image's header.

## 3.3   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.** Each of those bytes will contain a value from 0-255, representing the intensity of that color in the pixel. Conveniently, an `unsigned char` is able to store values in the range of 0-255! So if a file had a size of 200x300, it would contain 60,000 pixels, each of which contains 3 bytes of data. Figure 3.2 shows the start of the image data of an example TGA file, where pixels are stored in series.
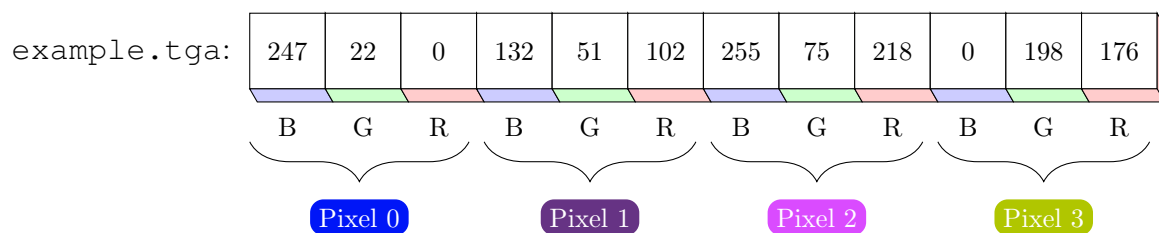


Figure 3.2: Series of pixels stored in the `example.tga` file. Pixels are stored in series, and each contain a B, G, and R value.

To store these values in your program, you could make a 1-D array/vector with 180,000 entries, where there are three new entries in the array for every pixel (being the B, G, and R values for each pixel). Or, you could make a 2-D array/vector of 60,000 entries, with each entry being a vector storing three elements, the B, G, and R values of the specific pixel. The approach you take should depend on what you feel comfortable with.

What about the order of the pixels themselves? In many image files (including TGA files), the first pixel in the file represents the bottom left corner of the image, while the last pixel represents the top right corner of the image, as shown in Figure 3.3. If you read, store, and write the pixel data in the same order, you don't really have to worry too much about this. If you wanted to copy data into a particular part of the image, however... that can be a bit tricky. For example, to copy some 2x2 image into the top left corner would require you change pixels 16, 17, 24, and 25.
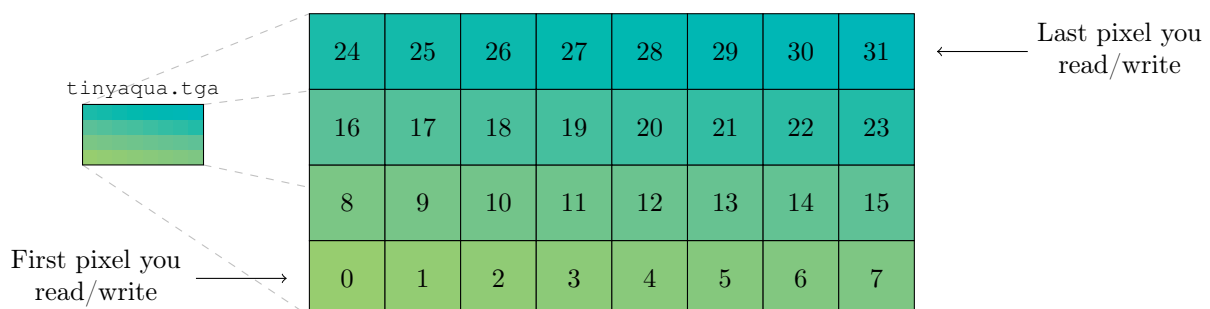


Figure 3.3: Order in which pixels are retrieved and written to in a TGA file.

So, to summarize: The file contains a header, which is 18 bytes in length. Stored within those 18 bytes are pieces of information describing the image content—the width and height of the image, how the color data is stored, and so on. All you need from the header is the width and the height. However, when writing a file, you should provide **all** the header data, whether you are using it or not. Because of this, you should store this data along with the image data itself.

# 4   Part 1: Reading and Writing

Great - you should now have a good understanding of TGA files. Now, we encourage you to try reading and writing one of the `.tga` files we give you. While this isn't worth any points on its own, this part serves as good practice for the next section. Specifically, try:

1. Reading the header and pixel data from one of the TGA files provided to you

2. Storing the information in data structures for your header, pixels, and overall image.

3. Writing the contents of those data structures to another TGA file with a different name.

If you do not write the binary data in the correct format to the `.tga` file, your viewer will not be able to open the file. Try opening a file from the `input/` folder to be sure your viewer is not broken.

Once you feel confident reading and writing files, then you should be ready to move to Part 2.

## 4.1   Implementation Tips

As you move throughout this project, we'll offer some small bits of advice to help you along your journey. Like a great narrator, we'll only complement your adventure.

- **If you try opening a `.tga` file created by your program and see that the file cannot be opened by your viewer, or that your viewer crashes when attempting to open the file, your code implementation is likely broken.**

- Start by planning out the data structures you will need, and how you'll store those objects. Planning out the design early makes the later implementation much easier.

- If your image looks like white noise (ie, pixels are random colors), you may have read the header incorrectly. If you can see the subject of the image, but the image colors are slightly off, make sure you read the pixel data in BGR format and that you are using the right data types to store your binary data.

- Remember to read the file in binary mode. This isn't default, you need to pass an argument to your `fstream` constructor.

- Use classes and structs. Do not try to not use them - you will save yourself more time by declaring these structures up front and using them throughout your project.

- Pass objects around by reference, not by value, or your program might become very slow and memory inefficient.

- If you see a segmentation fault, or if you're only able to read zeroes, this may mean that your program is not able to see the image file you are referencing. You can use the `is_open()` function to see if the file you are referencing has been successfully opened by your program.

- A BGR value of (0, 0, 0) represents a black pixel. A BGR value of (255, 255, 255) represents a white pixel. If your image is all black or all white, you might be reading all zeroes or all max values.

- If your output image has vertical or horizontal lines which are all the same color, then your loops are likely incorrect. You might be printing the same row/column of pixels repeatedly, without incrementing to the next row/column.

# 5 Part 2: Image Manipulations

If you've used Photoshop or a similar tool, you're likely aware of image manipulation. Tools in these programs allow you to change the pixel data in image data using common algorithms. These algorithms allow you to blend colors together, highlight certain colors, or even make objects disappear. You will be implementing some of these algorithms as part of this project.

## 5.1 Algorithms

These algorithms are described in the Table 5.1. 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 5.2.

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 Table 5.1 three times per pixel, one time per channel.

| 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 5.1: Formulas for common manipulation algorithms.

Note that the "Overlay" method shown in Table 5.1 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.

## 5.2 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. As an example, imagine two pixels, one with a green value of 100, and one with a green value of 200. If you attempt to add the green components together, you will end up with a value of 300, which will overflow to $300 - 255 = 45$.

- **For operations involving addition**, you will need to clamp values. When completing an operation, you will likely need to use a data type that can support a large range of values, such as `int`. Then, after the operation is complete, all values lower than zero become zero, and all values greater than 255 become 255.

  Failure to clamp can lead to incorrect values which may not be noticeable at first. Figure 5.1 illustrates this: subtracting the two green values results in a negative number, which would typically overflow, resulting in an incorrect result for this task. However, clamping the green value (by making the negative result zero) leads to the correct answer.
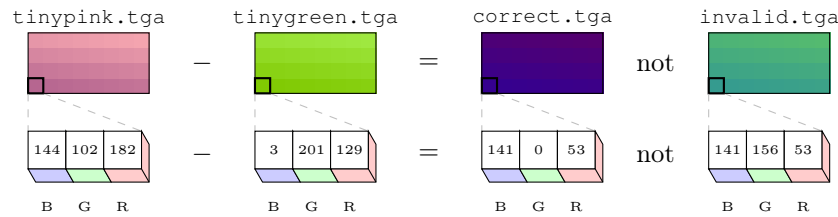
Figure 5.1: When clamping is not done, an entirely different image can arise.
Notice how the green value is appropriately clamped to 0 in the correct image,
while it overflows to $102 - 201 = -99 + 255 = 156$ in the incorrect image.

- **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).

## 5.3 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...?

## 5.4   Implementation Tips

Again, let's talk about some tips that might help you out with this part of the project:

- Re-read over the implementation tips listed in subsection 4.1.

- Be careful when casting values to specific types. If you're completing an operation and then casting to a specific type, remember to wrap the operation in parentheses, as in `(unsigned char)(a + b)`, not `(unsigned char)a + b`.

- Make sure you know which file is the "top layer" and which file is the "bottom layer" in operations where that matters.

- Make separate functions for each operation. While you could just write all the operation code in your `main()` function, using separate functions will likely make Part 3 (the next part) easier to understand.

## 5.5   Testing

After you have completed the tasks, you'll want to make sure that your output matches the expected output we actually give you. To do this, you can use an external program to compare your output with our output, or you can write your own code to do so. We typically recommend both methods, as the external tools will always be correct in their analysis, while your tool will likely give you more helpful information about which pixels are wrong in your output.

Your images and the example images given must match exactly. This includes the header and all pixels stored in both images. **Looking at two images side-by-side is not a valid testing method.**

### 5.5.1   External Comparison Tools

To tell if two files are different, you can use command-line tools installed with your operating system.

- **Windows**: In your terminal or PowerShell, you can use the `fc.exe` program to tell whether two binary files are different. Use the `/b` flag followed by the name of each of the binary files. If there are any differences, the program will print them to you. Otherwise, it will tell you that no differences were found.

- **macOS, Linux**: Use the `diff` command. This command will simply output whether the two files you provided as arguments differ in content. If the program prints no output, then the two files provided as arguments have the same content.



(a) Windows, using `fc.exe`



(b) macOS/Linux, using `diff`

Figure 5.2: Comparing the contents of two files, in Windows and macOS. First, the contents of two equivalent files are checked. Then, the contents of two different files are compared.

### 5.5.2   Writing Tests

If your files differ in content, you might have noticed that the commands aren't super helpful in telling you *how* to fix your program. That's where you come in!

If your file differs in content from the examples provided, you will probably want to write a small extension to your program that shows you which pixels are different, and how they are different. This program doesn't need to be too complex — but it should help you debug faster by showing you which pixels are incorrect, and how they are incorrect. This program should load both files into memory, and compare their pixels against each other, letting you know when it finds a difference. Writing this step is not required at all, so build it however you think it will help you most.

# 6   Part 3: Command-Line Interface

You should now feel comfortable reading TGA files, manipulating their contents, and writing the contents to new TGA files. However, the tasks you've completed can't be easily changed by the user — they have to open the source code, change which algorithms are executed themselves, and then re-build the program. That's complicated, ripe for new bugs, and is hard for users who may not be familiar with C++.

To solve this, you're going to develop a command-line interface to your program. This will allow the end user to provide different arguments to your program, which will change the behavior of your program on the fly. This allows the end user to change the manipulation methods used, the input files used, or the name of the output file, all without needing to re-build.

## 6.1   Accessing the Command-Line

First, we will need to understand the different ways to access your command-line, in order to test this part of your program. There's nothing you need to build or test yet, this is just meant to show you how to access the command-line in your editor of choice.

Typically, the command-line is accessed through a dedicated terminal app. However, most modern development environments today have embedded terminals that allow you to test a command-line app from the IDE itself, rather than needing to use an external program. Furthermore, many IDEs even support debugging of programs running with specific command-line arguments! We'll take a look at how you can access these environments in the following subsections.

### 6.1.1   CLion

Typically, programs are run in CLion by clicking the green triangle (▶) in the upper right-hand corner of the program, which builds *and* runs the project. However, this button does not allow us to input arguments into our program before running it. Therefore, we should instead use the hammer (🔨) to only build the project. Then, navigate to the "Terminal" menu at the bottom of CLion — this is your command-line. The specific executable built by your program will be given the name listed in your CMakeLists.txt file. You can run your program by typing `./cmake-build-debug/executablename arg1 arg2 arg3 ...`
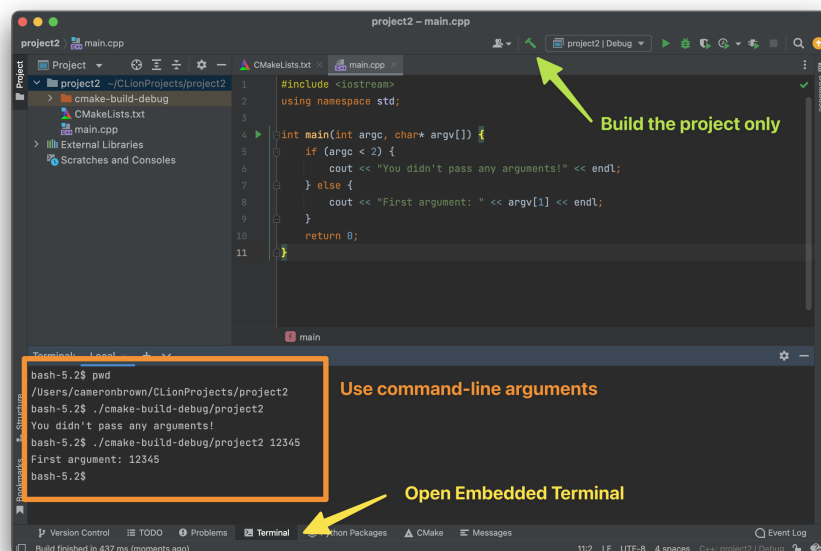


Figure 6.1: Accessing the command line through CLion.

To enter debug mode while specifying command-line arguments, in your menu bar, go to Run ⇒ Edit configurations... ⇒ Program arguments, and input the list of arguments you'd like to have called in debug mode.

### 6.1.2 Visual Studio

You can access the Terminal in Visual Studio by going to View ⇒ Terminal. The executable file is typically stored in Visual Studio under a folder named `arm64\Debug` or `amd64\Debug`, or something similar. You can call this executable and optionally pass arguments to it, as shown in Figure 6.2.
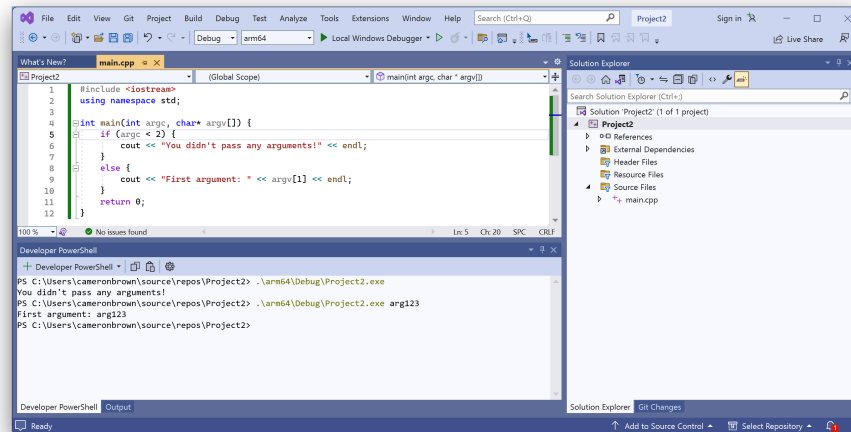


Figure 6.2: Accessing the command line through Visual Studio.

If you would like to open debug mode and still pass arguments in, you can add these arguments in your project settings, as shown in Figure 6.3.
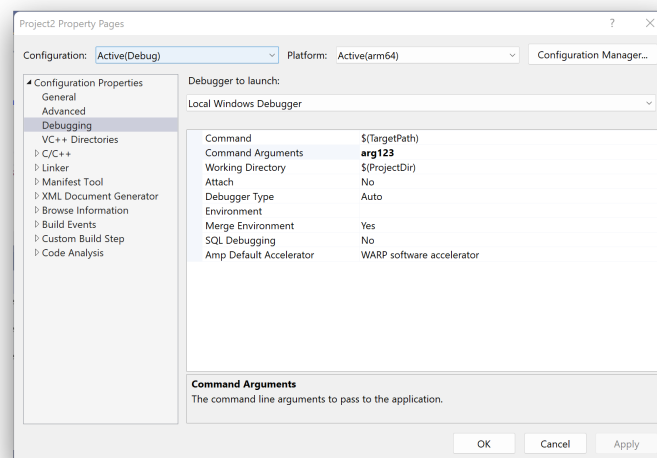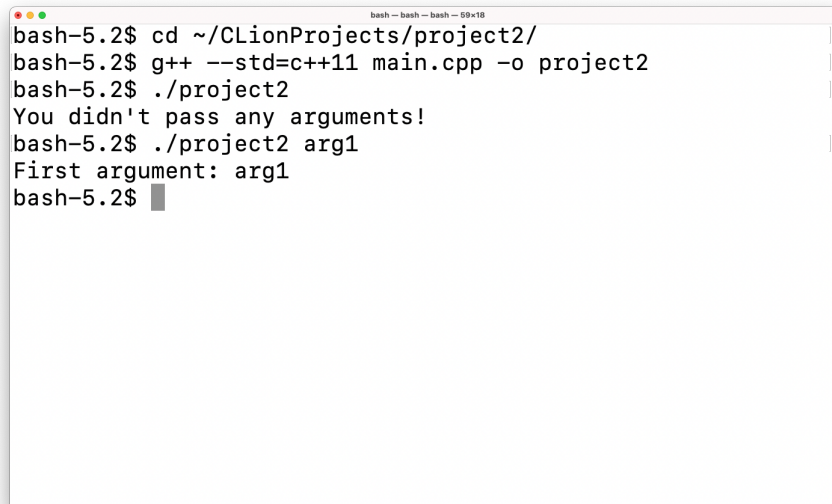


Figure 6.3: Altering the command-line arguments used in Debug mode. The option that is modified in the project properties is "Command Arguments".

### 6.1.3 Terminal

If you would like to test your command-line program using your Terminal, open up your terminal program of choice. On Windows, Windows Terminal is a good option, while on macOS, Terminal.app is a good option. After opening the Terminal, navigate to your project directory using the `cd` command. Then, use `g++` to build your project files using the C++11 standard. Finally, run the executable with the desired arguments by typing `./executablename arg1 arg2 arg3 ...`

Figure 6.4: Accessing the command line through Terminal on macOS. Terminal applications on other operating systems should have a similar interface.

## 6.2 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.

## 6.3 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.

### 6.3.1 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 subsubsection 6.3.3). 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.

You can keep track of the tracking image by storing a variable representing your image before the user calls CLI arguments. This variable could be an image class you have written, it could be a vector of pixels, whatever makes sense in the context of the program. Ensure that it can be updated whenever a new image manipulation method is run.

Note that this method of implementing the command-line interface is not necessarily required, but the specification was set up to support this style of developing your command-line interface.

### 6.3.2   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 red 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 red 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 red 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 red channel. This will need to be converted to an integer.

### 6.3.3   General Specification

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. The help message to print is shown as the first command example shown in Listing 6.1. Your help message should be **exactly** the same for the tests to pass.

- The first argument will be the name of the output file. If this argument is not provided, or if the argument does not end with `.tga`, print "Invalid file name."

- 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 arguments describe the first image manipulation method.

  - The third argument will be the name of the first image manipulation method. If this argument is not provided, or if the method does not exist, print "Invalid method name."

  - If the first image manipulation method selected requires additional arguments (such as the "multiply" method, which requires the name of a second file to multiply the first with), then those arguments will be provided after the name of the manipulation algorithm. If the arguments are not provided when they should be, print "Missing argument." If the first method selected does not require additional methods (such as the "flip" method), then any following arguments will be related to the next image manipulation algorithm.

    * If the method expects a filename argument, and the argument does not end in `.tga`, print "Invalid argument, invalid file name." If the file does not exist, print "Invalid argument, file does not exist."

    * If the method expects a number, but receives something other than a number, print "Invalid argument, expected number."

- After the first image manipulation method arguments are read, any additional arguments should represent more steps in your program. Unlike the first step, these methods act on the tracking image, and therefore one image supplied to the method will be the output of the previous step.

  - The first argument of successive methods is the name of the image manipulation method. If the method does not exist, print "Invalid method name."

  - Any additional arguments required by the method will be provided next. If an argument is missing, print "Missing argument."

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

### 6.3.4   Examples

Listing 6.1 shows some examples of what your program should output given various arguments.

```
bash-5.2$ ./project2.out --help
Project 2: Image Processing, Spring 2023

Usage:
    ./project2.out [output] [firstImage] [method] [...]
bash-5.2$ ./project2.out
Project 2: Image Processing, Spring 2023

Usage:
    ./project2.out [output] [firstImage] [method] [...]
bash-5.2$ ./project2.out out.tga car.tga multiply part1.tga
Multiplying car.tga and part1.tga ...
... and saving output to out.tga!
bash-5.2$ ./project2.out out.tga car.tga multiply part1.tga screen part2.tga
Multiplying car.tga and part1.tga ...
... and subtracting part2.tga from previous step ...
... and saving output to out.tga!
bash-5.2$ ./project2.out out.tga car.tga flip flip flip flip
Flipping car.tga ...
... and flipping output of previous step ...
... and flipping output of previous step ...
... and flipping output of previous step ...
... and saving output to out.tga!
bash-5.2$ ./project2.out out.tga car.tga scalegreen 5 addred 100 scaleblue 0
Scaling the green channel of car.tga by 5 ...
... and adding +100 to red channel of previous step ...
... and scaling the blue channel of previous step by 0 ...
... and saving output to out.tga!
bash-5.2$ ./project2.out out.tga red.tga combine blue.tga green.tga
Combining channels of red.tga, blue.tga, and green.tga ...
... and saving output to out.tga!
bash-5.2$ ./project2.out out.tga
Invalid file name.
bash-5.2$ ./project2.out out.tga part2.tga weirdmethod
```

```
Invalid method name.
```

Listing 6.1: Example successful outputs given various arguments.

### 6.3.5  Implementation Tips

Here are some tips to assist you in your implementation of the command-line interface:

- Don't forget to handle `--help` and passing no arguments. That should be handled first.

- Consume arguments one by one; you should only need to look at each argument once, in order. If an argument is missing, you can print an error right away.

- You cannot compare `char*` to `std::string` directly. Instead, use `std::strcmp(char* value, std::string otherValue)`, located in the `<cstring>` header, which will return zero if the two string representations have the same value. Alternatively, you can cast construct an `std::string` from the argument.

- Segmentation faults commonly occur from trying to access an array index that does not exist.

- When handling receiving numbers via the command line, you can use `std::stoi`, which will convert a string representing a numeric value to an `int`. It will throw a `std::invalid_argument` exception if no conversion can be performed (aka, if the argument was not a number).

## 6.4  Testing Your Interface

Once you feel confident with your interface, test it as a replacement for your tasks! Try running task 1 from subsection 5.3 using the command-line, and ensuring that the output file is the same as the example task 1 file. This might look like:

```
bash-5.2$ ./project2.out output/part1.tga input/layer1.tga multiply input/pattern1.tga
```

Listing 6.2: Running task 1 through the command-line.

For best results, you should ensure that all ten tasks can be executed through the command-line. Furthermore, ensure that your program reacts appropriately when you supply invalid or missing arguments.

Currently, to test your tasks you have to re-type the executable name followed by a certain set of command-line arguments to run each task. In the next part of the assignment, we'll use a Makefile to store these commands in a file. This will allow us to run commands that generate all ten parts of the project using just the command line.

# 7  Part 4: 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 be writing a Makefile with two rules, one rule to generate an executable named `project2.out`, and another rule to run all tasks using different command-line arguments. This executable will be generated by `g++`, which will be the command you reference in your first rule. Your second rule will call your executable multiple times with the appropriate command-line arguments to complete all tasks. You can learn more about Makefiles below.

## 7.1  Introduction

Compiling from the command-line interface (CLI) can be a much faster way to build a project. It's not without its drawbacks, however. Consider the following example, where you have a file, `main.cpp`, that you want to build. A simple command to build it might be (for GCC):

```
bash-5.2$ g++ -std=c++11 -o MyProgram main.cpp
```

In this example:

- `g++` is the executable, and everything else is an argument sent to that program.

- `-std=c++11` sets which version of the C++ standard to use (C++11, in this case). Other values could be `c++14`, `c++17`, or other new versions when they are released.

- `-o <file>` sets the name of the output to whatever you specify in `<file>` (in this example, the result is `MyProgram`). If nothing is specified, the file produced has `a.out` (or `a.exe`, on Windows) as the default name.

`main.cpp` is, of course, the file that you want to compile and ultimately turn into an executable. So far so good. But, what if you wanted to add some additional files? Instead of just `main.cpp`, let's say you have `main.cpp` and a class called `FileReader` which is split into `FileReader.h` and `FileReader.cpp`.

The command to build this might now change to be this:

```
bash-5.2$ g++ -o MyProgram main.cpp FileReader.cpp
```

Still not so bad, right? What if you then added classes called `FileDatabase`, `UI`, and `UIControl`? You'd again have to adjust the command:

```
bash-5.2$ g++ -o MyProgram main.cpp FileReader.cpp FileDataBase.cpp UI.cpp UIControl.cpp
```

Still not the end of the world, but... having to type that every time could be a bit tiresome (not to mention how easy it gets to forget a single file from the list). What if you then added a class called `Stopwatch`, and another called `UserPreferences`, and another...

A better way to do this would be to save that command in a file, and then call on that file to execute the command. That way, you only have to write the information once, and never need to worry about forgetting any part of it. The file that you would store this in is called a Makefile.

## 7.2   Using Makefiles

A makefile is a plain text file (very commonly just named `Makefile` with no extension) which can contain instructions to build your project (and possibly do a lot of other things). In order to execute these instructions you can simply run the following command from a directory that contains a makefile:

```
bash-5.2$ make (macOS/Linux)
bash-5.2$ mingw32-make (Windows)
```

Listing 7.4: Invoking `make` from the command-line.

Note that the command used to run your Makefile varies between operating systems, and it might be neither `make` nor `mingw32-make`, especially if you use an advanced setup. This assignment is only going to scratch the surface of Makefiles and how to use them.

## 7.3   Creating a New Document

You will need to write your Makefile in a file named `Makefile`, with no extension. The easiest way to make this file is to use your terminal to make this file. Remember to change directories before making the new file, so you know where it is!

```
bash-5.2$ touch Makefile (macOS/Linux)
bash-5.2$ type nul > Makefile (Windows)
```

Listing 7.5: Creating a Makefile using the command-line.

If you want to use your computer's user interface, that works as well. You can create makefiles similarly to how you create any other text file:

1. Create a new text document using your file explorer. On Windows, right click and choose New ⇒ Text Document. On macOS, launch TextEdit.app, create the document in the appropriate folder, and choose Format ⇒ Make Plain Text (otherwise, it will be a rich text format).

2. When asked to save the file, remember to use no extension. You may need to enable hidden file extensions in your operating system to see if your file has an extension before submitting it. This can be done in Windows by selecting "File name extension" in the View tab of File Explorer. This can be done in macOS by opening Finder, navigating to Preferences ⇒ Advanced, and then enabling "Show all filename extensions."

3. If your file has the `.txt` extension, remove it before submitting.

## 7.4 Structure

Great, you should now have a text file to write your Makefile in. Let's begin writing your first Makefile.

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:

```
1  build:
2      g++ -o MyProgram main.cpp FileReader.cpp FileDataBase.cpp UI.cpp UIControl.cpp
3
4  clean:
5      rm a.out
6      rm output
```

Listing 7.6: Makefile with two rules.

Here, we can run either rule by calling `make` followed by the rule name— for example, `make clean` would run the two `rm` commands. As mentioned before, emitting the name of a rule executes whichever rule comes first in the file.

### 7.4.1 Wildcards

The `build` command shown Listing 7.6 is great, but we have to keep updating it as we add more and more C++ files. Luckily, there's a `wildly` better way to do that, using **wildcards**. These are expressions using the `*` keyword to substitute the place of other terms. For example `*.c` targets any filename ending in `.c`.

Let's use this to update our build command from Listing 7.6:

```
1  build:
2      g++ -o MyProgram *.cpp
```

Listing 7.7: Makefile using a wildcard.

This looks much better, and will update whenever we add more C++ files to our project.

### 7.4.2 Appropriate Files

There are two notes about the files you reference in your Makefile.

1. If you need to reference files in a given directory, you can use `dirname/filename` to reference that file. This works for wildcards, too, but note that this can be trickier. Wildcards matching directories and subdirectories can be confusing. In this assignment, you shouldn't have too many levels of nested directories in your submission, so this won't be an issue.

2. You should not use header files in your compiler command. Headers do not need to be compiled, only the implementation files do. After the implementation files are compiled, the header files will be appropriate linked with your source.

## 7.5 Writing your Makefile

For this project, you will need to write two rules in your Makefile. The default (first) rule in your Makefile should build your project. For this rule, note that some characteristics we want from the build command include:

- The output executable should be named `project2.out`.
- The C++11 standard should be used.
- You should compile all of the files relevant to your project.

The second rule in your Makefile should run all ten tasks listed in subsection 5.3, in order. The commands in this rule should produce the appropriate output files using specific command-line arguments with your executable. There should be around twelve individual commands under this rule (ten for each step, except for task 8, which will have three individual calls to your executable). **This second rule should be named `run`**. One component of your project grade will be counting the number of correct files after calling `make run`.

Your `run` rule should call the executable using command-line arguments just as you did in part three.

```
1  run:
2      ./project2.out part123.tga input/infile1.tga multiply input/infile2.tga
```

Listing 7.8: An example `run` rule which completes one task.

# 8 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.

## 8.1 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`

## 8.2 Testing on Unix

Your code will be run and graded in a Unix environment. Specific operating systems are out of the scope of this course, but generally, Unix refers to a Linux-like operating system. Most importantly, **not Windows**.

You should ensure that your code is able to run in a Unix environment before submitting it. The recommended way to do this is to make a pretend submission (as described in section 10) and send this submission to a Unix computer. We recommend testing your code on the CISE computers in the CSE building on-campus, or using the CISE Thunder virtual machines.

### 8.2.1 Testing on CISE Computers

The CISE department at the university hosts a suite of Ubuntu servers which you can use for personal projects, development, and research. You can use these computers to test if your submission will be graded correctly.

If you don't already have an account with the CISE department, you will need to make an account, as listed on this page (you will need to sign in with your Gatorlink to view the page). Your account approval might take a few days to process, so we recommend doing this step early, even if you haven't finished the project.

After you have access to a CISE account, you can test your code. To ensure you can signin, try using SSH to enter your virtual machine:

```
bash-5.2$ ssh mygatorlink@thunder.cise.ufl.edu
```

Listing 8.1: Entering the virtual machine using SSH.

If you are able to enter your virtual machine after entering your Gatorlink password, you're all set! Next, let's move your files over to your virtual machine. You can use a program like `sftp` or `scp`. In our examples, we'll use `scp`:

```
bash-5.2$ scp lastname.firstname.project2.zip mygatorlink@thunder.cise.ufl.edu:project2.zip
```

Listing 8.2: Copying the project zip file to the remote machine using `scp`.

If this command does not produce any errors, you should have successfully transferred your project to your virtual machine, under the name of `project2.zip`. Congrats! You should also transfer the `input/` and `examples/` folders over as well so your program can reference these files when running.

Now, re-enter your virtual machine using SSH, as shown in Listing 8.1. Unzip your project using the `unzip` command and ensure that all files are present. Move your folders into the right structure, and then run `make`. Ensure that your executable builds without error. Run the executable using the proper command line arguments, and ensure that there are no differences between the files appearing in your `output` folder and those in the `examples/` folder using the `diff` command as shown in Figure 5.2.

If there are no differences, then you've completed the project.

# 9 Grading

Table 9.1 explains how the entire project will be graded.

| Standard Points | | |
|---|---|---|
| Task | Points | Description |
| Tasks 1-10 | 80 (8 points each) | Successful completion of task. All pixels must match in order to receive 8 points for a task. 4 points are awarded if at least 95% of the pixels of the image match. If less than 95% of the pixels match, then no points are awarded. |
| Makefile | 10 | Successful development of a Makefile. Points are awarded if the Makefile matches all specifications outlined in the document. If the Makefile successfully produces an executable but does not meet all requirements, not all points may be awarded. |
| Command-line Interface | 10 | Successful development of a command-line interface, according to specifications. If the command-line interface does not work, or fails to have an effect on the program, no points can be awarded for this section. Points can still be awarded for other components if these components successfully complete when the built executable is run with no command-line arguments. |
| Deductions | 0 (up to -10 possible) | Any deduction listed in the Deductions section will result in a removal of 10 points. |
| Total | 100 | |

Table 9.1: Rubric for the entire project.

## 9.1   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 paths are incorrect. This includes using absolute paths, or using backward slashes. Please see the Paths for more information.

- Your project takes more than 2 minutes of time to complete all ten tasks.

- Your output executable was not named `project2.out`.

- 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).

- Your output filenames are formatted incorrectly. The general format is `part#.tga`: they should be similar to `part5.tga`, not `part_5.tga`, `part 5.tga` or `PART5.tga`.

- Your submission to Canvas is not named **lastname.firstname.project2.zip**, or you did not submit a zip file. Canvas sometimes adds numbers to the end of filenames, this is okay, and will not result in a loss of points given the general convention is followed.

- Including `Windows.h` in your program, which will cause a compilation error on any computer that isn't running Microsoft Windows.

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.

# 10   Submission

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 Canvas.

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 10.1. `OtherFile.h` is an example alternate source file.

```
lastname.firstname.project2.zip
├── Makefile
└── src/
    ├── main.cpp
    └── OtherFile.h
```
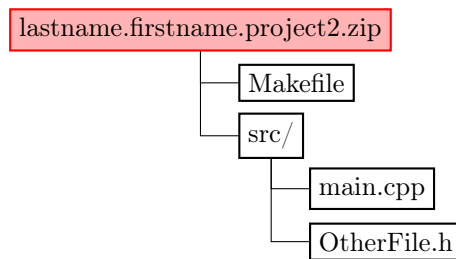
Figure 10.1: The appropriate tree structure for your submission.

The submission should be named **lastname.firstname.project2.zip**, and should be submitted on Canvas. Failure to follow this convention may result in loss of points. Canvas sometimes adds a number to the zip file name, this is okay.

> **!** 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.

# A   Document Revisions

Table A.1 shows the edits have been made to the document since its initial release.

| Revision | Date | Section | Changes |
|---|---|---|---|
| 9 | March 16, 2023 | section 6 | Introduction of "tracking image" idea and clearer statements about how to consume arguments. |
| 8 | March 16, 2023 | subsection 5.3 | `pattern2.tga` was incorrectly labeled as the "top" layer in task 4. |
| 7 | March 16, 2023 | subsubsection 6.3.3 | The "overlay" method now explicitly states it takes the "top layer" first. **??** was added to show examples of command ordering. |
| 6 | March 15, 2023 | subsection 5.3 | Task 2 now correctly states that `layer2.tga` is the bottom layer, and `car.tga` is the top layer. |
| | | subsubsection 6.3.3 | Further information was added for the "combine," "onlyred," "onlygreen," and "onlyblue" algorithms. |
| 5 | March 9, 2023 | subsubsection 6.3.3 | The spec no longer requires checking if the output file is an already existing file. |
| 4 | March 7, 2023 | section 10 | Figure 10.1 was added. Added that neither `output/`, `input/`, nor `examples/` need to be submitted. |
| 3 | March 6, 2023 | subsection 9.1 | The reference to an extra credit task was removed. |
| 2 | March 3, 2023 | subsubsection 6.3.3 | The CLI spec for the "screen" method now requires taking in the "bottom layer" first. |
| 1 | March 1, 2023 | — | Initial release. |

Table A.1: Changes that have occurred in the document since its initial release.