Introduction

This week's lab uses images to test your general computer science thinking skills you've developed so far and is a great way to combine many of the things we've learned so far.

Lab Objectives

By successfully completing today's lab, you will be able to:

- Conduct a brainstorming session with one or two individuals in the class.
- Create a planning document outlining your solution to the code.
- Implement a programming project that will produce images of the flags of Poland, Netherland, and Italy.

Prior to Lab

- You should be familiar with input and output, conditionals such as if, if/else, for loops, and arrays.
- This lab corresponds with zyBooks Chapter 4 and 5, especially 5.9 and 4.7.

Deadline and Collaboration Policy

- Part 2 of this assignment is due by 11:59 PM on Tuesday (10/2/2019) via Canvas.
- Part 3 of this assignment is due by 11:59 PM on Friday (10/4/2019) via Canvas.
    - More instructions on what and how to submit are included at the end of this document.
- You should write your solutions to Part 2 and Part 3 of this lab by yourself. In this lab, you will talk at a high level with others in your lab about a solution. However, you will create all code by yourself and **you should not talk about specific code to anyone but a course instructor or lab teaching assistant.**
- Your zyBooks chapters and lecture slides are available resources you can use to assist you with this lab.
- For this lab, when you talk with a TA for help, you must present your planning document! You should be able to use your planning document to help explain your code and what you're working on and having trouble with. This should hopefully make the process easier, and give you better feedback.

Lab Instructions

This week, it's time for a multimedia lab. We'll be making flags in image files!

Images

Images (e.g. digital photos) consist of a rectangular array of discrete picture elements called **pixels**. The height of the image in pixels is the number of pixel rows, and the width is the number of columns, or pixels in each row. An image consisting of 200 rows of 300 columns (or pixels per row) contains 300 x 200 = 60,000 individual pixels. A color image requires 3 bytes per pixel or 180,000 total bytes!

PPMs

In this lab, we'll be working with a file format called **PPM**. The Portable PixMap (.ppm) format is a particularly simple way used to encode a rectangular image as an uncompressed data file. A ppm image consists of two components: A header, and pixel data. Let's break down how these work.

The Header:

- A label to identify the file format as a color ppm file ("P6"), followed by a single newline character ('\n')
- The width of the image in pixels,
- The height of the image in pixels,
- The maximum possible pixel color value (in our case this will always be 255),
- And finally, the header ends with a single newline character ('\n').

The header of a color ppm image that is 4 pixels wide and 5 pixels high has the following format:

Example PPM header

```
P6
4 5 255
[PIXEL DATA]
```

The next line after the 255 is where the pixel data begins. You can open the .ppm file using vim or some other editor, and you'll be able to see and makes sense of the header; but since the pixel data is binary, the rest of the file after the header will look like a bunch of unreadable characters.

Why would it look like unreadable data? That's because pixel data is written to a file as an unsigned char (which is eight-bits, or one-byte), so that we can preserve raw binary values defining the color of each pixel. Since an RGB pixel can represent each color with a value from 0 to 255, an 8 bit value is perfect for us to use, as $2^8 = 256$. This way, when your computer tries to display the image, it has a much simpler time and can simply read each color as a raw value.

The Pixel Data:

- Each pixel consists of 3 one-byte values of type unsigned char
- Data is listed starting in the top left of the image and continues to the right, then down to the next row.
- NO spaces, tabs, newlines may be embedded in the file, or your program will directly interpret the value of whatever character was included, because we're using raw binary data.
- Pixels use red/green/blue (or RGB) image encoding.[1]

<u>Using RGB:</u>

- The three bytes of each pixel represent the color intensities of the:
  - red component (255, 0, 0) is bright red It looks like this
  - green component (0, 255, 0) is bright green It looks like this
  - blue component (0, 0, 255) is bright blue It looks like this
- Colors are additive
  - red + green = bright yellow (255, 255, 0) It looks like this
  - red + blue = magenta (purple) (255, 0, 255) It looks like this
  - blue + green = cyan (turquoise) (0, 255, 255) It looks like this

[1] RGB is a very simplistic representation of color that is by far the most common usage. While it can't represent every color that our eyes can actually see, it does a pretty good job of doing what is needed for most purposes, but there are other standards that exist and are starting to become more common in every-day devices.

- red + green + blue = white (255, 255, 255)
- When red, green, and blue are all the same value that is less than 255, a gray "color" is produced.
  - (25, 25, 25) would be a light shade of gray
  - (200, 200, 200) would be a darker shade of gray

Creating Our Image

Normally, there's some processing done to ensure that what you want to write to the terminal is correctly formatted. For example, when we use the integer format specifier, %d, the program has to convert the integer value (say, a value of 4) to an ASCII character (the character '4'), which is what your computer can display.

This is why we are using an unsigned character to write our values. The %c format code ensures that the program will not convert the numerical value we give it (as it would normally)[2], and instead just outputs the value directly.[3] Knowing this, we can write the following statement to write a red pixel:

```
printf("%c%c%c", 255, 0, 0);
```

Producing Files

Next, you'll need to know how to make your program create the image files. Since we haven't covered working directly with files yet, we'll use output redirection. Output redirection takes what a program is writing to the terminal and sends it into a file instead (just like the | operator takes the output of one command and inputs it into the next when using commands). This is useful for what we want to do, as shown in the example below.

```
ls
a.out           flags.c

./a.out > poland.ppm
What country's flag do you want to create? 1
What width (in pixels) do you want it to be? 500

Making country 1's flag with width 500 pixels...
Done!

ls
a.out           flags.c   poland.ppm
```

Wait a minute, how did the program write to the terminal if it was redirected to the file?

Output redirection only redirects data written to standard output (stdout) because this is what typically outputs to the terminal. In this program, you'll have to use the fprintf() function to specify where you want to print things. In

[2] For example, the character "H" is 72 in ASCII. The binary representation would be 1001000.
[3] This is why the characters are unreadable when you open a ppm in a text editor, because the character value of 11111111 (or 255) doesn't correspond to a letter or number character that your editor can display.

our case, we can write to standard error (stderr), which will be displayed in the terminal because it won't be redirected to our image file.[4] To do this, use the code below as an example.

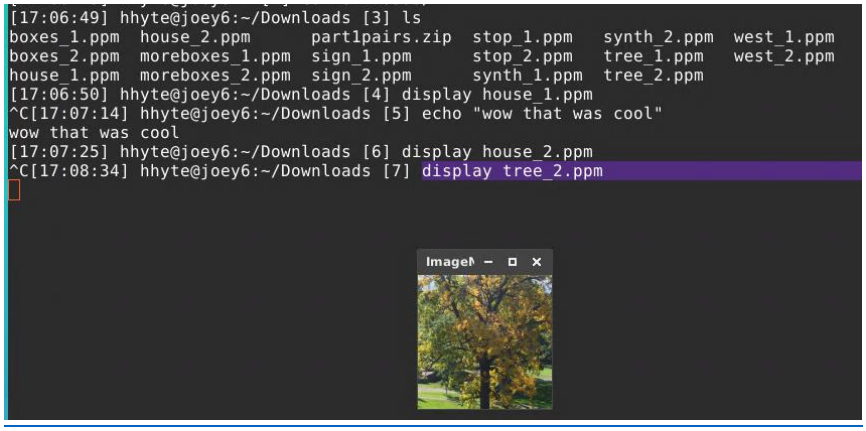
```
fprintf(stderr, "Some text to write to the terminal and not the file!\n");
```

Viewing Images

Now that we've created an image, how do we look at it to make sure it's correct? To open an image file on our computer, we can use the display function. This will work on all School of Computing machines.

```
display Poland.ppm
```

Now, your computer should look something like this:



If you're using ssh to connect to a different School of Computing machine, however, you'll need to make sure to use the -X tag in your ssh command. For example,

```
display Poland.ppm
 #it worked!

ssh joey3
display Poland.ppm
#that didn't work

ssh -X joey3
display Poland.ppm
#that did work! Yay!
```

If you're working from your laptop or at home, check out the Working Remotely section at the bottom of the lab.

[4] While this technically seems a bit hack-y, it'll be okay for our purposes.

Remember, you're always welcome to use the McAdams 110 main lab (which is avaliable nearly 24/7) if you need a computer!

Assignment

Now that we've defined everything, let's take a look at what you need to do for this lab.

Lab 07 Structure	
Tuesday	Thursday
30 minutes: work with 1-2 other individuals on Part 1 20 minutes: work individually on Part 2 Submit results of Part 2 to Canvas by 11:59PM.	50 minutes: work individually on Part 3 Submit to Canvas by 11:59PM on Friday.

Part 1: Planning with a Group (First 30 minutes)
***** Do not use a computer or calculator in this part *****

Since planning is incredibly important for being successful in this lab, we have doubled the number of points for the planning documents. We will also require that before speaking with a TA about a question, you must show them your planning document, so make sure you cover all the cases and logic that you can think of! Make sure that all of your group members have a good understanding of what's necessary with this lab before moving on.

Working with the people in your row, brainstorm what will be required for this program to accomplish it's goal. The specifications and starter code is listed in Part 3 of this document, so use that as a starting framework. If you have ANY questions about anything, ask your group members, or a TA.

Specific items that you should consider with your group during your planning session:

- What functionality do you still need to write?
- How do you write a double for-loop to iterate through a two-dimensional image?
- How can you determine which color to output using the modulus operator?

Part 2: Planning on Your Own (20 minutes of Tuesday Lab)
***** It's okay to use a computer and/or calculator for this portion *****

Note: This should not be when you're coding your solution. Make sure that you follow the steps below, using the guidelines and sample code above, to write and then submit your document before you begin.

Taking the information that was created during your brainstorm, write your own solution *in psuedocode*. How you choose to structure the planning document is of your choice, but you need to do the following:

- Review the explanation of *pseudocode* in the Safari playlist for this lab (https://learning.oreilly.com/playlists/1480ac0b-84d1-4ef6-97f4-f6936af15daf) or using the pseudocode.pdf document in Canvas.
- *Pseudocode* is NOT code. It is structured English that where you convey the key parts of your algorithm.
- Thoroughly consider all the issues you need to fully create a program as specified above (such as how to make the formatting look correct, how many variables do you need, what type, etc.).
- Include the following mandatory header:
    - Your First (or preferred name) and Last Name

- Your Lab and Lecture Section
- Lab 06 – Part 1
- Today's Date
- Collaboration Statement: In this statement you indicate who you worked with, and which lecture sections they are enrolled in.

Submit this part of the assignment to Canvas by 11:59 PM (as specified in the submission Guidelines). Late submissions will not be accepted. This part is to help you think before you code.

Part 3: Implementation (50 minutes on Thursday).

Lab Exercise

Write a program called flags.c that is capable of creating an image of the flags of Poland, Netherland, and Italy. country_code should be 1, 2, and 3 for Poland, the Netherlands, and Italy, respectively.

You will need to write the code to create the chosen flag. To do this you'll need to:

1. Use fprintf(stderr, ...) to prompt the user for input.
2. Use an if-else, or switch statement based on the country_code and put the code to create the flags within each part of the if-else or switch, whichever you have chosen.
3. Use nested for loops to iterate through the image, and generate an image of the correct size as determined by the user's input to width.
4. Determine what row and column the program is currently iterating through, and write the correct color based on where on the flag that is.

Whichever way you choose to complete this lab it is recommended to try and complete the Polish flag first, because it is the easier one of the three. Make sure the Polish flag works properly before continuing to Netherland's flag; make sure that one works, then do the last one.

You should always break up your programs into small parts, doing each one incrementally, compiling in between each step before continuing to the next step. Working in smaller sections to accomplish the larger goal will help you, especially if you utilize the thinking you developed in previous labs for splitting tasks apart.

Specifications
Your program should:

1. Match the example output above.
2. Use the proper colors and width-to-height ratios as defined on Wikipedia for the flag of each country. (Check out the Additional Resources section at the bottom of the lab).

Bonus Exercise

You may optionally complete one of the below exercises.

OPTION A
Create a copy of your flags.c file called flags_checkered.c that produces a checkered flag, with a random color for each square in the flag.

OPTION B

Create a copy of your flags.c file called flags_shapes.c that produces another flag, of your choice, that uses more interesting shapes than rectangles in the design.

Some possibilities include Bavaria (the variant with diamond shapes), the basic Brazilian flag (without the stars), Palau, Jamaica, or Japan.

Use the starter code below to help you get started, since this is a lot to keep track of. Just fill in the TODOs according to the instructions above. (**The starter code should be downloaded from Canvas.**)

Starter Code

```
/*
Remember to include the mandatory header!
*/

#include <stdio.h>

/*
Here are some helpful functions you can use!
Don't worry if you haven't seen these before! :-)
You'll learn more about how to make these yourself later, but for now
Just know that you can call these as you would any other function
And you write how they work down below.
*/
void make_pixel (int r, int g, int b);
void make_ppm_header (int width, int height);
void make_ppm_image (int country_code, int width);

int main()
{
        int width, country_code;

        fprintf (stderr, /* TODO: prompt for country code */);
        fscanf(stdin, /* TODO: read country code as an integer*/);
        fprintf(stderr, /* TODO: prompt for width of the flag in pixels*/);
        fscanf(stdin, /*TODO: read the width as an integer*/);

        fprintf(stderr, "\nMaking country %d's flag with width %d pixels... \n", country_code, width);

        // Write the image data
        make_ppm_image(country_code, width);

        fprintf(stderr, "Done!\n\n");
        return(0);
}
```

```
// Creates a pixel with the colors you tell it to use when you call it
// To call it, just do something like make_pixel(244, 244, 244);
void make_pixel (int r, int g, int b)
{
	fprintf(stdout, "%c%c%c", r, g, b);
}

// Creates a header with the desired width and height when you call it
void make_ppm_header (int width, int height)
{
	fprintf(stdout, "P6\n");
	fprintf(stdout, "%d %d %d\n", width, height, 255);
}

// Creates a complete ppm image when you call it
void make_ppm_image (int country_code, int width)
{
	// Here's an example of calling one of the functions we made earlier

	/*
	TODO
	Figure out the correct height per flag based on the country code
	and pass it into the make_ppm_header function
	*/
	make_ppm_header(width, height);

	/*
	TODO
	Write your logic to print out the pixel data here for each flag
	First, make an if-else or switch depending on the country
	Second, use nested for loops to iterate over the dimensions of the flag
	based on the flag you're creating and your position in the flag
	use make_pixel() to write the pixel the appropriate color

	You use make_pixel like you would any other function,
	just pass in the values you want for r, g, and b
	*/
}
```

Submission Guidelines

- Part 1: No submission, but don't skip it. Thinking aloud and collaborating at a high level is incredibly important in computer science. It'll help you ask better questions at office hours, write cleaner code, do great group work, and even do better in interviews.
- Part 2: Submit your writeup to the Canvas assignment associated for this lab by 11:59PM on Tuesday (10/1/2019). You should check within Canvas to make sure your submission was uploaded correctly and in its entirety.
- Submit your flags.c program to Canvas assignment associated for this lab by 11:59PM on Friday (10/4/2019. You should verify within Canvas to make sure your submission was uploaded correctly and in its entirety.
    - If you opt to complete flags_extenced.c, submit that to the associated Canvas assignment by 11:59PM on Friday (10/4/2019).

Grading Rubric

- If you are not present and attending lab during Part 1 and Part 2 of this lab on Tuesday, you will not receive credit for the planning portion of this lab.
- If your document for Part 2 is not submitted successfully to Canvas by the due date, you will not receive credit for the planning portion of this lab.
- If you do not check in your code with a TA for Part 3 of this lab on Thursday, you will not receive credit for the coding portion of this lab!
- If your code for Part 3 is not submitted successfully to Canvas by the due date, you will not receive credit for the coding portion of this lab.
- Your assignment will be graded out of 100 points. The approximate grading distribution is:
    - Brainstorming document completeness/accurate
        - Logic covers all requirements specified — 20 points
        - Document is simple and easy to follow — 20 points
    - Program flags.c
        - Program produces an open-able image output — 30 points
        - Program produces the desired country flag when input — 5 points
        - Flags are of the correct colors and width-to-height ratios — 5 points
    - Program compiles without errors or warnings — 10 points
    - Proper code formatting and commenting (See Code Style on Canvas) — 10 points
    - Optional bonus exercises
        - OPTION A
            - flags_checkered.c — up to + 5 bonus points
        - OPTION B
            - flags_shapes.c — up to + 10 bonus points
        - Requires flags.c is submitted, and only one may be submitted.

Working Remotely

Since this lab has a visual component, you may need to transfer your files to your local computer and then install another program to view ppm images. GIMP (the GNU Image Manipulation Program) is a great free program that can open PPMs, and works on Linux, Mac, and Windows. To transfer your files, you'll need to use FTP (using a GUI client, which are provided for free by CCIT, or by using the sftp command) or the scp command (which is talked about in Lab 2 if you need some help).

Additionally, you could choose to use a virtual machine through a remote desktop. Check out the link below.

Additional Resources

- Flag information:
  - Poland: https://en.wikipedia.org/wiki/Flag_of_Poland
    - RGB: white (255, 255, 255), red (255, 0, 0)
  - The Netherlands: https://en.wikipedia.org/wiki/Flag_of_the_Netherlands
  - Italy: https://en.wikipedia.org/wiki/Flag_of_Italy
- GIMP (The GNU Image Manipulation Program): https://www.gimp.org/
- CCIT Free Web Downloads: https://ccit.clemson.edu/support/faculty-staff/software/web-downloads/
- Virtual Desktops Help: https://www.cs.clemson.edu/help/remoteaccess.html#virt-desktop