Introduction

In this week's lab, we're revisiting Lab 7 with some different tools for constructing our program.

Lab Objectives

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

- Re-design existing code to produce PPM images using structs and arrays using modular functions.

Prior to Lab

- You should be familiar with input and output (including redirection), conditionals such as if, if/else, for loops (including nested for loops), and arrays.
- Review Lab 7 for instructions on how PPMs work (Headers and Pixel Data), redirecting output, working with nested for loops, displaying images from the command line, and working remotely with image files.
- This lab corresponds with zyBooks Chapters 6 and 7.

Deadline and Collaboration Policy

- This assignment is due by 11:00 PM on Friday (10/26/2018) via Canvas.
    - More instructions on what and how to submit are included at the end of this document.
- You should write your solutions to this lab by yourself. In this lab, **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.

Lab Instructions

In last week's lab, we worked with functions for the first time. We learned about the convention of a **driver**, that is, a main function that doesn't contain all the code for your program, but instead splits it up into several other **helper functions**. This week, we'll continue working with that design principle, but use it to help improve a program we've already written.

When you first wrote Lab 7, you didn't have experience using functions, and as a result, nearly everything that happened in your program occurred in the `make_ppm_image` function which you called from main. While this is better than cluttering our main with everything that needs to happen, it isn't perfect. Ideally, most programs strive for a **separation of concerns**.

The separation of concerns is the idea that functions of a program should only deal with what is necessary. If you want to write very complex programs, it becomes incredibly hard to figure out how to fix bugs if you don't have clear boundaries around what each function does. When complex actions are split apart over multiple simpler functions, it's easier to tell what's happening, because you can check if each individual part is working one by one! (Kind of like when you were writing your first programs, and used `printf()` statements every few lines to understand what was happening).

For this week's lab, we'll be having you split up the tasks that your program had to perform in Lab 7 into multiple smaller functions to aid the separation of concerns. We'll also be using structs to save our data, so we can see more clearly what we are working with when we refer to certain variables.

Structs are user-created variables that can be store whatever the user wants. A classic example of their usefulness would be a contact list. In a contact list, there is often a lot of data that needs to be preserved that would be a lot more difficult when using normal variables. Without structs, if we wanted to store data on a person, we'd have to make a bunch of variables.

```
char person1_first_name[] = "Examplefirstname";
char person1_middle_initial = 'E';
char person1_last_name[] = "Examplelastname";

char person2_first_name[] = "Not";
char person2_middle_initial = 'A';
char person2_last_name[] = "Realperson";
```

You can see how this would quickly get out of hand with more than two people. There's also no great way to iterate through this, if we wanted to keep track of everyone we know. Instead, let's create a person struct, and use that instead.

```
struct person
{
    char first_name[];
    char middle_initial;
    char last_name[];
};

struct person 1011_lab_friend;

strcpy(1011_lab_friend.first_name, "Firstname");
1011_lab_friend middle_initial = 'I';
strcpy(1011_lab_friend.last_name, "Lastname");

printf("My friend's middle initial: %c", 1011_lab_friend.middle_initial);
```

Now, we have a clear definition of our variables. For example, a middle initial is only important in relation to who it belongs to, and doesn't make much sense as a variable all its own floating around in our code. Using this, we could even create an array of people and iterate through that, if we wanted to store something like a class roster or a list of all the people we met at a tech conference, and print out all their names.

Now, let's get onto the assignment. You'll notice that this week doesn't have a planning document, since you've already thought through most of the requirements and logic for this program, and we're simply adapting them to a better design structure. Read the assignment below carefully to determine the new specifications.

Assignment

Lab Exercise

Write a program called `flags_revised.c` that extends the functionality of your solution for Lab 7 and matches the sample output below, as was shown in Lab 7. (You can use your code from that Lab as your starter code).

Sample Output

```
% prompt: ./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!

% prompt: ls
a.out          flags.c        poland.ppm
```

Lab 7 – PPM Flag Program Specifications (Your program should already do these things)

General

- Your program should be capable of creating a PPM image of the flags of Poland, Netherland, and Italy.
- Flags generated should 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).

Variables

`country_code` should be 1, 2, and 3 for Poland, the Netherlands, and Italy, respectively.

Functions

`main`

- Writes to standard error to prompt the user for necessary input.

`make_pixel`

- Writes passed pixel data to standard output as unsigned characters, without spaces or newlines.

`make_ppm_header`

- Writes a PPM header to standard output.

`make_ppm_image`

- Uses nested `for` loops to iterate through and generate a PPM image.
- It takes the parameters `width` and `country_code`.
- It should call the new functions as appropriate to create the image.

Lab 9 - New Specifications

General

- Your program should also be capable of creating a PPM image of the flag of Benin.
- The new flag generated should use the proper width-to-height ratio as defined on Wikipedia for the flag.
    - For the colors, use pure red, and yellow, and a value of 170 for green.
        - Pure color values are talked about in Lab 7.

- The left green field should be 6/15ths of the width of the flag.

Variables

`country_code` should also include a new option, 4, for the flag of Benin.
`struct pixel` contains 3 integer values `r`, `g`, and `b`
`struct image` contains 3 integer values `width`, `height`, and `country_code`

Functions

`main`

- Functions as a driver –
  - calls other functions as appropriate, does not print out the flag or do any calculations.
- Should ensure that the user enters a valid number for the `country_code` and `width`.

`make_pixel`

- Instead of receiving 3 integers as its passed parameters, this function should instead take one `pixel` struct.

`make_ppm_header`

- Instead of receiving 2 integers as its passed parameters, this function should instead take one `image` struct.

`make_ppm_image`

- Should not perform height calculations or include logic to determine the color of the pixel to print.
- This function should only contain **ONE** set of two nested for loops.
- This function should call the appropriate new and old functions to create the PPM image.

`get_color`

- When called, it should return a `pixel` struct of the appropriate color.
- Feel free to create and call smaller functions for each country (such as `get_color_italy`) from this function, but that is not required.
- It takes the parameters `column`, `row`, and an `image` struct.

`calculate_height`

- When called, it should return a completed `image` struct, with the appropriate number of pixels in height a flag should be when made with the passed width set.
- It takes the parameters `width` and `country_code`.

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.

For this assignment, it is probably be a good idea to first make your program from Lab 7 work for the flag of Poland with a pixel struct, and then add the new functions one by one. Once you have that completely working, then try to adapt the other flags.

Code Formatting Guidelines

While you are learning the basics of programming, it's important to keep in mind that programming style matters for code readability. Here are some basic practices that we will look for when grading your labs from here on out:

1. **Consistent Indentation**. By now, you have probably seen first-hand that consistent indentation helps you spot errors quicker. There are several ways to ident code---just be sure to do so consistently.

2. **Commenting & Documentation:** A good rule of thumb for commenting is: "Would this help me remember why I'm doing what I'm doing if I look at this code 2 years down the road?" Comments should help make clear what your thought process is, what assumptions are being made, etc. However, you do not want to be redundant by over-commenting.

3. **Code Grouping:** If a task within your code requires a few lines of code, it's generally a good idea to keep these tasks within separate blocks of code. Declare variables close to where you use them so you don't lose track of anything.

4. **Consistent Naming Conventions:** Word boundaries are typically via either camelCase or by using underscores. Temporary variables, such as counters, use the same common naming scheme. Variables should typically have descriptive names to help you remember what is happening.

5. **Avoid long horizontal lines of code:** Humans generally prefer reading tall and narrow columns of text. Writing shorter lines of code makes it easier to spot errors (because some of the code may be off the screen to the right!). A line of code is generally preferred to be 80 characters or fewer.

If you'd like to see an example of good versus bad code, check out the Code Style section at the bottom of Lab 5.

Submission Guidelines

- Submit your `flags_revised.c` program to Canvas assignment associated for this lab by 11:00PM on Friday (10/26/2018). You should verify within Canvas to make sure your submission was uploaded correctly and in its entirety. Your code should be follow good coding formatting practices and proper documentation. You must include the academic honesty header and *explicitly* list the names of TAs that you asked for help outside of lab.

Grading Rubric

- If you are not present and attending lab on Tuesday and Thursday, you will not receive credit for this lab.
- If you do not check in your code with a TA on Thursday, you will not receive credit for this lab.
- If your code is not submitted fully to Canvas by the due date, you will not receive credit for this lab.
- Your assignment will be graded out of 100 points. The approximate grading distribution is:
    - Program `flags_revised.c`
        - Program prevents invalid numbers from being entered — 5 points
        - Program produces the desired country flag when input — 5 points
        - PPMs
            - Program produces an open-able image output — 10 points
        - Flags
            - New flag is produced correctly when input — 5 points
            - Flags are of the correct colors and width-to-height ratios — 5 points
        - Functions
            - Program is divided into new functions as specified — 20 points
        - Structs

- Program used a pixel and image structure as specified 20 points
- Program compiles without errors or warnings 10 points
- Proper code formatting and commenting (See section above) 20 points

Additional Resources

- Information about working with PPMs in lab and remotely is available in the Lab 7 document.