Lab 2: PPM Reading/Writing + Structs

CPSC 1021 – Spring 2018

Due Date: Friday, January 26, 2018 @ Midnight

Lab Objective

- Reading and Writing PPM Images
- C Struct Review
- Practice using malloc and free
- Header Files
- Practice Writing Reusable Code

Introduction

In this lab, we will begin working with PPM images which will become a theme this semester. Aside from the interesting image applications we can write, working with images is an interesting case study in complex file I/O and dynamic memory structures. Make sure to review the PPM format, both P3 and P6, as well as basic C I/O.

Resources

Netpbm format specification: https://en.wikipedia.org/wiki/Netpbm_format

C File I/O: https://www.cprogramming.com/tutorial/cfileio.html

C Command Line Arguments: https://www.cprogramming.com/tutorial/c/lesson14.html

C Structs: https://www.cprogramming.com/tutorial/c/lesson7.html

Assignment

You should first download the starter code and PPM image from Canvas. This will include **puppy.ppm** and **ppm_utils.h**. You should then complete the following objectives:

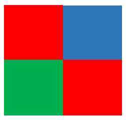
- Create 2 files: **lab2.c**, **ppm_utils.c**
- **lab2.c** will contain a main method which should:
    - Require 2 **command line arguments** to be passed at execution: an input file name and an output file name. If these arguments are not provided, the program should exit with a status of **1** and print some error message.
    - Open the input file for reading and the output file for writing. If either fails, return **1** and print some error message.
    - Using the functions in ppm_utils, read in the image file creating an **image_t*** struct
    - Using the functions in ppm_utils, write the same image out as a **P6** ppm image
    - free() the **pixel_t*** array and the **image_t*** struct
    - close() the open files
    - return **0**
- **ppm_utils.c** should implement all the function defined in **ppm_utils.h** including:
    - header_t read_header(FILE* image_file);

- void write_header(FILE* out_file, header_t header);
- image_t* read_ppm(FILE* image_file);
- image_t* read_p6(FILE* image_file, header_t header);
- image_t* read_p3(FILE* image_file, header_t header);
- void write_p6(FILE* out_file, image_t* image);
- void write_p3(FILE* out_file, image_t* image);

The details of each function are described after a brief image/pixel review:

Images, PPM, Pixels

Remember from your discussion in class that a PPM image stores data in 2 pieces: a header line and R/G/B values for each pixel. A pixel is one point of color in an image, and there are WIDTH * HEIGHT pixels in any given image. The R/G/B values will range from 0 to 255, and can be stored as integers (P3) or as 1-byte characters (P6). For example, here is a 2 x 2 PPM image as well as what it would look like on disc:



```
P3 2 2 255

244 0 0 0 0 244

0 244 0 244 0 0
```

```
P6 2 2 255

⌠ \0 \0 \0 \0 ⌠

\0 ⌠ \0 ⌠ \0 \0
```

Notice that the P6 image will most likely display in a text editor as the ASCII encoded character of the color value. P6 is a **smaller** file since it uses at most 1 byte per color value, while P3 is a **larger** file since it will use at most 3 bytes for a single color value. P3 is useful for debugging, which is why we are supporting it.

When reading or writing a P3 you should use the **%d** code for fscanf()/fprintf() to make sure you get data in the correct format. For P6, you should use **%c** so that you only get 1 byte of data.

IMPORTANT!
We will be storing the pixel data in a **1-D array!** Just like they are stored in the file, we want one long chunk of pixel data. This simplifies our implementation, and if we want to recover row/column info we can simply do some math using the width/height stored in the header.

Function Details

<u>header_t read_header(FILE* image_file);</u>

Input is a PPM image file of any type. The function should read the PPM header as defined in the specification and store the values in a header_t struct which is returned. See the struct definition in the header file. This should use a fscanf() function call, and should only be **3** lines of code!

<u>void write_header(FILE* out_file, header_t header);</u>

Input is an output file and an existing header_t struct. Should write the header to the file using fprintf(), and should only be **1** line of code! Remember that there must be whitespace (or a newline) after the header in a PPM image!

image_t* read_ppm(FILE* image_file);

Input is an open PPM image file. This function should **read the header using the above help functions**. After reading the header, determine which of the below read functions should be called, i.e. if you read the header and find a P6 header, call the **read_p6()** function and return the result.

image_t* read_p6(FILE* image_file, header_t header);

Input is an open image file and a populated header. Should use the header to determine how many pixels exist in the image (width * height). Using this pixel number, **malloc()** an image_t* struct and **malloc()** a pixel_t* array large enough to hold all the image data. **Note that you need 2 calls to malloc()!** Now use fscanf() to read in the red, green, and blue components of each pixel and populate the pixel array. For this function your format string in fscanf() will look like **"%c%c%c"** since P6 is stored as raw bytes. Note that the image_t struct stores all color values as integers, so you will need to cast the read in values before assigning them to a pixel_t.

image_t* read_p3(FILE* image_file, header_t header);

Same as the above function (so copy and paste your code…) but the format string should be **"%d %d %d"** Notice the spaces between %d's!

void write_p6(FILE* out_file, image_t* image);

Input is an output file to write the image to and the image to write. This function should call **write_header()** first. Note that you should **alter the image header** to include the correct magic number. There is no guarantee that the image passed in is a P6, so we will need to set the magic number correctly and cast the pixel data down to **type char** when we use **fprintf()**. This allows us to convert P3 to P6 and P6 to P3 if we want.

Note that the code is going to look remarkably similar to your read image code!

void write_p3(FILE* out_file, image_t* image);

Same as the P6 version, but the header should be set to P3 and you should write integers to the file. Remember to reuse your code as much as possible!

Tips and Tricks

None of the functions in ppm_utils should be more than **12-13** lines depending on your formatting. Some will be much less. Don't overthink it, and ask questions often so you don't get confused! **This code will be used throughout the semester, and will be needed for projects as well!** We're doing it in lab to save you time down the road.

Remember that when using a **pointer to a struct** such as an image_t*, you should use the arrow notation to access members. For example:

```
image_t* image = read_image(…);
pixel_t* pixels = image->pixels;
header_t header = image->header;
```

When you just have an **instance** of a struct, you should use dot notation:

```
pixels[0].R = 215;      // Set the 0th pixel's red value to 215
int num = header.WIDTH * header.HEIGHT;    // Calc number of pixels
```

Your main method should be short as well… in all you should write a little over **100** lines of code. Much of this can be copy/pasted with slight alterations to boot. The goal is to produce code that we can reuse throughout the semester with few alterations.

Make sure to **check out the linked resources for a C refresher**.

Deliverables

When I run your program as below:

```
gcc lab2.c ppm_utils.c -o ppm_conv
./ppm_conv puppy.ppm p6.ppm
```

The result should be an exact copy of **puppy.ppm** (which is stored as a P3) but now it is in the P6 format. To verify that your code works, try the following command:

```
display puppy.ppm
display p6.ppm
```

Both should look the same!

Submit all **.c source files as well as the .h header file** in your tar archive.

Compile and Execute

Use GCC to compile your code as follows:

```
gcc lab2.c ppm_utils.c -Wall -o ppm_conv
```

Execute the program

```
./ppm_conv puppy.ppm p6.ppm
```

Formatting

Your program should be well documented!

1. Each file should include a header →
2. Your program should consist of proper and consistent indention
3. No lines of code should be more than 80 characters

```
/*
  Nick Glyder
  CPSC 1021-002      ← Your Lab Section here
  Lab 2
*/
```

5 – 10 points will be deducted for each of the above formatting infractions.

Submission Instructions

- Test your program on the School of Computing server prior to submitting.
- Use the tar utility to tar.gz all source files. **Do not tar an entire directory!**
  When I decompress your archive, I should see all of the files you included, not a top level directory!
  Failure to correctly tar may result in up to a 25 point penalty!

```
tar –czvf nglyder-lab2.tar.gz *.c *.h
```

- Name your tarred file **<username>-lab<#>.tar.gz** (ex. nglyder-lab2.tar.gz)
- Use handin (http://handin.cs.clemson.edu) to submit your archive