CpSc 1111 Lab 11
Parsing Header Info from a ppm Image

Overview

This week, you will write a program that consists of multiple source files to parse the header information from a ppm image. Some ppm images and some starter code will be provided. You will need to write the function to parse the header, as well as a makefile, and a header file. The files from this lab will be starter files for the third programming assignment.

Background Information

For this lab, you will write a program that parses the header of a ppm image and prints out the dimensions of the image to the user. This will be used as starter code for your parsing function for PA3 in lecture.

- -

ppm Images
As you have learned in lecture, ppm images are a type of image format. It is easy to write and analyze programs to process this type of image.

There are two parts to ppm images: the header and the pixel data. The header consists of a special tag indicating the type of image – we will be using images with the P6 tag. The dimensions follow that, with columns stated first then rows; and then the largest value for any pixel, which is usually 255. If you open an image file using an editor like vim, you can see the header information at the top of the file.

The image pixel data immediately follows the newline character after the 255, and is binary data, so it is not anything that will make any sense when viewing the file with an editor like vim.

- -

Command-Line Arguments
You also saw in lecture that when programs that use command-line arguments, the main() function signature changes from

```
int main(void) {
```

to

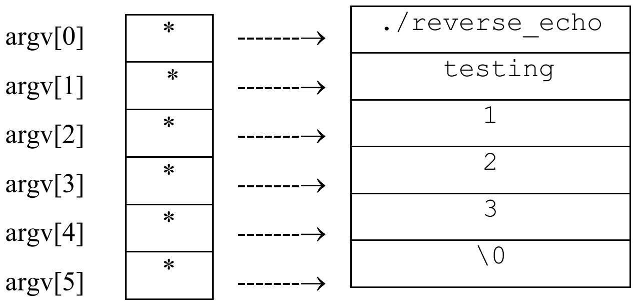
```
int main(int argc, char *argv[]) {
```

where `argc` holds the number of items entered at the command-line, and `argv` is the array that holds each item (or actually, a pointer to each item) that was entered at the command-line.

For example, with the reverse_echo program from the previous lab, you saw that to run that program, you would type something like the following:

```
./reverse_echo testing 1 2 3
```

In that case, the value of `argc` would be 5, and the `argv` array would look like the following:



For this lab, and for the third assignment, your program will use command-line arguments. To run the program, you will type the executable followed by the image file name, such as:

```
./a.out tiger.ppm
```

- -

Pointers
In yesterday's lecture, you were introduced to pointers. You will use pointers in this lab for the parameters of the `parseHeader()` function that you will be writing.

As you know, functions can only return one value, but we need to get two values from the header information of the image file: the number of columns and the number of rows.

You also know by now that variables that are passed by value to functions are copied into the function parameters. When the function ends, those local copies go away and any changes that were made to those local copies in the function are not reflected back in the arguments that were sent in to that function.

So to be able to get more than one value changed or assigned or modified from a function, addresses to the variables are sent in to the function. This is called pass by reference. An example of a function using pass by reference can be found in the swapPBR.c source file found on Canvas in the module for Chapter 10, Pointers).

When the function assigns the values to those address locations sent in to the function parameters, the variables sent in will contain those values too because those function parameters point to those very same memory locations.

- -

File Pointers
Also covered in lecture earlier were file pointers. The general form for declaring a file pointer is the following:

```
FILE *<variable_name>
```

So the following line declares a variable called `inFile`, which is a file pointer type.

```
FILE *inFile;
```

Then, you can assign to the file pointer variable the result of opening a file (assign it to point to a file). If you are trying to open a file to read from called `tiger.ppm`, the code would look like:

```
inFile = fopen("tiger.ppm", "r");
```

If the file doesn't exist or that `fopen()` function fails for some reason, the value of `inFile` would be NULL. You can add some error checking code after trying to open a file, like the following:

```
if (inputFile == NULL) {
   fprintf(stderr, "File open error. Exiting program.\n");
   exit(1);  // need to #include <stdlib.h>
}
```

But, when providing the name of the file as the second argument on the command-line, you can open the file contained in `argv[1]` rather than hardcoding the file name in your code; that way you can run your program using different files without changing the hardcoded name in your code.

- -

File I/O Functions

C provides quite a few functions that can be used to read characters or strings or entire lines from a file. See the notes posted on Canvas in the module for Chapter 15, File I/O. Or you can use the appendix at the back of your book, or use an online reference such as cplusplus.com: http://www.cplusplus.com/reference/cstdio/

Lab Assignment

1. While logged in to one of the School of Computing servers (adas, babbages, or joeys e.g. ada1.computing.clemson.edu) type the following copy command:

   ```
   cp /group/course/cpsc111/public_html/F18Labs/lab11/* .
   ```

   Don't forget the `.` (dot) at the end. That copy command above says the following: copy all the files (the wildcard `*` means all) at that location `/group/course/cpsc111/public_html/F18Labs/lab11/`) to your current location (indicated by the `.` (dot)).

   The following files will be copied to your directory:
   - `mainDriver.c` with starter code; need to fill in some code
   - `parse.c` with header comment and #include for defs.h – this is where the `parseHeader()` function goes
   - `defs.h` with #includes and a spot for the prototype of the `parseHeader()` function
   - two image files

   You will also need to write a makefile that at the very least contains a target for compilation.

2. Your `parseHeader()` function will need the file pointer sent to it from `main()` as well as pointers to the column value and the row value.

   Since the 3 parts of the header can be on separate lines or not, and may or may not have comments in between them, parsing the header can be tricky. Take a look at each of the image files to see what the headers look. If your program for this lab parses the columns and rows from those two files, then it will be sufficient for the lab. For PA3 in the lecture, however, you should make sure to add code to handle other variations of the header.

   You will use file I/O functions to read in character by character, or string by string, or an entire line at a time. There are many different ways that this can be done. You will have to take a look at the different functions available and think about what makes the most sense to you.

   Remember, that for this lab, the goal is just to parse from the header the columns and the rows and print those values to the user. But for PA3, you will have to add more code to continue parsing through the header so that you move the file pointer to the next item after the 255; then at that point, the image pixel data can be read in.

   2 SAMPLE RUNS OF THE PROGRAM:

   ```
   [20:14:07] chochri@joey4: [51] ./a.out tiger.ppm

   width is 690, height is 461



   [20:14:13] chochri@joey4: [52] ./a.out julia.ppm

   width is 500, height is 500
   ```

Reminder About Formatting and Comments

- The top of **all of your source files** should have a header comment, which should contain:
  - Your name
  - Course and semester
  - Lab number
  - Brief description about what the program (or function) does
  - Any other helpful information that you think would be good to have.
- Variables should be declared at the top of the main function, and should have meaningful names.
- Always indent your code in a readable way. Some formatting examples may be found here: https://people.cs.clemson.edu/~chochri/Assignments/Formatting_Examples.pdf

Turn In Work

1. Before turning in your assignment, make sure you have followed all of the instructions stated in this assignment and any additional instructions given by your lab instructor(s). Always test, test, and retest that your program compiles and runs successfully on our Unix machines before submitting it.
2. Show your TA that you completed the assignment. Then submit all 4 of your files: `mainDriver.c`, `parse.c`, `makefile`, and `defs.h` using the handin page: http://handin.cs.clemson.edu. ***Don't forget to always check on the handin page that your submission worked. You can go to your bucket to see what is there.***

Grading Rubric

For this lab, points will be based on the following:

Functionality	70	(50 for parse.c, 20 for mainDriver.c)
defs.h	10	
Makefile	10	
Code formatting	10	

NOTE: there could be other possible point deductions for things not listed, such as:

- global variables
- use of break not in switch statements
- naming the file incorrectly
- warnings with compilation
- missing header comments at the top of each source file
- missing other comments in code
- missing return statement at bottom of main() function
- etc.