CS 100 Project Five – Spring 2016

Overview: This program copies, shrinks or expands image files.

Pictures can be stored in many formats. You've probably heard of the more common ones, such as **jpeg** (from the Joint Photographic Experts Group) and **gif** (Graphics Interchange Format). The vast majority of these formats store the image data in a binary file. There is one format, ppm, that uses **ASCII** files to store an image. These **ppm** files consist of header information and then a long string of numbers representing the red, green and blue components of each pixel in the image. It is not widely used – since the images are stored in **ASCII**, they are much larger than other (binary) formats. However, as these files are (readable) **ASCII** data, they are a good format for an introductory programming class.

The ppm format that is used for this project is shown at the right. The very first line is always **P3**. After that, you have three values (width, height, colors) that can be on a single line or separate lines.

```
P3
width-in-pixels    height-in-pixels
maximum-color-value
pixel-1-1-red pixel-1-1-green pixel-1-1-blue      pixel-1-2-red pixel-1-2-green pixel-1-2-blue ...
pixel-2-1-red pixel-2-1-green pixel-2-1-blue      pixel-2-2-red pixel-2-2-green pixel-2-2-blue ...
...
pixel N-1-red pixel-N-1-green pixel-N-1-blue     pixel-N-2-red pixel-N-2-green pixel-N-2-blue ....
```

Finally, you have the actual RGB values (three integers) for each pixel in the image. A very tiny **ppm** file that is four pixels wide and six pixels tall, with the top two rows the color red, the middle two rows the color green, and the bottom two rows the color blue, is shown at the right below.

```
P3
4   6  255
255 0 0   255 0 0   255 0 0   255 0 0
255 0 0   255 0 0   255 0 0   255 0 0
0 255 0   0 255 0   0 255 0   0 255 0
0 255 0   0 255 0   0 255 0   0 255 0
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
```

This program will be manipulating **ppm** image files. In order to see if your program is working properly, you need to be able to view these images. Your system might (or might not) have a viewer that supports **ppm** images. If not, then download the free program **GIMP** (GNU Image Manipulation Program). Download **GIMP** at **http://www.gimp.org/downloads**

You can use **GIMP** to convert additional pictures into **ppm** format for testing if you wish. Simply load an existing image that you have into GIMP and then select the "**Export As**" option and a file type of PPM image. Make sure to click **ASCII** when asked how to export. When you use **ppm** to convert an image into a **ppm** file, it puts a comment on the second line. You need to delete that comment line, as our program does not handle comments in **ppm** files.

You can retrieve (using **wget** or **curl**) several sample **ppm** files from **troll.cs.ua.edu/cs100/projects/project5**. The files are alabama.ppm, beach.ppm, mountain.ppm, bryant-denny.ppm, monalisa.ppm**,** milky-way.ppm, river.ppm and earth.ppm. These pictures are listed from smallest to largest. The first three are pretty small. The last three images are large images. It will probably take your program a few seconds to process them.

All input to this program is given via the command line. The user should specify whether to copy, grow or shrink the file, the input file name, and the output file to be created. The command-line arguments will always be given in the order shown: either ***copy*** or ***grow*** or ***shrink***, then the input-file-name, and then the (new) output-file-name.

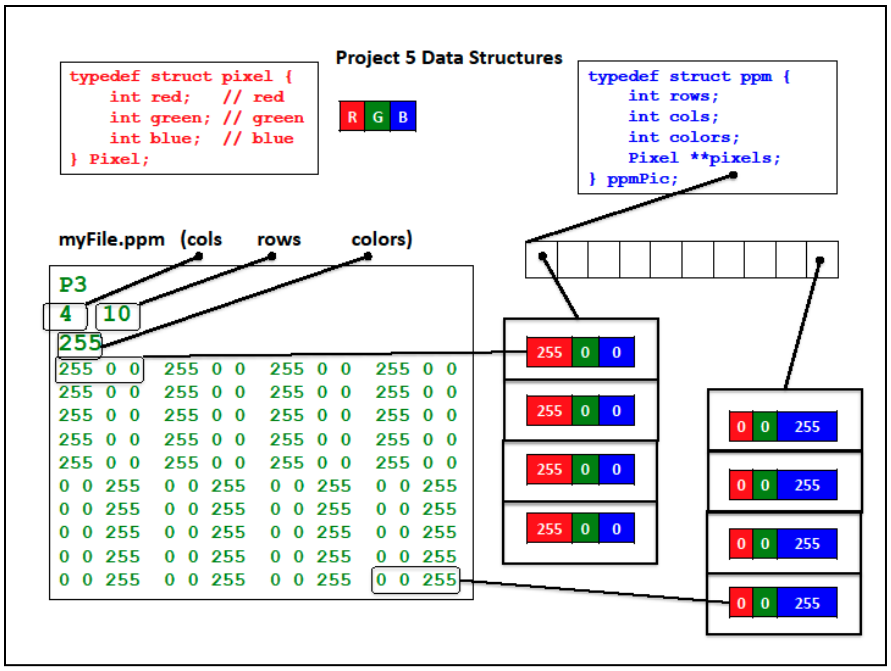
```
./a.out  copy   beach.ppm  newBeach.ppm
./a.out  grow   alabama.ppm  UA.ppm
./a.out  shrink  mountain.ppm  m2.ppm
```

You **must** use the two struct definitions shown at the top of the next page to store the ppm image that you read from the input file. These should be declared in the file `ppm.h`. Your main program will need this include file as it will need to know about these two new types.

```
typedef struct ppm {                        typedef struct pixel {
    int rows;       // number of rows              int red;    // red value
    int cols;       // number of columns           int green;  // green value
    int colors;     // number of colors            int blue;   // blue value
    Pixel **pixels; // actual pixel data        } Pixel;
} ppmPic;
```

In the **ppmPic** structure, the first three fields should be fairly obvious. The first and second, ***rows*** and ***cols***, are the number of rows and the number of columns in the file. The third, ***colors***, is the maximum color depth. You get these values when you actually read the file data. After this, you have **`Pixel **pixels`**. This represents the two-dimensional array that holds all the R–G–B values for each pixel in the image.

In order to help you understand how these data structures work, the diagram at the right helps illustrate the use of these data structures. The text below discusses how to create and read a **ppm** file.



```
typedef struct pixel {
    int red;   // red
    int green; // green
    int blue;  // blue
} Pixel;
```

```
typedef struct ppm {
    int rows;
    int cols;
    int colors;
    Pixel **pixels;
} ppmPic;
```

```
P3
4 10
255
255 0 0   255 0 0   255 0 0   255 0 0
255 0 0   255 0 0   255 0 0   255 0 0
255 0 0   255 0 0   255 0 0   255 0 0
255 0 0   255 0 0   255 0 0   255 0 0
255 0 0   255 0 0   255 0 0   255 0 0
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
0 0 255   0 0 255   0 0 255   0 0 255
```

To allocate space for this image, you need to allocate a new **ppmPic** object.

ppmPic *myPic = malloc (sizeof (ppmPic));

Once you have that object, you then read the first few items (columns, rows, number of colors) for the object.

Once you know the number of rows and columns, you must then allocate space for the actual two-dimensional array of pixels. In the case above, you have ten rows of Pixels and four Pixel values on each row. First, you allocate ten pointers that point to the first Pixel on each row.

```
myPic->pixels = malloc(sizeof(Pixel *) * 10);
```

You then allocate space for the four Pixels that exist on each row.

```
myPic->pixels[i] = malloc(sizeof(Pixel) * 4);
```

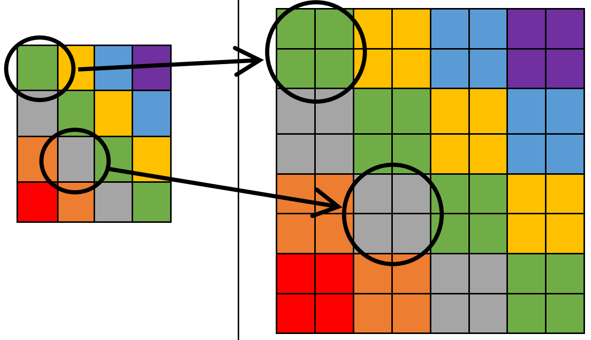
As an example of referencing individual elements using array notation for the case above, consider the code below

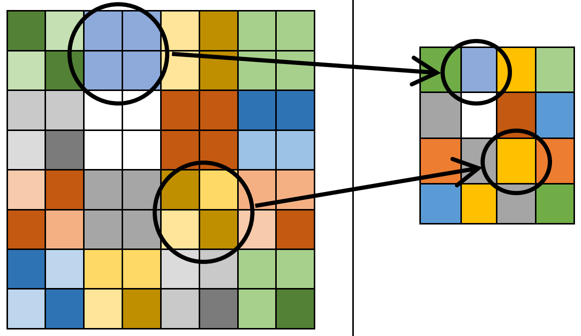
```
// Assign 255 to red field for last Pixel on second row (row=1 and col=3)
myPic->pixels[1][3].red = 255;
// Assign green value for second Pixel on first row to val (row=0 and col=1)
val = myPic->pixels[0][1].green;
// Assign 0 to blue field for first Pixel on last row (row=9 and col=0)
myPic->Pixels[9][0].blue = 0;
```

Growing and Shrinking an Image

Once you have read an image, the process of growing (or shrinking) the image must be identified. We will use a simple process that is not very sophisticated. There are better algorithms out there, but they are more complicated than the simple algorithm that we are going to implement for this project.

Our algorithm for growing an image doubles the width and height of the image, thus increasing the actual size by a factor of four. Likewise, our algorithm for shrinking an image cuts the width and height of an image in half, thus decreasing the actual size by a factor of four. When shrinking an image, you generate a new Pixel that is the **average** of the red and green and blue values for each of the four Pixels in the block you are shrinking.

Growing an Image	
Original ppm	New ppm
	

Shrinking an image	
Original ppm	New ppm
	

If your original image has an odd number of rows (or columns) and you are shrinking the image, just ignore the final (largest) row (or column). For example, if your original image had 9 rows, your new image would only have 4 rows and would be constructed from data found in the first eight rows of the original image (the 9th row is ignored).

<u>Checklist for Completing this Project</u>

1. Create a directory called **project5** on your machine. In that directory, you will have files named **ppm.c** and **ppm.h** and **main.c** and **Makefile**, as well as **scanner.c** and **scanner.h** and a sample ppm data files.
2. The main routine (main.c) controls execution of the program by reading the command line arguments and calling the ***copy*** or ***shrink*** or ***grow*** function. It is relatively short, roughly 20-25 lines. The functions that actually carry out the work (***read***, ***write***, ***copy***, ***grow***, ***shrink***) are found in ppm.c.
3. You must have a Makefile that will compile your program.
4. At a minimum, your **ppm.c** module should have five functions, one to read an image, one to write an image, one to copy an image, one to take an existing image and shrink it, and one to take an existing image and grow it. You can add other helper functions to this module if you wish.
5. Observation #1 – When you use real images, a **ppm** file gets big quickly. For example, the **mountain.ppm** file contains over 50 million characters. This makes debugging on real files problematic. However, if you use really small files then it is hard to see them using **GIMP** or other image display software, they are just too small. One suggested option for debugging is to create a very small image, such as the ones shown previously. You can examine the output of these files by hand. Once you think your algorithm is working properly, try it on large files.
6. Observation #2 – Performing a ***grow*** and then a ***shrink*** should result in your original file. However, performing a ***shrink*** and then a ***grow*** gives close approximation to the original, but not an exact copy.
7. Before tackling ***grow*** and ***shrink***, we recommend you get ***copy*** working. The ***copy*** function requires you to be able to read and write **ppm** images. Once you get your input and output (and ***copy***) functions working, then focus on ***grow*** and ***shrink***.

<u>When you are ready to submit your project:</u>

- Make sure your program runs properly on **cs-intro.ua.edu**. Your program is graded on that system.
- Bundle your **project5** directory (be sure to include **main.c** and **ppm.h** and **ppm.c** and a **Makefile**) into a (compressed) zip file.
- Once you have a compressed zip file that contains your **project5**, submit that file to Blackboard.

Project Five is due at 5:00pm on Monday, April 25. Late projects are not accepted.