ECE 209 Program 3: `image`

Fall 2009

***Due Tuesday, October 27, 2009 @ 11:45pm***

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of -100 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

**DO NOT copy code from the Internet**, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program will do some simple manipulations on grayscale images, stored in the PGMA (Portable Grayscale Map (ASCII)) format. Your program will be able to read an image file, rotate the image, flip it, invert the grayscale, and write it out in PGMA format. The transformations might also apply to only a portion of the image. This will involve the use of the file I/O functions that we've used already, together with 1D and 2D arrays.

## Images, Pixels, etc.

Digital images are made up of pixels[1], each of which represents a fixed-size region of the image. Data are stored for each pixel that represent its color, brightness, and other properties needed to draw the pixel on the screen or print it on the page.

The simplest representation is a "bitmap," in which each pixel is represented by a single bit – the pixel is either black (1) or white (0). A slightly more sophisticated representation is to associate a "grayscale" value – between 0 and 255, for instance – that represents a shade of gray between totally black (0) and totally white (255). Color images generally provide values for the amount of red, green, and blue (RGB) in each pixel. Increasing the number of bits associated with each pixel allows for more choices of color.

The image is represented as a two-dimensional collection of pixels. For example, an 80×80 image consists of 80 rows, with 80 pixels in each row. The size of an image is normally expressed as *width* × *height*, so a 40×60 image has 60 rows (height) with 40 pixels in each row (width).

For this assignment, we'll be dealing with grayscale images. There is one value associated with each pixel; the value ranges from 0 to 255, representing totally black to totally white, respectively. An example 4×4 image's data might look like this as a two-dimensional matrix:

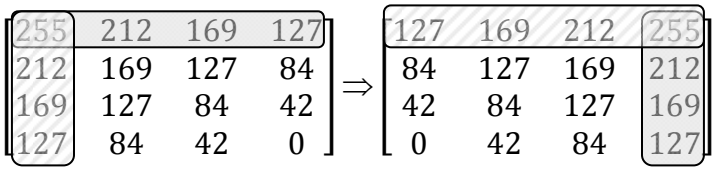
$$\begin{bmatrix} 255 & 212 & 169 & 127 \\ 212 & 169 & 127 & 84 \\ 169 & 127 & 84 & 42 \\ 127 & 84 & 42 & 0 \end{bmatrix}$$

[1] The term "pixel" is derived from "picture element."

The common numbering scheme for matrix elements is (x,y), where x is the row number and y is the column number. The top left corner is (0,0). Moving to the right, the x-coordinate increases, yielding (1,0). Moving down from the top left corner increases the y-coordinate to (0,1). Thus the image represented by this matrix transitions from white in the top left corner – element (0,0) – to black in the bottom right corner – element (3,3).

## Image Transformations

We can make changes in the image just by rearranging the order of the pixels in the matrix. For example, consider *rotating the image clockwise by 90 degrees*. In that case, the top left corner pixel (0,0) moves to the top right (3,0). The next pixel on the same row (1,0) moves to the corresponding spot on the rightmost column (3,1). So the top row of values is moved to the right column, as illustrated in the figure below. Similarly, row 1 moves to column 2, row 2 moves to column 2, and row 3 moves to column 0.



We can generalize this transformation: element (x,y) becomes element ((n-1)-y, x) where n is the number of rows/columns (assuming a square matrix).

Similarly, we can create transformations to rotate counter-clockwise, flip horizontally, etc.

We can also make color transformations by adjusting the values. To create a "negative" image, where we invert the blackness/whiteness of each pixel, we subtract the current value from 255. A completely black pixel (0) becomes white (255-0 = 255), and a completely white pixel (255) becomes black (255-255 = 0). A dark gray pixel (200) becomes light gray (255-200 = 55).

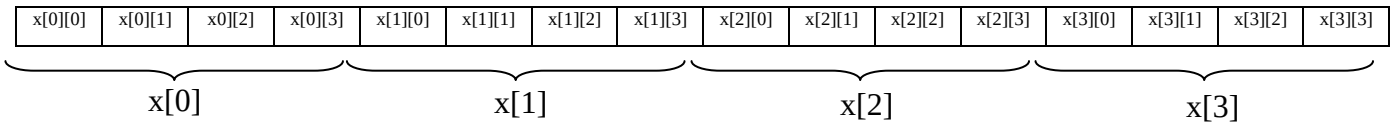
## Two-Dimensional Arrays

The natural way to represent two-dimensional data, like the image data above, is with a *two-dimensional array*. A four-by-four array of integers is declared as:

```
int image[4][4];
```

This is literally an array of arrays: image[0] is an array of four integers: image[0][0], image[0][1], image[0][2], and image[0][3].

Arrays are always laid out in contiguous memory locations, and a 2D array is no exception. Since each element is an array, the elements of the 2D array are arranged in memory as follows:




To calculate the address offset[2] of element [i][j] from the beginning of the array, we first skip to the i-th element. Since the size of each element is four (because it's a four-element array), this becomes i×4. Then we skip to the j-th element of that sub-array (and the size of each element is 1). So the total offset from the beginning of x is **i×4 + j**. The offset for element [i][j] in a $dim_1 \times dim_0$ array is $i \times dim_0 + j$. (For different element types, we have to multiply the offset by the size of the elements, of course.)

[2] We're assuming LC-3 addressing and data sizes for this example.

Relating this structure to the image data above, we store the value for pixel (0,0) in element [0][0] of the array. In general, pixel (i,j) is stored in image[i][j]. Note that elements are stored in "column-wise" fashion – that is, the first set of elements corresponds to the leftmost column of the image, where the x-coordinate is 0.

Just like 1D arrays, 2D arrays are passed to functions by reference. Because the size of the rightmost dimension is needed to calculate the offset properly, this dimension must be specified explicitly in the function declaration and definition.

For example, suppose we're working only with 4×4 grayscale images, as described above. We represent the image as a 2D array of integers. We want to create a function that rotates the image, storing the new pixel data back into the original array. The declaration for such a function could look like this:

```
int rotateImage(int image[4][4]);  /* rotate image, return status */
```

The *status* return value tells us whether the rotation was successful, or whether there was some sort of error.

If we want the function to work for an image with a variable number of rows, we can leave off the leftmost dimension. Then we would need to pass in the number of rows in the image.

```
int rotateImage(int image[][4], int rows);
```

Another way to express the type of the first argument is "pointer to an array of 4 integers." This would be written as:

```
int rotateImage(int (*image)[4], int rows);
```

The parentheses are needed because the [] has a higher precedence than the *. The type:

```
int *image[4]
```

would mean "array of four integer pointers," not "pointer to an array of four integers."

For more discussion about multi-dimensional arrays, see pages 356-360 and 378-386 in the *C Primer Plus* textbook.

## Program Specification

The program for this assignment must perform a sequence of commands, specified by the user. These commands include:

1. Read in data from an image file (in PGMA format), storing the data into a two-dimensional array of integers. For simplicity, we will only be working with square images and sub-images, where the width and height are equal. The largest image that your program must handle is 80×80 pixels.
2. Perform a transformation to the image or a portion of the image. These transformations will be stored "in-place," in the same image data array used to read and write image files.
3. Write the image to a file in PGMA format.
4. Quit.

### *User Interface*

There will be no prompting by the program in this assignment. The program will simply start reading commands from stdin, and will keep reading and performing commands until (a) the user enters the

"quit" command, or (b) the program encounters a fatal error. There are only two fatal errors: (1) unable to open a file for reading, and (2) unable to open a file for writing.

The commands are specified in the following table. All commands except for quit (qq) take one or more arguments. The *filename* argument is a string, no more than 16 characters (not including the null terminator). The *sx* parameter is the starting x-coordinate (in the horizontal direction) of the region to be transformed. The *sy* parameter is the starting y-coordinate (in the vertical direction) of the region to be transformed. The *size* argument is the number of pixels in each direction (x and y).

| ***Command*** | ***Arguments*** | ***Description*** |
|---|---|---|
| **`rf`** | *filename* | Read image data from the PGMA file. |
| **`wf`** | *filename* | Store image data in PGMA format to the file. |
| **`rr`** | *sx sy size* | Rotate right (clockwise) by 90 degrees. |
| **`rl`** | *sx sy size* | Rotate left (counter-clockwise) by 90 degrees. |
| **`ru`** | *sx sy size* | Rotate 180 degrees. |
| **`fv`** | *sx sy size* | Flip the image around its vertical middle axis. |
| **`fh`** | *sx sy size* | Flip the image around its horizontal middle axis. |
| **`ig`** | *sx sy size* | Invert the grayscale values of the image. |
| **`qq`** | | Quit. |

The (x,y) coordinate of the top left corner pixel is (0,0). The x-coordinate increases to the right, and the y-coordinate increases in the down direction. (For example: the bottom left corner of a 16×16 image would be (0,15).)

### *Functions*

In addition to main, you must use the following function declarations, and write function definitions for each of them. The function definitions must come ***after*** the main function in your program. (The function declarations come before main, so that the functions can be called from main.)

```
/* in each case, function returns 0 if successful,
   or 1 if some error occurred */

/* read a PGMA file and store data into imageData array */
int readPGMA(char *filename, int imageData[80][80]);

/* write image data to a PGMA file */
int writePGMA(char *filename, int imageData[80][80]);

/* rotate part of image to right by 90 degrees */
int rotateRight(int imageData[80][80], int sx, int sy, int size);

/* rotate part of image to left by 90 degrees */
int rotateLeft(int imageData[80][80], int sx, int sy, int size);

/* rotate part of image by 180 degrees */
int upsideDown(int imageData[80][80], int sx, int sy, int size);

/* flip part of image around its vertical axis
   NOTE: this means the vertical axis of the specified portion,
   not the entire image */
int flipVertical(int imageData[80][80], int sx, int sy, int size);
```

```
/* flip the image around its horizontal axis
   NOTE: this means the horizontal axis of the specified portion,
   not the entire image*/
int flipHorizontal(int imageData[80][80], int sx, int sy, int size);

/* invert grayscale color of part of image */
int invertGray(int imageData[80][80], int sx, int sy, int size);
```

If the input file cannot be opened in readPGMA, the function must return a value of 1.

If the output file cannot be opened in writePGMA, the function must return a value of 1.

For the other functions, an error value (1) is returned if (sx, sy) is not within the image, or if the size parameter specifies a region that falls outside of the image. If this happens, then the image data must not be changed – in other words, the command will be ignored.

*Example:* If the image is 40×40, and the command is "ig 10 10 40", then the command is ignored, because a 40×40 region starting at (10,10) falls outside the boundary of the image.

If any function is called prior to readPGMA – in other words, if no image has been read – then the function must return immediately with a value of 1.

An example of each transformation is given in Appendix C.

### *Global Variables*

You may use global variables to store the following information: width of image, height of image, maximum grayscale value of image. All three of these values are read from the PGMA file.

You ***may not*** use global variables to store the image data, or any other information other than the above.

### *File Format*

The PGMA file format is described in Appendix A. The test files that we give you will all be square images (width = height) and will be, at most, 80 pixels in each dimension. The maximum grayscale value (representing "white") is included in the file data. Don't assume that this will be 255 for all test files!!!!

Wherever you need whitespace in your output file, use a single newline character ('\n'). In other words, your output file will have exactly one decimal integer per line. (With the exception of the first line, which contains a two-character string.) There must be **no comments** in the output file that you create.

**It is crucial that you follow the file format exactly.** We will not be looking at your images – we will be comparing your output files to ours. If the format is off, then your output won't match ours, and you will lose points.

### *The main() Function*

The main function for this program is responsible for reading commands and their arguments, and for calling the function to carry out each command.

If an illegal command is entered, the program should discard the remaining character on that line and ignore the command. You may assume that a legal command will have the correct number and type of arguments needed for that command. (In other words, you don't need to worry about the user providing only two numbers for one of the transformation commands.)

If the readPGMA() or writePGMA() function returns an error, the program should exit immediately. If any of the transformation functions return an error, it is ignored – the program should continue with the next command.

NOTE: If you have not implemented the function needed for a command, your main() function must still recognize that command as legal, and move on to the next command as if the function were called correctly.

## Administrative Info

*Updates or clarifications on the Message Board:*

Any corrections or clarifications to this program spec will be posted on the Message Board. It is important that you read these postings, so that your program will match the updated specification.

*What to turn in:*

- Source file – it must be named **image.c**. Submit via Wolfware to the Program 3 assignment in your ECE 209P section (not 209-001). The program will be graded by your 209P TA, and your grade will be posted on the 209P gradebook.

*Grading criteria:*

15 points: File submitted with the proper name.

15 points: Compiles with no warnings and no errors (using –Wall and –pedantic). (If the program is not complete, as described above, then only partial credit is available here. In other words, you won't get 15 points for compiling a few trivial lines of code.)

10 points: Proper coding style, comments, and headers. No unnecessary global variables. No goto.

15 points: The main() function correctly processes commands and deals with errors.

15 points: The readPGMA() and writePGMA() functions are complete and correct. The output PGMA file must exactly match the specification in Appendix A.

30 points: The six transformation functions are complete and correct. (5 points each)

## *Appendix A: PGMA Format*

Source: http://people.sc.fsu.edu/~burkardt/data/pgma/pgma.html

PGMA is a file format that uses ASCII text for storing grayscale image data. It can be created and consumed by many different image manipulation programs. (See Appendix B.)

A PGMA file contains the following information, in the order given:

- A two-character string "P2" that identifies this as a PGMA file.
- Whitespace
- Width of the image, formatted as a decimal integer
- Whitespace
- Height of the image, formatted as a decimal integer
- Whitespace
- Maximum gray value, formatted as a decimal integer
- Whitespace
- Width * height gray values, each in ASCII decimal, between 0 and the specified maximum value, separated by whitespace, starting at the top-left corner of the graymap, proceeding in normal English reading order. A value of 0 means black, and the maximum value means white.

In addition, characters from '#' to the next newline character ('\n') are ignored (comments). No line should be more than 70 characters.

For the output of this program, "whitespace" must be interpreted as a single newline character. No comments may be used in the output file for this program. The output file must have exactly the same width, height, and maximum gray value as the input file.

These restrictions do not apply to the input file for this program – the input file may contain comments, and there may be any number of whitespace characters between file elements. Since most fscanf format codes skip whitespace characters, this will not impact the code for the readPGMA function. However, your code must be able to ignore comments. (See Appendix D for tips on how to do this.)

Note the order of pixel elements in the file. It says "English reading order," which means left to right, top to bottom. The first number is pixel (0,0), then (1,0), then (2,0), etc.

## *Appendix B: Running the Program and Viewing the Images*

There are a variety of image processing applications that can read and display PGMA files. GIMP (GNU Image Manipulation Program) is one that runs on Windows, Linux, and Mac OS X.

On the EOS Linux machines, GIMP is available as /usr/bin/gimp.

To install GIMP on your Windows or Mac laptop, see http://www.gimp.org.

You can also create your own images, or convert them from other formats to PGMA. Remember that your code is designed to work only with square images, which must be no more than 80 pixels in each dimension.

Instead of typing commands over and over, you might want to create a text file that contains a sequence of commands. You can then redirect standard input to that file, and scanf will read from there instead of the keyboard.

For example, create a file named commands.txt that contains the following:

```
rf homer.pgm
rr 0 0 80
ic 20 20 40
wf homer2.pgm
qq
```

Then run your program like this:

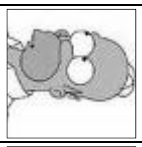
```
eos% ./image <commands.txt
```

Then use GIMP to look at the output image, to make sure that the proper transformations were done. You can also edit the PGMA file with a regular old test editor, of course, but it won't be easy to tell if the output is correct.

We will provide sample input files, command files, and output files to help you test your program. (You can "diff" your output file with ours, which is what we will do when grading your program.)

## *Appendix C: Examples of Transformations*

Each of the transformations below begins with the same original 64x64 image (homer.pgm). The box is not part of the figure – it has been added for clarity.

| Original Image | Transformation | Result |
|---|---|---|
|  | `rr 0 0 64` |  |
|  | `rl 0 0 64` |  |
|  | `ru 0 0 64` |  |
|  | `fv 0 0 64` |  |
|  | `fh 0 0 64` |  |
|  | `ig 0 0 64` |  |

All of these examples perform the transformation on the entire image. We recommend that you get that working first, and then modify the code to transform regions.

## *Appendix D: Tips and Hints*

Q: How do I remove comments in the input PGMA file?

A: Except for the first two characters ("P2"), you should be reading integers from the file. The return value from fscanf will tell how many integers were scanned and assigned when fscanf was called. So, if we read one integer (using "%d") and fscanf returns 0, we know that fscanf encounter a non-digit character.

Then use fscanf (or getc) to read that character and see whether it's '#'. If so, you have just found the beginning of a comment. Use fscanf (or getc) to read and discard characters until a newline character is read.

Q: Should I use string I/O or character I/O?

A: For reading the first two characters of the PGMA file, you can use either. But it would seem easier to read a two-character string.

Same thing for the commands – read a two-character string. Then, depending on the command, either read a string for the filename, or three integers for the command arguments. (Or nothing, if the command is "qq".)

Q: How do I change the image from inside a function?

A: Remember that the image array is being passed by reference. So any changes you make to the array elements will also be seen by the caller.

We recommend creating a local array inside the function, and writing the new values there. Then you can copy the data from the local array into the array that was passed in.

Q: How do I implement the transformation functions?

A: See the earlier section about image transformations. For the rotate and flip functions, you're just moving pixel data from one place to another. For each pixel in the original image, think about where you want that pixel to end up.

Don't try to do this "manually" with fixed values. The sizes of images and regions will change from call to call. So you need to think of some mathematical relationship between the starting position and ending position of the pixel.

For the "ic" command, you simply want to subtract the current pixel value from the maximum gray value specified by the input file.

Q: Are all the images 80×80 pixels?

A: No, but the maximum size is 80×80. All images will be square.

Q: Can I assume that the maximum gray value is 255?

A: No.

Q: Can I use a char array for the image data, rather than an int array?

A: No. We have no guarantee that the maximum gray value will fit in a char. Also, the functions all expect a two-dimensional integer array as the argument. Passing a char array instead of an int array won't work.

Q: What if the filename argument is longer than 16 characters?

A: We will not test your code with a filename longer than 16 characters. If you want to be safe, though, you should truncate the input at 16 characters, and then discard all of the remaining input characters, up to the next newline.

Q: Does the program exit after an image file is written?

A: No. You can perform additional transformations, read another file, output to another file, etc. The program keeps going until the user enters the "qq" command.

Q: I don't understand two-dimensional arrays.

A: That's not a question. Read pages 356-360, 378-386 in *C Primer Plus*. If you still don't understand, talk to your instructor, TA, or fellow students.

Q: Do I have to implement the functions listed in the specification?

A: Yes.

Q: Can I implement other functions in addition to the ones listed in the specification?

A: Of course.

Q: Can I split my program across multiple files?

A: No. You must submit exactly one source code file. We'll probably use multiple files for Program 4.

Be sure to read the Message Board for more questions, answers, changes, and clarifications. If you have a question, please post it to the message board, so that we don't have to answer it a dozen times. If you have a question that requires sharing code, send email to ece209-sup@wolfware.ncsu.edu.

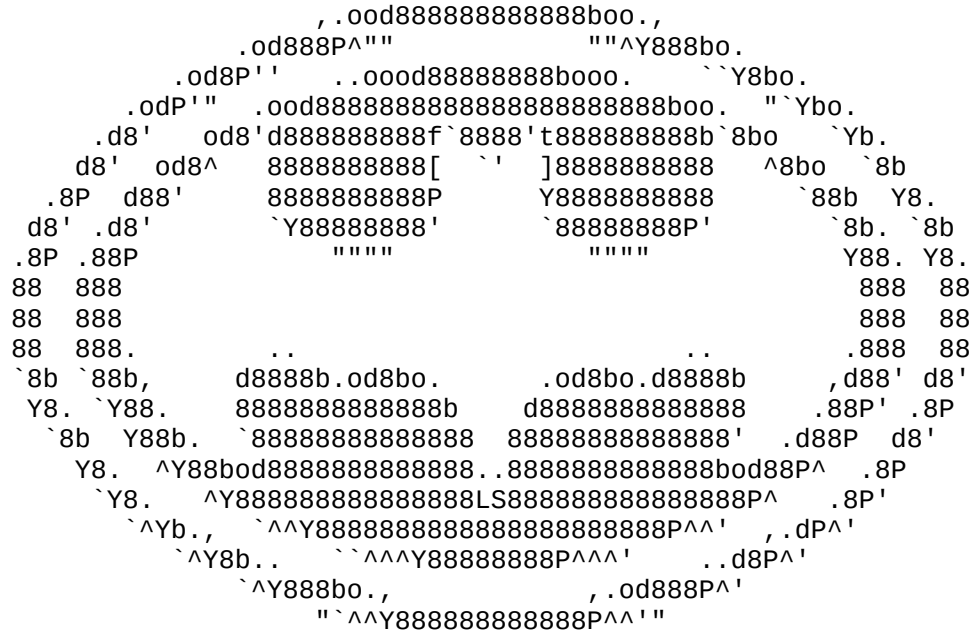
## *Appendix E: Just for Fun*

If you finish the program early, and are looking for a little more fun, try implementing a "wa" command that writes the image as an "ASCII art" file.

In ASCII art, we use a single ASCII character to represent each pixel. Different characters have different amounts of "ink," so they can correspond to different levels of gray. For example, the following list of 16 characters can be used to represent 16 gray levels, from darkest to lightest.

```
M # @ 8 O o + i = I ; : ~ . '  (space)
```

Map the grayscale value for the each image pixel to one of these 16 characters. Print a row of characters, followed by a newline, for each row of pixels. Then you have an ASCII file that looks like the image, something like this:




(Picture copied without permission from http://www.geocities.com/SouthBeach/Marina/4942/ascii.htm. This picture uses a different mapping than the one described above. You can play around with the characters used, and/or the number of characters in the list, to see what yields better results.)

### ***Another Idea***

How would you change your program to deal with color images? See the PPMA file format description at http://people.sc.fsu.edu/~burkardt/data/ppma/ppma.html.