Lab 6: Part 2: Functional Decomposition ...

Due date: Monday, November 16, 8:00am.

Note: November 16 is the **instructor's furlough day**. You will receive **Lab 8** assignment on Friday, November 13, and you are expected to start working on the Lab 8 assignment on Monday during the lab period.

Lab Assignment

Assignment Preparation

Lab type. This is an **team programming lab**. For this lab you will organize into teams of four-to-five people (one team will be of size five). You get to pick your team partners for the lab. I encourage you to form the teams in the following manner: create a team out of two pairs who worked on Lab 6-2 assignment. Alternatively, pair up with someone else you've already worked with, and find another pair of students.

Each team can use the Lab 6 code created by any of the team members (or his/her pair). *No other code sharing between teams on this lab assignment is allowed.*

Purpose. This lab will test your ability to construct functions, test them, and use them to solve simple problems. It will also allow you to create and maintain C's .h *header files.*

Programming Style. All submitted C programs must adhere to the programming style described in detail at

http://users.csc.calpoly.edu/∼cstaley/General/CStyle.htm

When graded, the programs will be checked for style. Any stylistic violations are subject to a 10% penalty. Significant stylistic violations, epsecially those that make grading harder, may yield stricter penalties. Also note the the Lab 2 requirement for the content of the header comment in each file you submit applies **to each assignment** (lab, programming assignment, homework) in this course.

Testing and Submissions. Any submission that does not compile using the

```
gcc -ansi -Wall -Werror -lm
```

compiler settings will receive an automatic score of 0.

Please, make sure you test all your programs prior to submission! Feel free to test your programs on test cases you have invented on your own.

Problem Overview

Each team will create two sets of deliverables: a library of C functions that draw a variety of shapes and objects on a PPM image, and a collection of programs that test functions in your new library, and use these functions to create "interesting" images.

The functions from your graphics library will use the functions from the instructor's `images.h` library and create more complex objects (mostly inspired by the assignments you had to complete in Lab 6-2, with some extra objects thrown in).

Creation and maintenance of header files and function libraries

The function library you will be creating is described in a `graphics.h` header file, that is made available to you by the instructor. You will need to implement **all** functions that are declared in `graphics.h`.

To do so, you will create a C program file `graphics.c` and will include **definitions** of all `graphics.h` functions in it. The `graphics.c` file will look as follows:

```
/* CPE 101 Fall 2009 Alex Dekhtyar (instructor) */
/* Team <Team Name>:                            */
/* <Student 1>                      <Student 2> */
/* <Student 3>                      <Student 4> */
/* Graphics library implementation              */

#include "graphics.h"              /* functions you are implementing */
#include <math.h>                  /* you will need it               */
#include <stdio.h>                 /* for debugging purposes only    */

#define MY_CONST1 0.5 /* #define any constants you need for the implementation */
/* .... */



/* putSmiley():  draw an image of a smiley face       */
/* char image[][][] : image array                     */
/* x,y : center of the smiley face                    */
/* radius : radius of the smiley face                 */
/* color: color of the smiley face                    */

void  putSmiley(char image[][WIDTH][COLORS], int x, int y, int radius, char color[]) {
  /* implementation of putSmiley() goes here */
```

```
 return;
}

/* other functions implemented below */
```

Compilation. The version of your `graphics.c` file that you submit should not contain `main()` function. (note, you may choose to include a temporary `main()` for debugging purposes, but it has to be removed later). Instead of compilation into an `executable` file, you will compile `graphics.c` into a binary object, `.o` file and add this file when compiling your "drawing" programs.

To compile `graphics.c` into an object file, execute the following command:

```
> gcc -ansi -Wall -Werror -c graphics.c
```

The `-c` flag indicates to `gcc` that only the *compiler* (and not the *linker*) should run. The result of executing this command (assuming your program compiles) is a `graphics.o` file located in your current directory.

Suppose now, that you have written a `C` program `testGraphics.c` that uses the functions from the graphics library you have written. In order to properly compile your program, you need to issue the following command:

```
> gcc -ansi -Wall -Werror -lm -o testGraphics testGraphics.c image.o graphics.o
```

Note, that you should include both `image.o` and `graphics.o` files in your command, as the code in the `graphics.c` file (and hence, in the `graphics.o` file) relies on `image.o` to provide the code for the "low-level" functions (the ones you used in Lab 6-2).

Use of instructor's object file. You are provided with the `C` object file `graphics-alex.o`. This file is the compiled version of instructor's `graphics.c` program which implements all functions you need to implement for this assignment. This is done for *testing purposes*. You should be able to compile your test programs using the instructor's `graphics-alex.o` library in place of your `graphics.o` implementation.

To obtain an executable file that uses the instructor's `graphics-alex.o` binary, you need to issue the following command:

```
> gcc -ansi -Wall -Werror -lm -o testGraphics testGraphics.c image.o graphics-alex.o
```

Working as a Team: Implementation and Testing

This your first team assignment in the CPE 101 course. Generally speaking, the *key distinction* between *team assignments* and *pair programming* assignments is that when working as a team you **must** distribute the work between team members.

For this project, you will be distributing the work as follows. For each functions from the `graphics.h` library you need to designate two sub-teams:

1. **Implementation team.** This *sub-team* is responsible for creating the code of the given function as part of the team's `graphics.c` file.

2. **Test team.** This *sub-team* is responsible for creating a number of test programs that **validate** (i.e., confirm that it works correctly) the work of the function implemented by the implementation team.

Work distribution. In general, on a single task, both the implementation and the test teams are expected to include two people[1]. You can manage your work as follows. Pair up within your teams (e.g., by pairing with your previous lab partner). Split the functions in `graphics.h` into two roughly equal groups. Each pair of students within the team becomes the implementation team on one of the groups of functions and then, it becomes the test team on the second group of functions.

Testing in the context of this assignment. To test an implementation of a specific function you need to write a short C program that uses the function you are testing in a variety of contexts. For example, to test that your `putSmiley()` function works correctly, you need to verify that you can draw smiley faces of different size (radius) and different color.

Your test programs can be developed independently of the actual function implementations, in fact, for *some functions*, (e.g., `putCar()` and `putHouse()`, which require considerable development, it is extremely useful to have your test programs written before the functions are completed (and perhaps before the development of them starts). You can test your test programs, in turn, by compiling them using the instructor's library `graphics-alex.o` — compile your programs using it, produce the PPM file, see if it represents what you wanted, then *pass your test program to the implementation team.*

In general, you can create as many test programs for your functions as you deem necessary. Your test programs may **combine** the use of different functions, but, in general, each program needs to test only **one function** (this is known as **unit testing**).

You will be asked to submit two test programs for *some of the functions* (some functions, e.g., `putRoof()` will be used within other functions you are developing, so you won't need to submit your test programs for them, although you may need to generate some tests for those functions as well).

A sample test program for testing one of the functions is made available to you. I am also making available the PPM files illustrating the results of running some of my test programs using `graphics-alex.o`.

graphics.h Library

This section documents the functions you have to implement for this assignment.

The `graphics.h` header file declares the following **eleven (11)** functions.

```
void  putSmiley(char image[][WIDTH][COLORS], int x, int y, int radius,
                char color[]);
void  putYingYang(char image[][WIDTH][COLORS], int x, int y, int radius,
                  char ying[], char yang[]);
void putHouse(char image[][WIDTH][COLORS], int topX, int topY,
              char color1[], char color2[]);
```

[1] If you are a team of five students, one of the sub-teams will, of course, be of size three.

```
void putCar(char image[][WIDTH][COLORS], int topX, int topY,
            char color[]);
void putBrickWall(char image[][WIDTH][COLORS], int topX, int topY, int width, int height,
                  int brickWidth, int brickHeight,
  char color1[], char color2[]);
void putCross(char image[][WIDTH][COLORS], int topX, int topY,
              int width, int height,
              int thickness, char color[] );
void putCrescent(char image[][WIDTH][COLORS], int x, int y, int radius,
                 int orientation, int shift, char color[], char bgColor);
void putStar(char image[][WIDTH][COLORS], int points[][2], char color[]);
void putBrick(char image[][WIDTH][COLORS], int topX, int topY,
              int width, int height,
              char color1[], char color2[]);
void putWindow(char image[][WIDTH][COLORS],int topX, int topY,
                    int width, int height, char color[]);
void putRoof(char image[][WIDTH][COLORS], int startX, int startY,
              int width, char color[]);
```

Note that `graphics.h` contains the `include "image.h"` directive. In general, your implementations of `graphics.h` functions should use the graphics primitives from the `image.h` library.

The brief summary of the functions is in the table below:

Function name	Meaning
`putSmiley()`	draw a smiley face of a given color and size
`putYingYang()`	draw a ying-yang symbol of given colors and size
`putCross()`	draw an equal-sides cross of a given color
`putCrescent()`	draw a crescent of a given size, color and direction
`putStar()`	draw a five-point star given point coordinates
`putBrick()`	draw a brick of given size and color
`putBrickWall()`	draw a brick wall of given size and color
`putWindow()`	draw a window with a frame
`putRoof()`	draw a triangular roof
`putHouse()`	draw a house inside a 100×100 box
`putCar()`	draw a car inside a 100×100 box

Functions in Detail

`void putSmiley(char image[][WIDTH][COLORS], int x, int y, int radius, char color[]).` The parameters for this function are:

char image[][WIDTH][COLORS]	image array
int x, int y	the center of the smiley face
int radius	the radius of the smiley face
char color[]	color of the smiley face

This function draws a smiley face centered at the `(x,y)` coordinates with the radius `radius` and of the color specified in the `color` array. Use your Lab 6-2 smiley face program for inspiration.

`void putYingYang(char image[][WIDTH][COLORS], int x, int y, int radius, char ying[], char yang[]).` This function takes the following parameters:

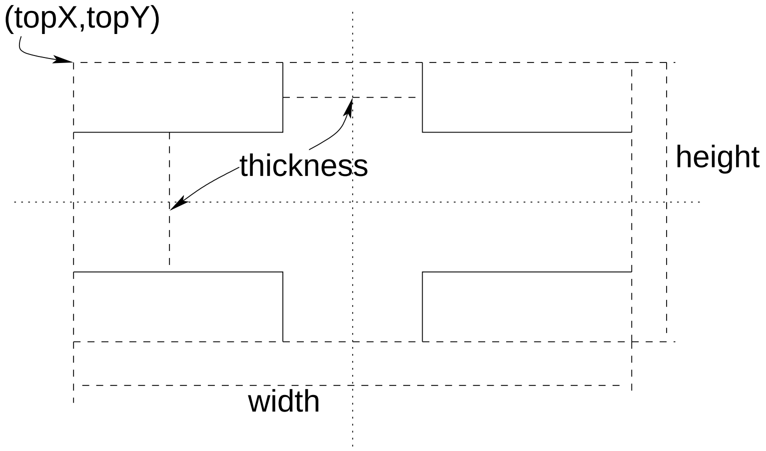

Figure 1: The anatomy of a cross.

char image[][WIDTH][COLORS]	image array
int x, int y	the center of the ying-yang symbol
int radius	the radius of the ying-yang
char ying[]	color of the ying (top) portion
char yang[]	color of the yang (bottom) portion

This function draws a ying-yang symbol centered at the `(x,y)` coordinates with the radius `radius` and of the colors specified in the `ying` and `yang` arrays. Use your Lab 6-2 ying-yang program for inspiration.

`void putCross(char image[][WIDTH][COLORS], int topX, int topY, int width, int height, int thickness, char color[] ).` This function takes the following parameters:

char image[][WIDTH][COLORS]	image array
int topX, int topY	top left corner of the bounding box
int width, int height	the width and the height of the bounding box
int thickness	the thickness of the cross
char color[]	color of the cross in `putCross()` function.

This function draws an equal-width cross inside the bounding box specified by the top left corner `(topX, topY)` and the `width` and the `height`. The cross is `thickness` pixels thick and its color is represented by the `color` array.

Note, that the `(topX, topY)` parameters, represent the coordinates of the bounding box for the cross, NOT the coordinates of a corner of the actual cross bar. The chart in Figure 1 should help you understand the meaning of the parameters for this function.

`void putCrescent(char image[][WIDTH][COLORS], int x, int y, int radius, int orientation, int shift, char color[], char bgColor).` This function takes the following parameters:

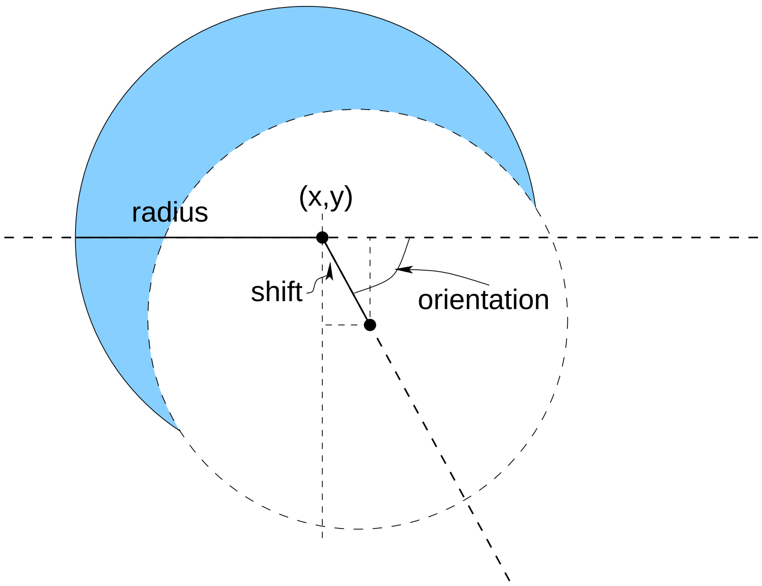

Figure 2: The geometry of a crescent in the `putCrescent()` function.

char image[][WIDTH][COLORS]	image array
int x, int y	center of the crescent image
int radius	radius of the crescent
int orientation	the angle (in degrees) of the crescent
int shift	the shift of the background circle
char color[]	color of the crescent.
char bgColor[]	background color.

This functions draws a crescent. If you successfully implemented drawing a crescent in Lab 6-2, it should come as no surprice to you, that a crescent is formed out of two circles: a foreground one and a background one. The first, foreground, circle, centered at `(x,y)` has a radius `radius` and its color is found in the `color` array. The second, background, circle should have the color of the background, found in the `bgColor` array. This circle should have a shifted center, and may have a different radius. The `putCrescent()` function received parameters that control the shift of the center, but the radius of the background circle is to be determined by you. It may be set to a constant (e.g., 0.9 of `radius`), or it may be set to be a function of the `radius` and `shift`.

The `shift` is the distance of the center of the background circle from the center of the foreground circle. The exact position of the center is guided by the `orientation` parameter, which is an angle (in degrees) at which the center is shifted. Figure 2 illustrates the geometry of the crescent parameters.

`void putStar(char image[][WIDTH][COLORS], int points[][2], char color[]).`
The function takes the following parameters:

char image[][WIDTH][COLORS]	image array
int points[][2]	the array of points for the star
char color[]	color of the star

This function draws a five-point star by connecting with lines the points stored in the `points` arrays. The color of the lines resides in the `color` array.

The `points` array must have at least five points in it. If it has more than five points, use the *first five points* stored in the array. `points[i][0]` is the *vertical* (row) coordinate of the pixel `i`, while `points[i][1]` is the *horizontal* (column) coordinate of the pixel.

Note, that depending on where the points are located, the shape drawn may by an arbitrary pentangle, not a star. This is ok, and is expected. You are responsible for drawing a proper star **if the points are provided in the counterclockwise order**.

`void putCar(char image[][WIDTH][COLORS], int topX, int topY, char color[])`. This function takes the following parameters:

char image[][WIDTH][COLORS]	image array
int topX, int topY	top left corner of the bounding box
char color[]	color of the car body

The function draws a car that fits into a 100×100 pixel bounding box with the top left corner located at `(topX, topY)`. The `color` array is the color of the body of the car. You should feel free to use other colors to draw various parts of the car.

The actual shape of the car drawing is up to you. Instructor's version of the car image is different from the car image drawn for the Lab 6 assignment. You are not expected to match either of them specifically (although you are not prohibited from that either).

This function produces cars all going in the same direction. This is one of its chief limitations. Instructor's car goes from right to left, but you may orient your car in any direction.

Drawing a house

The next collection of functions (culminating with the `drawHouse()` can be used in cohort to draw a house.

`void putBrick(char image[][WIDTH][COLORS], int topX, int topY, int width, int height, char color1[], char color2[])`. This function takes the following parameters:

char image[][WIDTH][COLORS]	image array
int topX, int topY	top left corner of the brick
int width, int height	width and height of the brick
char color1[]	color of the brick
char color2[]	color of the cement

This function draws a single birck surrounded by a thin layer of cement. The color of the brick is in the `color1` array, the color of the cement `color2` array. (`topX, topY`) is the top left corner of the drawing, `width` and `height` are the width and the height respectively of the brick/cement combination.

Note, that the actual dimensions of the brick are a bit smaller than the `width` and `height`. You have the responsibility to determine how thick the layer of cement around the brick should be (it must be at least one pixel thick, and, the larger the brick, the thicker the layer of cement should become in pixels).

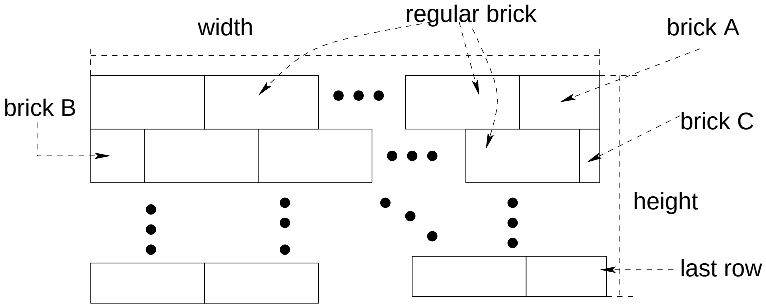

Figure 3: The geometry of a brick wall in the `putBrickWall()` function.

`void putBrickWall(char image[][WIDTH][COLORS], int topX, int topY, int width, int height, int brickWidth, int brickHeight, char color1[], char color2[]).` This function has the following parameters:

char image[][WIDTH][COLORS]	image array
int topX, int topY	top left corner of the brick wall
int width, int height	width and height of the brick wall
int brickWidth, brickHeight	width and height of the bricks
char color1[]	color of the bricks
char color2[]	color of the cement

This functions draws a brick wall whose top left corner is at `(topX, topY)` with the width and height stored in `width` and `height` parameters respectively. Each *regular* brick in the wall should have the `brickWidth`, `brickHeight` dimensions (including the surrounding cement). The color of the brick is in `color1` array; the color of the cement is in `color2` array.

The assumption is that `putBrickWall()` will use `putBrick()` to draw individual bricks in the wall.

The mechanics of building a wall out of individual bricks is illustrated in Figure 3. As seen from the diagram, all rows of the wall, except the last row should have the same *height*, while the height of the last row is essentially *"whatever's left"*. The wall consists of two types of rows that alternate. Rows of the first type start with a full regular brick, and might have an incomplete brick (brick A on the diagram). Rows of the first type start with a half-brick (brick B on the diagram), followed by regular bricks, and ending, possibly with another incomplete brick (brick C) on the diagram.

The computation of the sizes of all incomplete bricks, as well as the height of the final row of the wall is left to you.

Notice that in the paragraph above a "brick" is the result of the `putBrick()` function, so it is an actual brick surrounded by a layer of cement.

`void putWindow(char image[][WIDTH][COLORS],int topX, int topY, int width, int height, char color[]).` The parameters for this function are:

char image[][WIDTH][COLORS]	image array
int topX, int topY	top left corner of the window
int width, int height	width and height of the window
char color[]	color of the window

This function draws a window in a window frame, with the top left corner at `(topX, topY)` and with the specified `width` and `height`. The window should have the color passed in the `color` array. The color of the frame should be selected by your function internally.

Essentially, this is the same function as `putBrick()`, except that the frame color is fixed internally (while `putBrick()` requires a second color).

`void putRoof(char image[][WIDTH][COLORS], int startX, int startY, int width, char color[]).` This function takes the following parameters:

char image[][WIDTH][COLORS]	image array
int startX, int startY	bottom left (starting) corner of the roof
int width	width of the roof at the bottom
char color[]	color of the roof

This function draws a triangular roof whose bottom left corner is `(startX, startY)`. The width of the bottom of the roof is `width` and the color of the roof is passed in the `color` array.

`void putHouse(char image[][WIDTH][COLORS], int topX, int topY, char color1[], char color2[]).` The parameters for this function are:

char image[][WIDTH][COLORS]	image array
int topX, int topY	top left corner of the bounding box
char color1[]	main color of the house
char color2[]	accent color of the house

This function draws an image of a house inside a 100×100 bounding box whose top left corner coordinates are `(topX, topY)`. The function should use two colors, represented in arrays `color1` and `color2` to draw the house.

It is assumed that `putHouse()` uses in its code the `putRoof()` and `putWindow()` functions. The instructor's version of this function uses `putBrickWall` to create the walls of the house. It uses `color1` for the brick color and `color2` as the window color.

Your use of colors may vary, but both colors passed to your function must be present on the image. You may also use additional colors for parts of the image (e.g., cement color).

You are not expected to replicate the exact house image produced by the instructor's code, although neither are you prohibited.

As in Lab 6-2, your images of the house (and the car) must be similar or better in the overall complexity (number of elements used), and should be recognizable.

Programs to submit

You will submit the following programs.

graphics.c . Your implementations of all `graphics.h` functions should reside in a file named `graphics.c`.

Note: you may use multiple files during the development process, to ensure that each team member does not interfere with the rest of the team's work, but **at the end** all function implementations **must be merged** into a single `graphics.c` file.

Test programs. You will submit two test programs for each of the following functions:

1. `putSmiley()`
2. `putYingYang()`
3. `putCross()`
4. `putCrescent()`
5. `putStar()`
6. `putBrickWall()`
7. `putHouse()`
8. `putCar()`

The naming scheme for your files is as follows. The test files for `putNAME()` function should be named `testNAME01.c` and `testNAME02.c`. For example, for `putSmiley()` the test files will be `testSmiley01.c` and `testSmiley02.c`.

Please note, that your test programs should work **both** with your `graphics.c` implementations, **as well as** with the **instructor's version**. That is, compiling your programs using either your `graphics.o` binary object file or using the instructor's `graphics-alex.o` binary object file, should result in a working executable that outputs the expected PPM image. Note that images may be different due to differences in implementation of individual functions (your car shape, or window frame color, etc... may be different from mine).

Mystery images. Each team shall submit **four (4)** programs that draw relatively complex images using the functions from the `graphics.h` library. (the programs may also use the `image.h` library functions, but only as auxillary ones — the main purpose of these images is to demonstrate what you can do with the new functions). Name your programs `drawing01.c`,...,`drawing04.c`.

Submission.

Files to submit. Submit 22 files:

```
team.txt,
graphics.c,
testSmiley01.c,
testSmiley02.c,
testYingYang01.c,
testYingYang02.c,
testCross01.c,
testCoss02.c,
testCrescent01.c,
testCrescent02.c,
testStar01.c,
testStar02.c,
testBrickWall01.c,
testBrickWall02.c,
testHouse01.c,
```

`testHouse02.c`,
`testCar01.c`,
`testCar02.c`,
`drawing01.c`,
`drawing02.c`,
`drawing03.c`,
`drawing04.c`

Files can be submitted one-by-one, or all-at-once, **BUT ALL FILES SHOULD BE SUBMITTED BY THE SAME PERSON IN THE GROUP!** Please choose the designater "submitter" in advance and make sure this student has all files necessary to submit.

Submission procedure. You will be using `handin` program to submit your work. The procedure is as follows:

- `ssh` to `vogon` (`vogon.csc.calpoly.edu`).

```
> handin dekhtyar lab07  <your files go here>
```

`handin` is set to stop accepting submissions 24 hours after the due time.

Testing and Grading

Any submitted program that does not compile earns 0 points. Please download an run instructor's test to see the exact output produced. Your programs are expected to match this output.

Appendix A: Instructor's Graphics Library `image.h`

The instructor's graphics library is provided to you in two files:

- `image.h`: is the header file for the library. It contains **function declarations** for all functions implemented by the instructor. Additionally, it contains a number of symbolic constant definitions.
- `image.o`: is the **object file** that contains the binaries of instructor's implementation of all functions declared in `image.h`. In order to use instructor's functions, you will **link** the `image.o` file to your code during compilation time. Instructions are given below.

Image Primitives Library: overview

The image primitives library can be used to produce Portable Pixel Map files of predefined size. The size of the image is defined as a pair of symbolic constants, `HEIGHT` and `WIDTH` in the `image.h` file.

An *graphics primitive* is a C function that draws a single simple shape. For example, `image.h` header file declares functions that draw a single point, a line, a circle, a rectangle and an ellipse, among others.

Most functions declared in the `image.h` header file take as input (via the call-by-reference mechanism) a 3dimensional array representing the color assignment to `WIDTH`× `HEIGHT` pixels. This array is similar to the ones you used in your Lab 5 assignments.

Each graphics primitive works by changing the colors of a specific set of points (determined by the type of the function) from its input image array. For example, a function that draws a line between two points, computes which pixels along the way lie on the line between two points specified as function parameters, and colors these pixels according to the color, another parameter of the function.

For simplicity, all graphics primitives take as an input parameter an `char` array of size 3, that represents a triple of RGB color intensities. Each graphics primitive uses only one color to "paint" the image, namely, the color passed to it as a parameter using such an array.

The `image.h` file contains declarations of a number of auxillary functions. These functions may not be needed for you, but they are used by other functions in the `image.h` library.

image.h library description

`image.h` file declares a collection of symbolic constants and a list of functions loosely partitioned into three categories: graphics primitives, work with PPM format, and auxillary functions. All of these are described below.

image.h Symbolic Constants

The following symbolic constants are defined in the `image.h` file:

```
#define COLORS 3       /* number of colors components in an RGB color */
#define HEIGHT 400     /* height of the image produced                 */
#define WIDTH  600     /* width of the image produced                  */
#define PI 3.14159265  /* PI                                           */
```

At present, the `image.h` library is designed to produce images of size 600×400. In general, this can be modified by changing the values of `HEIGHT` and `WIDTH` constants.

Additionally, the `image.h` file declares the following symbolic constants designed to represent RBG colors:

```
#define BLACK 0,0,0
#define WHITE 255, 255, 255
#define RED 255, 0, 0
#define DARK_RED 128,0,0
#define BLUE 0,0,255
#define YELLOW 255,255,0
#define BROWN 140,70,20
#define CYAN 0,255,255
#define ORANGE 255, 128,0
#define GREEN 0,255,0
#define DARK_GREEN 0,128,0
#define MAGENTA 255,0,255
#define GRAY 128, 128, 128
#define LIGHT_GRAY 196, 196, 196
#define DARK_GRAY 64, 64, 64
```

Note: Some instructor's examples use different colors. Similarly, you are NOT restriced in your selection of colors for these assignments. The colors above are specified for your convenience.

image.h Functions

The following functions are declared in the `image.h` header file. For each function we provide its declaration, explain the meaning of all arguments and specify what the function does.

PPM Image functions.

```
void drawImage(char image[][WIDTH][COLORS]);
void printHeader(int w, int h);
```

Graphics Primitives.

```
void blankImage(char image[][WIDTH][COLORS], char color[]);
void putPixel(char image[][WIDTH][COLORS], int i, int j, char color[]);
void putRectangle(char image[][WIDTH][COLORS], int topY, int topX,
                   int height, int width, char color[]);
void putCircle(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius, char color[]);
void putPie(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius, int start, int end, char color[]);
void putRing(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius1, int radius2, char color[]);
void putArc(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius1, int radius2, int start, int end, char color[]);
void putEllipse(char image[][WIDTH][COLORS], int centerY, int centerX,
                   int radius1, int radius2, char color[]);
void putLine(char image[][WIDTH][COLORS],
             int fromY, int fromX, int toY, int toX,
     char color[]);
```

Auxillary functions.

```
void setColor(char color[], char r, char g, char b);
int getMin(int i, int j);
int getMax(int i, int j);
float getAngle(int i, int j);
```

These functions are briefly summarized in the table below:

Function name	Meaning
`drawImage()`	print contents of an image array
`printHeader()`	output the PPM file header info
`blankImage()`	fill image with chosen color
`outPixel()`	draw a single pixel of the image array
`putRectangle()`	draw a rectangle with given coordinates
`putCircle()`	draw a circle with given center and radius
`putPie()`	draw a segment of a circle
`putRing()`	draw a ring with given center and radii
`putArc()`	draw a segment of a ring
`putEllipse()`	draw an ellipse with a given center and radii
`putLine()`	draw a line connecting two points
`setColor()`	set the values in the input color array
`getMin()`	return the smaller of two numbers
`getMax()`	return the larger of two numbers
`getAngle()`	return an angle (in degrees) given a point in space

Detailed descriptions of the functions are found in your Lab 6-2 assignment.