Lab 10: Loops and Arrays...

Due date: Wednesady, February 22, beginning of the class.

Lab Assignment

Assignment Preparation

Lab type. This is a **pair programming lab**. You keep the same partner as in Lab 9.

Purpose. The lab allows you to practice the use of loops and arrays.

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.

Program Outputs must co-incide. Any deviation in the output is subject to penalties. **PLEASE, USE BINARY EXECUTABLES PROVIDED BY THE INSTRUCTOR!**. The exception is made in case of floating point computations leading to differences in the last few decimal digits. You can check whether or not a program produces correct output by running the `diff` command.

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.

The Task

You will write a few programs drawing PPM images using two-dimensional arrays. You will also write a few programs extending your prior assignments.

Next lections program: elections.c

The final version of the elections program will compute the election results using loops and arrays.

Input. The input to the program shall be the same as in Lab 3: a sequence of 52 numbers. All numbers except for the 29th input are either 0 or 1, the 29th input is an integer in the range $0 \ldots 5$.

The first input is a mode indicator. It it is equal to 1, the program runs in the verbose mode, if it is 0, the program runs in silent mode.

Organization. Your program will now declare a an array

```
int electoralCollege[] = {AL, AK, AZ, AR, ..., WA, WV, WY}
```

(here we assume that symbolic constants `AL,..., WY` are `#define`d in the program).

The program, as before will read the first input, determine the mode (verbose or silent), and then read 51 numbers representing the results of the state vote. The program shall keep track of the Obama's and McCain's vote totals, and, if verbose mode is selected, will inform the user about the winner of each state (just as in Lab 3). Only now, this will be done using a loop.

Don't forget the special case, Nebraska (input number 29 of the program).

Output. The output of the program shall be **almost the same as in Lab 7**. The only extra thing you need to report is how many states voted for each candidate. The output format for the last two lines shall be:

```
Electoral College vote total for John  McCain (R) is <VOTES> (<STATES> states)
Electoral College vote total for Barack Obama (D) is <VOTES> (<STATES> states)
```

Here `<VOTES>` and `<STATES>` are the electoral college vote totals for respective candidates and the number of states that voted for them.

State name generation. To make your program a bit simpler, you need to be able to generate actual state names on the fly. Since your program will output everything from within its main loop, you can no longer hardcode state names in your `printf()` statements without ugly hacks.

Instead, use the following solution. Declare an array `char *stateName[]` using the declaration below:

```
char *stateName[] = {"Alabama", "Alaska", "Arizona", "Arkansas", "California",
      "Colorado", "Connecticut", "Delaware", "District of Columbia", "Florida",
      "Georgia",  "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
```

```
    "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
    "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
    "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
    "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
    "Oklahoma", "Oregon", "Pennsilvania", "Rhode Island", "South Carolina",
    "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
    "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"};
```

(Note, this is your first encounter with `C` strings. We will talk about them later during the quarter.)

To output the state name use the following `printf` statements:

```
printf("%s goes to Obama\n", stateName[i]);
```

and

```
printf("%s goes to McCain\n", stateName[i]);
```

Here `int i` is the variable indcating which state name needs to be reported.

Program Name. Name your program `elections.c`.

Using arrays for image production

Until now, we produced PPM images "on the fly" - for each pixel enumerated row-wise we computed its color and immediately outputed it. This is convenient and fast for simple images, however, images with more objects on them can be created in a much more convenient way by preparing an array of pixel colors first, and then, by printing it out in one fell swoop. One of the benefits of this approach is that a color initially assigned to a specific pixel can be later overwritten, i.e., replaced with a new color. This makes it easier to draw multiple overlapping objects: you establish the order of depth and paint objects one after another starting with the deepest object on the image.

Given symbolic constants `HEIGHT` and `WIDTH` representing the dimensions of the PPM image (and a sybolic constant `COLORS` defined as `#define COLORS 3`, a PPM image can be represented by the following array:

```
unsigned char image[WIDTH][HEIGHT][COLORS];
```

Here `image[j][i]` refers to the color of pixel in row `i` and column `j` of the PPM image if we assume that the first pixel has coordinates $(0, 0)$. The `Red` component of the color is `image[j][i][0]`, the `Green` component is `image[j][i][1]` and the `Blue` component is `image[j][i][2]`.

To output a pixel, you can use the following printf statement:

```
printf("%c%c%c",image[j][i][0],image[j][i][1],image[j][i][2]);
```

Using arrays to represent PPM images, you will create a few programs described below. Each program will consist of two stages. On stage 1, the program will fill the `image` array with pixel colors according to what the image should look like. On stage 2, the completed array is scanned row-wise and the pixel colors are output in `printf()` statements forming a PPM file (this will, in fact, be done by a function you will write).

Rainbow Box image: rainbox.c

Write a program that outputs a 700×700 PPM image which constructs a *rainbow of boxes* starting with **violet** color on the outside and ending with **red** color on the inside. The full order of colors, the dimensions of the boxes and the RGB values to use in the program are specified in the table below. All boxes are concentric (it is your job to correctly identify where each square/box starts).

No.	Square size	Color	Red	Green	Blue
1.	100×100	Red	255	0	0
2.	200×200	Orange	255	128	0
3.	300×300	Yellow	255	255	0
4.	400×400	Green	0	255	0
5.	500×500	Light Blue	0	178	255
6.	600×600	Blue	0	0	255
7.	700×700	Violet	85	26	139

Your program **shall have no** *conditional statements* used to decide pixel colors. Instead, use an array to represent the image and color it consecutively with the seven colors as defined above.

Program name. Name your program `rainbox.c` (yes, I know, this is a horrible pun.)

Chessboard image: chessboard.c

Create an 800 × 800 image that represents on empty chess board.

A chessboard consists of eight rows and eight columns of **white** and **black** squares. Each square on your image then will have size 100 × 100 pixels The columns are numbered `a, b, c, d, e, f, g` from left to right. The rows are numbered `1,2,3,4,5,6,7,8` from *bottom* to *top* (thus, pixel (0,0) is in the `a8` square).

Square `a1` is **black**. **Black** and **white** squares alternate both on rows and columns.

Your program shall use no *conditional statements* for determining the color of each pixel. It shall consist of three functions: `int main()`, and two functions to color the chessboard and to output the image:

```
void drawChessboard(unsigned char image[][800][3]);
void printImage(unsigned char image[][800][3]);
```

`void drawChessboard()` function takes as input an image array and colors it *completely* as a chessboard.

`void printImage()` function takes as input an image array and ouputs the PPM file describing it to standard output.

The code you use in this program may come useful in the next assignment.

Program name. Name your program `chessboard.c`

Checkers image: checkers.c

This program extends your `chessboard.c` program[1]. It will read from input a list of chessboard squares, and output a chessboard with checkers positioned on the specified squares.

Input. The input to your program has the following format. First, your program shall read a single `int` number. This number will specify how many checker pieces are to be put on the board. After that, your program shall read pairs triples of values. The first and second values are `char` characters, the third value is an `int`. The number of triples shall be equal to the first input value of your program. The legal values for the first value are `'b'`,`'w'`, the legal values for the second value are `'a'`,`'b'`,`'c'`,`'d'`,`'e'`,`'f'`,`'g'`,`h'`. The legal values for the `int` values are 1—8. A sample input may look as follows:

```
4
bb2
wc3
ba3
wd6
```

The first value in the input indicates the number of checker pieces to be put on the chessboard. Each subsequent pair of values indicates the color and the position of a checker on the board. `a1` is the bottom left corner of the chessboard, `h8` is the top right corner. `a` through `h` are known as *files* or *verticals*. 1 through 8 are known as *ranks* or *horizontals*. A triple `bb2` means that a `black` checker is to be placed on `b2`. A triple `wc3` means that a `white` checker is to be placed on `c3`.

Your program is responsible for determining the validity of each square description. If a square description is invalid (e.g., `wa9` or `bF4` or `ww5`), this input shall be ignored, **but counted** as one of the specified number of pieces. Similarly, if the color designation is incorrect (e.g., `aa4`) the input line shall be ignored but counted. For example, the following input, which contains one incorrect location and one incorrect color designation is correct:

```
4
ba1
wb3
wq4
qf5
```

After reading `wq4` and `qf5` shall be read by your program but ignored. The total number of input lines is four, as specified, but only two checker pieces will be rendered.

Checker pieces. A `white` checker piece in this program shall be represented by a *filled* **white** circle of *radius 40* centered in the center point with *relative coordinates* (49,49) of the appropriate chessboard square.

[1]International checkers are played on a 10×10 checkerboard. However, an 8×8 board is used for a version of checkers known as Russian checkers. For simplicity, this program will use the 8×8 chessboard that you have generated using `chessboard.c` program.

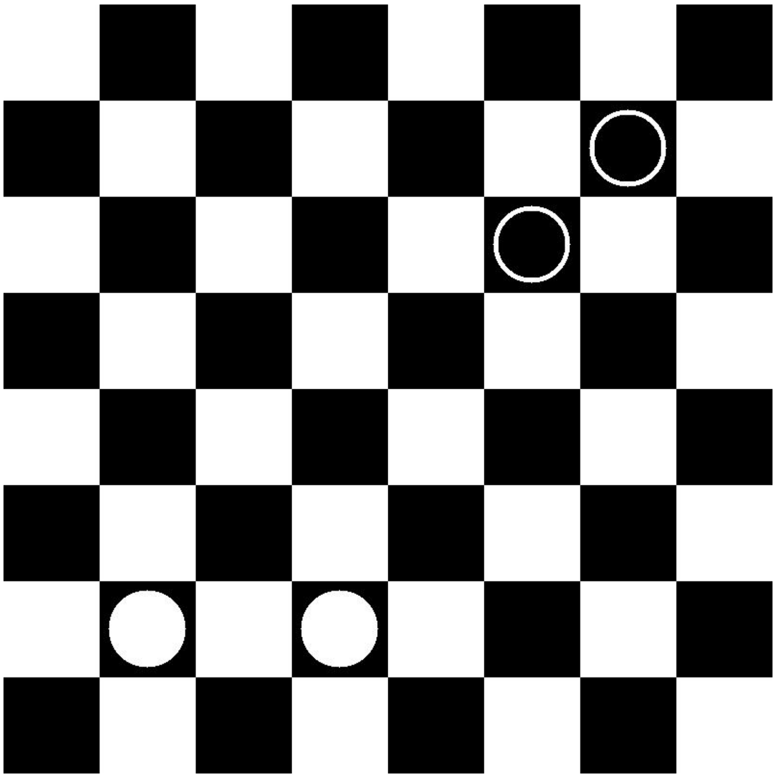

Figure 1: Checker Pieces.

Please note, that such pieces shall only be visible on **black squares** of the chessboard. This is by design (checkers are played only on black squares of the board).

A `black` checker piece in the program shall be represented by a *transparent white ring* with an inner radiur of 35 and outer radius of 40, centered at the *relative coordinates* (49,49) of the appropriate chessboard square.

On black squares, the checker piece will be visible as a small black "puck" with a white boundary. On white squares, black checker pieces would be visible. Since checker pieces cannot (by rules of the game) be present on white squares, your program shall check the color of the squares and **not attempt to place** black checker pieces on white squares.

Figure 1 shows how white and black checker pieces look on black squares.

Program flow. Start your program by setting the `image` array to contain the chessboard image as constructed by you in the `chessboard.c` program. As your image is of the same size, you can reuse the `drawChessboard()` function you wrote for that program.

Once the chessboard is prepared, start processing input. Read the total number of checker pieces. For each piece, read from input its color and its location and determine if both are valid. If the location and checker color are valid, check if the location is a white cell. If it is white, skip it, otherwise (the location is a black cell), draw the appropriate checker piece.

Once all checker pieces are drawn, output the image.

Functions. Use the following functions to help you with your program.

```
void getChecker(char *color, char * file, int rank);
int isBlack(char file, int rank);
void drawWhiteChecker(unsigned char image[][800][3], char file, int rank);
void drawBlackChecker(unsigned char image[][800][3], char file, int rank);
void setColor(unsigned char image[][800][3], int row, int col,
                         unsigned char R, unsigned char G, unsigned char B);
```

void getChecker() function takes as input references (pointers) to three variables. It shall read the description of one checker piece from standard input: its color (black or white), and its location (file and rank), set the values for the respective input parameters and return.

void setColor() function takes as input an array representing the image your program is drawing, a pair of pixel coordinates `row` and `col`, and an `RGB` triple of values. It colors the pixel at the intersection of the row `row` and the column `col` of the image array as designated by the `RGB` triple of color intensities.

int isBlack() takes as input a file and a rank of a chessboard cell and return 1 (true) if the cell is black, and 0 if it is white.

void drawWhiteChecker() takes as input an array representing the image your program is drawing and a pair of values (`file` and `rank`) representing the location of a checker piece to be drawn. The function then draws the `white` checker piece by overwriting the color values for the necessary pixels in the `image` array.

void drawBlackChecker() takes as input an array representing the image your program is drawing and a pair of values (`file` and `rank`) representing the location of a checker piece to be drawn. The function then draws the `white` checker piece by overwriting the color values for the necessary pixels in the `image` array.

Drawing a checker piece. You can draw a checker piece in the following way:

1. First, determine the top left corner of the chessboard square that should contain the piece. This requires some artihmetics, but notice, that your *file* designator (`'a'` through `'h'` can be turned into an integer with value of 1 through 8 using the following expression: `file - 'a' + 1`.

   The rest of the arithmetics is left to you.

2. While the actual checker piece is smaller than the entire chessboard square, for simplicity you can redraw the entire chessboard square. When redrawing it, you are essentially trying to create an image of a (**black** under normal circumstances) 100×100 square with a filled circle in the middle of it. This can be done in a manner similar to the national flag of Laos (your `Laos.c` program), except that you do not need to color the pixels outside of circle (since they already have been given a color).

3. To draw a `black` checker piece, you can either draw a white piece and then draw a smaller black circle on top of it, or you can check whether a pixel belongs to the *white ring*, and color it then, while leaving pixels that are in the circle *inside* the white ring untouched the same way as you do not recolor pixels outside of the ring.

Program name. Name your program `checkers.c`.

Submission.

Files to submit. Each pair submits one set of files from one account. The following files are mandatory:

```
team.txt,
elections.c
rainbox.c
chessboard.c,
checkers.c
```

`team.txt` file shall contain the name of the team and the names of all team members in each pair, and the Cal Poly IDs of each. E.g, if I were on the team with Dr. John Bellardo, my `team.txt` file would be

```
Go, Poly!
John Bellardo, bellardo
Alex Dekhtyar, dekhtyar
```

Submit `team.txt` as soon as you form your group.

Files can be submitted one-by-one, or all-at-once.

Testing

elections.c. The battery of tests for the elections program is the same as in Lab 7. File `election-tests.zip` contains all the tests. Instructor's executable is also provided.

rainbox.c. Instructor's `PPM` file of the *rainbox* image `rainbox-alex.ppm` is provided.

chessboard.c. Instructor's `PPM` file of the *chessboard* image `chessboard-alex.ppm` is provided.

checkers.c. A public test suite is made available. There are 10 files (`checkers-test01` through `checkers-test10`) available in the test suite. The entire suite is downloadable as a single zip file. Instructor's `PPM` images for each test case are made available. Instructor's executable `checkers-alex` is also provided. A test script to compile your program, run it on the test suite and output PPM images is included as well.

Submission procedure.

You will be using `handin` program to submit your work. Ssh to `unix1`, `unix2`, `unix3` or `unix4`.

Section 01:

```
> handin dekhtyar lab10 <files>
```

Appendix A: PPM Format Explanation

Portable Pixel Map (`.ppm`) file format is a simple format for storing graphical images. Files in this format can easily be created using C programs.

Basics. An computer image is a two-dimensional grid of pixels. Each pixel represents the smallest undivisible part of the computer screen. An image file is an assignment of color to each pixel. Pixels are referred to by their Cartesian coordinates. The top left corner of an image has the coordinates (0,0). The bottom right corner has the coordinates (n, m) where n is the width of the image in pixels and m is the height of the image in pixels.

Colors. Portable pixel map files use RGB (Red, Green, Blue) color format to represent the color of each pixel. In RGB format, a color of a single pixel is separated into three components: the red component, the green component and the blue component. The final color of an RGB pixel is determined by combining the Red, Green and Blue components into a single color.

In our course, all individual RGB component intensities range from 0 (not visible) to 255 (highest intensity) and are represented as integer numbers. RGB color (0,0,0) is black, RGB color (255,255,255) is white. The table below contains the list of colors used in this lab and their RGB values.

Color	RGB Red	RGB Green	RGB Blue
black	0	0	0
white	255	255	255
red	255	0	0
green	0	255	0
blue	0	0	255
yellow	255	255	0
purple	255	0	255
orange	255	128	0
dark blue	0	0	80

File format. There are two PPM formats: a "raw" PPM file and a "plain" (ASCII) PPM file. ASCII PPMs are human-readable, but they take too much space. Raw PPMs are smaller in size, but cannot be read by a human. In this lab you will be generating raw PPM files.

From http://netpbm.sourceforge.net/doc/ppm.html (with some modifications):

Each PPM image consists of the following:

1. *A "magic number" for identifying the file type. A ppm image's magic number is the two characters* `"P6"`.
2. *Whitespace (blanks, TABs, CRs, LFs).*
3. *A width, formatted as ASCII characters in decimal.*
4. *Whitespace.*
5. *A height, again in ASCII decimal.*
6. *Whitespace.*

7. *The maximum color value (Maxval), again in ASCII decimal. Must be less than 65536 and more than zero.*

8. *A single whitespace character (usually a newline).*

9. *A raster of* Height *rows, in order from top to bottom. Each row consists of* Width *pixels, in order from left to right. Each pixel is a triplet of green, blue and red intensities, in that order*[2].

Representing Colors in C. Outputing raw PPM files is actually quite simple. The idea is to use `unsigned char` variables to store information about RGB intensities.

Variables of type `char` and `unsigned char` are treated by C both as a character and as a number in the range -128 – 127 or 0 — 255 respectively. The following code outputs an RGB triple to stdout.

```
unsigned char Rcolor, Bcolor, Gcolor;
Rcolor = 255;
Bcolor = 0;
Gcolor = 128;

printf("%c%c%c", Gcolor, Bcolor, Rcolor);
```

[2] This is what worked for me. If your colors don't look right, switch to RGB order.

Appendix B: Electoral College Information

No.	State	Abbr.	Population	Electoral College Votes
1.	Alabama	AL	4,500,752	**9**
2.	Alaska	AK	648,818	**3**
3.	Arizona	AZ	5,580,811	**10**
4.	Arkansas	AR	2,725,714	**6**
5.	California	CA	35,484,453	**55**
6.	Colorado	CO	4,550,688	**9**
7.	Connecticut	CT	3,483,372	**7**
8.	Delaware	DE	817,491	**3**
9.	District of Columbia	DC	563,384	**3**
10.	Florida	FL	17,019,068	**27**
11.	Georgia	GA	8,684,715	**15**
12.	Hawaii	HI	1,257,608	**4**
13.	Idaho	ID	1,366,332	**4**
14.	Illinois	IL	12,653,544	**21**
15.	Indiana	IN	6,195,643	**11**
16.	Iowa	IA	2,944,062	**7**
17.	Kansas	KS	2,723,507	**6**
18.	Kentucky	KY	4,117,827	**8**
19.	Louisiana	LA	4,496,334	**9**
20.	Maine	ME	1,305,728	**4**
21.	Maryland	MD	5,508,909	**10**
22.	Massachusetts	MA	6,433,422	**12**
23.	Michigan	MI	10,079,985	**17**
24.	Minnesota	MN	5,059,375	**10**
25.	Mississippi	MS	2,881,281	**6**
26.	Missouri	MO	5,704,484	**11**
27.	Montana	MT	917,621	**3**
28.	Nebraska	NE	1,739,291	**5**
29.	Nevada	NV	2,241,154	**5**
30.	New Hampshire	NH	1,287,687	**4**
31.	New Jersey	NJ	8,638,396	**15**
32.	New Mexico	NM	1,874,614	**5**
33.	New York	NY	19,190,115	**31**
34.	North Carolina	NC	8,407,248	**15**
35.	North Dakota	ND	633,837	**3**
36.	Ohio	OH	11,435,798	**20**
37.	Oklahoma	OK	3,511,532	**7**
38.	Oregon	OR	3,559,596	**7**
39.	Pennsylvania	PA	12,365,455	**21**
40.	Rhode Island	RI	1,076,164	**4**
41.	South Carolina	SC	4,147,152	**8**
42.	South Dakota	SD	764,309	**3**
43.	Tennessee	TN	5,841,748	**11**
44.	Texas	TX	22,118,509	**34**
45.	Utah	UT	2,351,467	**5**
46.	Vermont	VT	619,107	**3**
47.	Virginia	VA	7,386,330	**13**
48.	Washington	WA	6,131,445	**11**
49.	West Virginia	WV	1,810,354	**5**
50.	Wisconsin	WI	5,472,299	**10**
51.	Wyoming	WY	501,242	**3**