Lab 6: Top-down design. Functions...

Due date: Friday, October 31, 11:59pm.

Lab Assignment

Assignment Preparation

Lab type. This is a **team programming lab**. For this lab, you need to form teams of four people (teams of three are OK too). Team members collaborate together on the assignment, but all other collaboration is not allowed.

Purpose. The lab allows you to practice the use of functions. You will also become acquainted with using header files to develop your C programs.

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.

Organizing in Teams

For this lab, you will organize in teams of 3-4 people. You are allowed to select your teammates at will. (Future assignments will involve more rigorous team formation, though).

Notice that the key difference between pair programming assignments and team programming assignments is that for team assignments, the overall work needs to be divided between all team members, and you can work on their individual assignments on their own, without other members of the team present. **However**, the assignment requires significant teamwork, both at the beginning, and, **especially**, at the end, when individual work is integrated into the final submission.

Start you work, by reading and discussing the assignment.

Formulate any questions to the instructor.

When you believe that you understand the assignment, discuss, which parts need to be done together, and which can be broken into individual tasks.

Assign individual tasks.

Discuss how the final submission will be constructed.

Use the lab period to commence the work on the parts of the program that need to be done jointly.

The Task

Game of Chess

The game of chess is played on an 8×8 square board, with alternating black and white squares. The game is played by two players, with two identical sets of chess pieces: white and black.

The full set of chess pieces includes:

8 pawns
2 rooks
2 knights
2 bishops
1 queen
1 king

The rows of the chessboard are called horizontals and are numbered (bottom-to-top) 1 through 8. The columns of the chessboard are called verticals and are referred to (left-to-right) 'a' – 'h'. Thus, the bottom left corner of the chessboard is referred to as a1, the top right corner — h8 and the square in the third row and third column is c3. (see Figure 1).

In future assignmnets, the rules of the game may become relevant, and they will be introduced as needed. For this lab, it is sufficient to simply know the

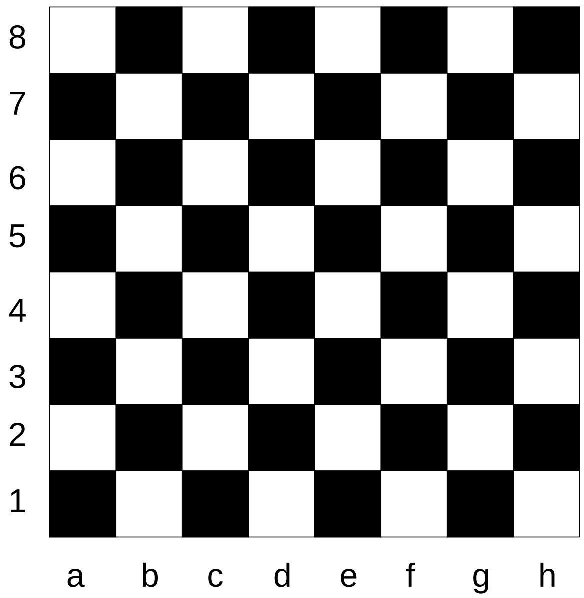


Figure 1: Chessboard.

information above.

Task Overview

Each team will build a library of functions that allows for display of chess pieces on the chessboard. In particular, each team has to prepare for submission, two components:

1. `chessboard.h`: a collection of C functions for drawing the chess board and chess pieces on it.

2. `chessboard.c`: a C program that reads from standard input description of a chess piece and its location, and using the functions described in `chessboard.h` outputs to `stdout` (standard output) a PPM file containing the image of a chessboard with the specified chess piece in the specified location.

Specifications

Chess board image. Your program/function library will work with PPM files of the size 800×800. Each square of the chessboard is represented by a 100×100 area of the image. Square `a1` (bottom left) is black.

Chess Pieces. Essentially, chess pieces are 100×100 '"icons". Chess pieces come in six varieties (pawn, rook, knight, bishop, queen, king) and two colors (white, black).

Graphical Elements. Each chess piece is constructed out of simple graphical elements. In this lab, there are only two types of graphical elements that are used to construct chess pieces: circle and rectangle.

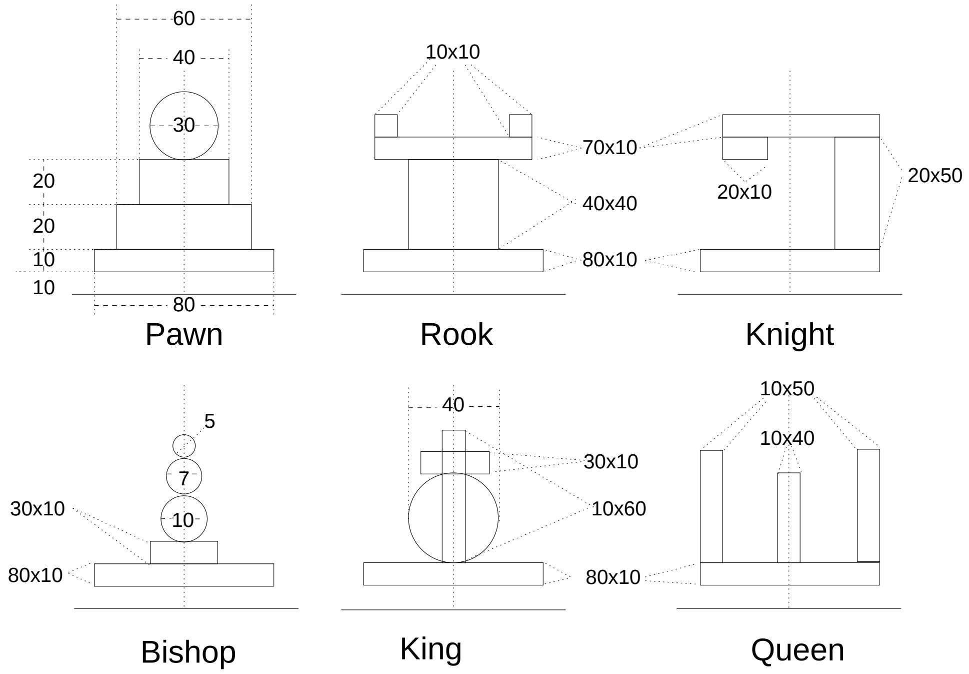
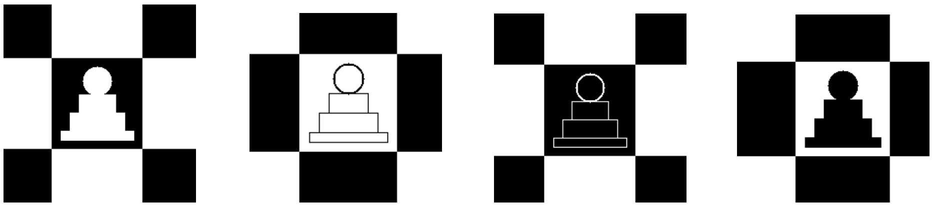


Figure 2: How to draw chess pieces on the chess board.



(a) white-on-black (b) white-on-white (c) black-on-black (d) black-on-white

Figure 3: Four ways of drawing a pawn.

Figure 2 shows how each chess piece is constructed out of circles and rectangles. Each image is a $100\times$ 100 icon. The bottom solid line on each image is the bottom of the chess square. Each chess piece is elevated 10 pixels above the bottom of the square, and sits on an 80×10 base rectangle. The dimensions of all rectangles, and the diameters of all circles are included in the Figure.

Coloring chess pieces. Each chess piece can come in one of two colors and may need to be drawn on either a black or a white square. Thus, there are four possible drawings you have to be able to make for each chess piece:

Color of chess piece	Color of square	Drawing style
white	black	solid white shape (shape boundaries are white)
white	white	black outline of each shape boundary
black	black	white outline of each shape boundary
black	white	solide black shape (shape boundaries are black)

Figure 3 shows an example of a pawn drawn in all four styles. See Appendix A for the images of all chess pieces as rendered by the instructor's program.

Atomic graphical elements. All chess pieces are constructed out of circles and rectangles.

For circles, the information provided throughout the program will be coordinates of the center (x,y) and radius (one half of the diameter from Figure 2).

For rectangles, the information provided throughout the program will be coordinates of the top left corner (x,y), width and height of the rectangle.

Input specification. Your `chessboard.c` program shall start its execution by reading the following information:

Input No.	Type	Name	Meaning
1.	`char`	piece	specifies the type of chess piece to draw
2.	int	color	specifies the color of the chess piece
3.	char	horizontal	the horizontal of the square where the piece is to be drawn
4.	int	vertical	the vertical of the square where the piece is to be drawn

The parameters may take the following values.

piece:

value	meaning
`'p'`	pawn
`'r'`	rook
`'b'`	bishop
`'n'`	knight
`'q'`	queen
`'k'`	king

color:

value	meaning
0	black
1	white

horizontal: `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'g'`, `'h'`.

vertical: 1,2,3,4,5,6,7,8.

For example, `r 1 a 8` means *place white rook on a8*; `k 0 f 3` means *place black king on f3* and `n 1 c 4` means *place white knight on c4*.

Invalid inputs. Your program shall check the validity of all inputs. If any input has a value different from the ones described above, your program shall print an error message and **terminate**.

Example. The following inputs are invalid:

```
a 1 h 1
r 2 h 1
n 0 w 1
n 0 h 10
n 0 h 0
```

Design

This section concentrates on the internal design of the software for your assignment.

The heart of your assignment is the `chessboard.h` header file. This file **shall contain all functions and constants** you define for this assignment. Before describing the overall structure of the library of functions you need to implement in `chessboard.h`, let us first discuss the underlying problem.

Decomposing the chess piece drawing problem

Recall, that so far we create PPM files in the following manner:

```
For each pixel of the PPM image (row-by-row)
   - determine its color
   - print the color to standard output
```

Because the number of pixels in PPM images is rather large, and we, thus far, can only use variables that store single (atomic) values, we cannot keep the entire color "matrix" of the PPM image in the memory while executing a program. On-the-fly computation of color for current pixel allows us to output PPM while using very few variables in the program (and thus - occupying very little memory space).

This, however, **comes at a disadvantage**. We **must** determine the color of each pixel in turn, and immediately output it into the PPM stream. This means, that once the color is printed to stdout, we *can no longer change it.*

With this in mind, let us analyze problem at hand.

Problem analysis. The assignment requires you to write a program that accepts four inputs from standard input, and based on them, outputs to standard output a PPM file which contains an image of a chessboard with a specified chess piece at the specified chessboard square.

1. **Check each pixel.** Because we must output a PPM file, we have to reduce our problem to the problem of determining the color of each pixel, for an image of a chess board with a single chess piece located in one of its squares.
2. **Two top-level pieces of functionality.** We break the problem above into two parts:
   (a) For each pixel, determine its color on an empty chessboard.
   (b) For each pixel determine whether it is inside the chess piece, and find its color.

   Note, that if we could perform both of these tasks, we could discover the color of each pixel using the following straightforward procedure:

   ```
   Determine if pixel is inside the chess piece.
   If yes, output the color of the chess piece (or boundary color)
   If no, determine which chess board square the pixel is in,
          and output the color of that square.
   ```

3. **Subtask 1. Square color.** We can find the color of the square to which a pixel belongs in two steps:

   (a) Step 1: determine which square of the chess board, the pixel is in.

   (b) Step 2: determine the color of this square.

   To accomplish the first task, recall that each chess board square is represented by one 100×100 square in the PPM file.

   To accomplish the second task, remember how we emulated a chess board in the example in class…

4. **Subtask 2. Chess Piece recognition.** We note that inside each chess board square, chess pieces will always be positioned in the same way: 10 pixels above the bottom of the square, and 10 pixels to the right of the left border of the square.

   Thus, we can conduct the test of whether the pixel belongs to the chess piece shape in the following manner:

   (a) Determine the chess board square the pixel is in.

   (b) Check if the input chess piece is located in the same chess board square.

   (c) If not, then the pixel is not part the chess piece shape.

   (d) If yes, check the pixel against the actual shape of the chess piece **within a 100×100 square**.

   (e) If the pixel is inside the shape, determine the color of the pixel based on whether or not, the pixel is on the boundary of any atomic shape of the chess piece, and whether boundaries need to be made visible in the output (black-on-black, white-on-white).

   (f) If the pixel is outside of the shape, its color is the color of the chess board square, the pixel is in

   The key here is the switch of pixel coordinates from the full chessboard to the coordinates within a single chessboard square achieved in (d).

5. **Recognition of chess piece shapes.** Each chess piece shape consists of a number of rectangles and circles. Essentially, given a pair of pixel coordinates within the 100×100 image (as computed by subtask 4.2), we can determine if the pixel is in the chess piece shape, using the following procedure:

```
For each graphical element (circle, rectangle)
                which is part of the input chess piece
    find if the pixel is inside currently considered graphical element.
If the pixel is inside at least one graphical element,
    then the pixel is inside the chess piece shape.
If the pixel is NOT inside any graphical elements,
    then the pixel is NOT inside any graphical element.
```

   Note, that due to different modes of drawing the same chess piece shape, some drawings (black-on-black, white-on-white) will use contrasting color for individual shape boundaries. Thus, in the process above, we have to discover whether a point is (a) inside the shape, (b) found on at least one graphical element shape boundary or (c) outside the shape.

6. **Recognition of Rectangles and Circles.** At the bottom of our solution lies the ability of the program to do the following tasks:

   (a) Given a rectangle (represented by its top left corner, width and height), and a point in a 100× 100 2D space, determine if the point inside the rectangle.

   (b) Given a circle (represented by its center and radius), and a point in a 100× 100 2D space, determine if the point inside the circle.

We have just successfully decomposed the problem of writing a program that draws a chessboard with a single chess piece on it, to the problems of

1. determining if a point is inside a rectangle;
2. determining if a point is inside a circle;
3. determining the color of each chess board square;
4. determining which chess board square a given pixel belongs to, and what is its relative position inside the square.

Towards the solution: a hierearchy of functions

We have completed the top-down design of the solution. Our implementation will be bottom-up. We start by specifying functions for implementing the most simple functionality found in our design. Once these functions are described, we proceed to *higher-level* functions, which use the simpler functions.

For this assignment, you shall implement the following layers of functions:

Layer 1. Recognition of atomic graphical shapes, determination of chess board square colors, conversion of pixels into chessboard squares.

Layer 2. Recogniztion of shapes of chess pieces.

Layer 3. "Drawing" (coloring) individual pixels on the chess board.

Layer 4. Coloring of the entire chess board.

Layer 1 functions. The first functions you shall implement are the six functions that provide the basic support for the assignment.

1. `int getHorizontal(int i).`

   This function takes as input the y coordinate of a pixel, and returns the horizontal of the chess board the pixel belongs to. The return values are 1–8, where 1 shall correspond to the bottom horizontal (high values of `i`), and 8 shall correspond to the top horizontal (low values of `i`)

2. `int getVertical(int j).`

   This function takes as input the x coordinate of a pixel, and returns the vertical of the chess board the pixel belongs to. The return values are 1–8, where 1 shall correspond to the a (leftmost) vertical and 8 shall correspond to the h (rightmost) vertical.

3. `int whiteSquare(int horizontal, int vertical).`

4. `int blackSquare(int horizontal, int vertical).`

   These two functions take as input a pair of integers representing the square of the chess board (the values of `vertical` and `horizontal` shall be between 1 and 8), and determine what color the square is.

   `whiteSquare()` returns 1 if `(horizontal, vertical)` is a white square, and 0 if it is a black square.

   `blackSquare()` returns 1 if `(horizontal, vertical)` is a black square, and 0 if it is a white square.

   Note: only one such function is actually needed for the program, but for the sake of symmetry, you shall implement both.

5. `int checkCircle(int i, int j, int centerY, int centerX, int radius).`

   This function checks if a pixel is inside a circle, on its boundary, or outside the circle. The arguments are:

Argument name(s)	Meaning
`i, j`	coordinates of the pixel
centerY, centerX	coordinates of the center of the circle
radius	radius of the circle

   `checkCircle()` shall determine the location of the pixel `(i,i)` with respect to the circle with radius `radius` centered in `(centerY, centerX)`. It shall return one of the following:

Function output	Meaning
-1	`(i,j)` is **outside the circle boundary**.
0	`(i,j)` is **on the circle boundary**.
1	`(i,j)` is **inside the circle**.

6. `int checkRectangle(int i, int j, int topLeftY, int topLeftX, int height, int width).`

   The `checkRectangle()` function checks the location of the input pixel with respect to the specified rectangle. The arguments of the function are:

Argument name(s)	Meaning
`i, j`	coordinates of the pixel
topLeftY, topLeftX	coordinates of the top left corner of the rectangle
height, width	height and width of the rectangle

   `checkRectangle()` shall produce the same outputs as `checkCircle()`:

Function output	Meaning
-1	`(i,j)` is **outside the rectangle boundary**.
0	`(i,j)` is **on the rectangle boundary**.
1	`(i,j)` is **inside the rectangle**.

Layer 2 functions. This layer contains six functions that recognize the shape of each individual chess piece. The functions are:

1. `int checkPawnShape(int i, int j).`
2. `int checkRookShape(int i, int j).`
3. `int checkKnightShape(int i, int j).`
4. `int checkBishopShape(int i, int j).`
5. `int checkQueenShape(int i, int j).`
6. `int checkKingShape(int i, int j).`

All these functions behave in a similar manner. The funtions take as arguments the pixel coordinates. The output of the function is:

Function output	Meaning
-1	`(i,j)` is **outside the boundary of the chess piece shape**.
0	`(i,j)` is **on some shape boundary line of the chess piece**.
1	`(i,j)` is **inside the chess piece** but not on any shape boundary lines.

Note: All chess piece shapes are 100×100 "icons". As such, the inputs to these functions (both `i` and `j`) have values ranging from 0 to 99 (or 1 to 100, if you want), which specify the *relative position of a pixel within its chessboard square.*

The shapes of each chess piece have been described above. See Figure 2 for the specific description of the graphical components of each chess piece shape.

Layer 3 functions. Functions in Layer 2 are designed to recognize whether a pixel belongs to a chess piece shape or not. In order to create the PPM file, however, we need to go one step further, and determine the color of each pixel. Pixel colors are determined based on two things:

- location of the pixel within the chessboard square: inside the chess piece shape, outside it, on a boundary line.
- colors of the chess piece and the chessboard square.

The functions in Layer 3 will determine the color of a pixel. For this assignment we adopt a straightforward approach: Layer 3 shall consist of six functions: one function for each chess piece type. The functions are:

1. `int pawn(int horizontal, int vertical, int color, int i, int j).`
2. `int rook(int horizontal, int vertical, int color, int i, int j).`
3. `int knight(int horizontal, int vertical, int color, int i, int j).`
4. `int bishop(int horizontal, int vertical, int color, int i, int j).`
5. `int king(int horizontal, int vertical, int color, int i, int j).`
6. `int queen(int horizontal, int vertical, int color, int i, int j).`

Each function works in a similar manner to "draw"[1] the chessboard with the designated chess piece (pawn, rook, knight, bishop, king, queen). Each function takes the following arguments:

Argument name(s)	Meaning
`horizontal, vertical`	the chess square in which the chess piece is to be drawn
`color`	color of the chess piece (0=black, 1=white)
`i, j`	coordinates of the pixel (on the 800×800 image)

The output of each function is a color:

0 if the pixel `(i,j)` needs to be painted **black**.

1 if the pixel `(i,j)` needs to be painted **white**.

For example, `rook(3,4,0,450,350)` asks *"What is the color of the pixel (450,350) on the chessboard containing a black rook on square* `d3`*?"*.

[1]The functions return the color of a pixel and do not have side effects, so, technically, they do not "draw" anything. However, we choose to call them "drawing" functions, because this is what the result of repetative use of the functions (for each pixel of an image) is.

Layer 4 functions. Functions in Layer 3 work on pixel-by-pixel basis, providing the color of individual pixels. Layer 4 functions actually output the entire PPM file. These are **the only** functions in your assignment that are allowed to have **side effects**. In fact, functions in this layer are `void`, i.e., they do not return any values.

There are two functions in this layer.

1. `void printPPMHeader(int width, int height, int intensity).`

   This function takes as input the width and the height of the PPM file your program is creating and the intensity number for the colors (which would typically be 255), and prints out the three-line PPM file header.

   For example `printPPMHeader(100,200,255)` shall result in the following text printed to standard output:

   ```
   P6
   100 200
   255
   ```

2. `void drawChessboard(char piece, int color, int horizontal, char vertical)`

   This function takes as input the following **validated** arguments:

Argument name(s)	Meaning
`piece`	the chess piece to be drawn on the chessboard
`color`	color of the chess piece (0=black, 1=white)
`horizontal, vertical`	the chess board square containing the chess piece

   The **validated input** requirement means that all values passed to `drawChessboard()` must be in the valid ranges specified on page 5. All input validity checks have to be conducted in the `main()` function of the `chessboard.c` program, not in the `chessboard.h` library functions.

   Note, that both `piece` and `vertical` are `char` variables. In fact the arguments to this function match the input values that the `main()` function of `chessboard.c` must read.

chessboard.c. Your `chessboard.c` file shall consist of a single `main()` function that does the following:

- reads the four inputs (piece, color, vertical, horizontal) from standard input.
- checks each input for validity, quits if any of the inputs are invalid.
- if all inputs are valid, calls the Layer 4 functions from `chessboard.h` file to output the proper PPM image.

Unit Testing

This lab assignment introduces a new concept related to testing of your software. Until now, your software consisted of a single `main()` function, containing all the program's functionality.

Now, **most** of the software functionality is hidden in four layers of the functions in `chessboard.h` function library. Functions from upper layers **depend**

on lower-layer functions. That is: if there is an error in implementation of a lower-layer function, the upper-layer functions will not work correctly.

Therefore, not only **testing** is important, but it must be done **differently**.

Unit testing is the process of testing your software that concentrates on validating individual components (units) of the software. The idea is to make certain that before an upper-layer function is tested, all lower-layer functions called from it **have been successfully validated.**

In this assignment units are individual functions. To unit-test functions, you can write simple C programs that check the work of those functions and those functions only, and use them to test the implementations of your functions.

Example. Consider, for example, the task of unit-testing the `checkCircle()` function from `chessboard.h`.

The advantage of putting functions in a `.h` library is that **multiple C programs can use them**. To test `checkCircle()` I can create a C program `circleTest.c`, that looks as follows:

```
/* circleTest.c    CPE 101, Lab 6, Alex Dekhtyar            */
/* tests checkCircle() function of the chessboard.h library */

#include <stdio.h>
#include "chessboard.h"

int main() {

 printf("%d\n", checkCircle(1,1, 50, 50, 10);   /* test 1: expect -1 */
 printf("%d\n", checkCircle(50,50, 50, 50, 10); /* test 2: expect 1 */
 printf("%d\n", checkCircle(40,50, 50, 50, 10); /* test 3: expect 0 */
 printf("%d\n", checkCircle(5,5, 10,10, 7);     /* test 4: expect 1 */
 printf("%d\n", checkCircle(80,57, 40, 60, 1);  /* test 5: expect -1 */

return 0;
}
```

Compiling and running this program will yield an output consisting of five numbers. If the output is

```
-1
1
0
1
-1
```

then `checkCircle()` has successfully passed the tests in the `circleTest.c`. At this point, I can either add more test cases to `circleTest.c` or declare the unit test of `checkCircle()` to be successfully completed.

If the output is different — `checkCircle()` contains errors in the code, which have to be debugged and fixed. Observing which test case fails will allow me to determine the nature of the error and fix it.

Instructions. You are **required** to unit-test each function you create for this program. In a team environment, unit-testing can happen in one of the following ways:

1. Everyone. Each student unit-tests his/her functions. Advantages: easy to coordinate; everyone acquires experience with unit-testing. Disadvantages: possible bias (each student owns the code they test), lack of uniformity (some students may perform more rigorous testing).

2. Designated Tester. The group selects one person, whose main work is to test the software. Each function, when completed is submitted to the tester, who, in turn designs and runs the tests, and reports any noticed failing tests to the programmer of the function. Advantages: uniformity in testing standards; similarity to industry practices. Disadvantages: requires higher level of coordination, gives only one student testing experience.

3. Circular Firing Squad. Each student participates both in writing functions and in testing them, however, students do not test their code. Rather, students form a circle, and pass their code along the circle for unit-testing. Advantages: all students participate, testing standards may converge to uniformity over time. Disadvantages: requires higher level of coordination and, potentially, evolving testing standards.

Each group can choose the approach to unit-testing that suits it best.

To verify that each group engaged in rigorous unit-testing, you are required to **submit** all .c programs that you have created for the purposes of unit-testing. These programs will not be compiled, run and graded, but their presence will be verified to establish, that, indeed, each function had been tested.

The only exception from unit-testing may be Layer 4 functions: you are allowed to use `chessboard.c` for testing purposes.

Submission.

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

`team.txt`, `chessboard.c`, `chessboard.h`

You also must submit all your unit test C programs. Feel free to use any filenames for them.

`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.

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

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

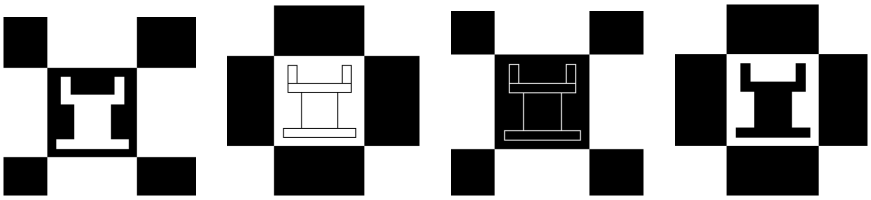


Figure 4: Four ways of drawing a rook.

- Students from **Section 009** shall execute the following submission command:

  ```
  > handin dekhtyar-grader lab06-09  <your files go here>
  ```

- Students from **Section 011** shall execute the command:

  ```
  > handin dekhtyar-grader lab06-11 <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.

Programs that do compile will be tested as follows:

- 20% of the grade: Compliance. Your submission must be organized as described, with function declarations and definitions matching the ones found in this document.
- 10% of the grade: Style. Your program must follow the course style. In particular, ALL CONSTANTS must be `#define`d in the `chessboard.h` file.
- 10%: Unit testing. Your unit test programs will be checked to verify that you have conducted unit testing.
- 60% of the grade.: Testing. 24 tests will ensure that your program renders each chess piece correctly in each color combination. Extra tests will be used to determine that your program validates inputs. These tests are public and the test data is released to you (see the course web page). A small set of private tests will allso be used to check for special cases.

Appendix A. Chess piece renderings

Figure 3 shows the rendering of the pawn. The renderings of the other five types of chess pieces are shown in Figures 4 (rooks), 5 (bishops), 6 (knights), 7 (queens) and 8 (kings).

Appendix B: 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.

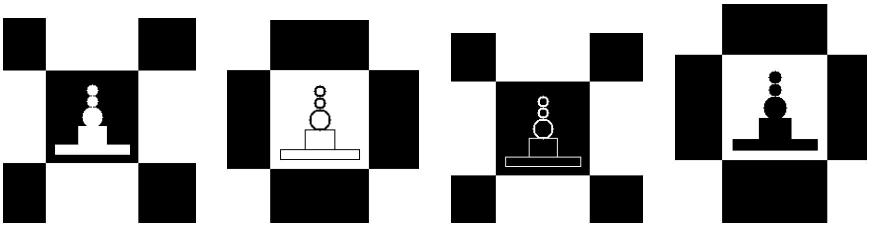


Figure 5: Four ways of drawing a bishop.

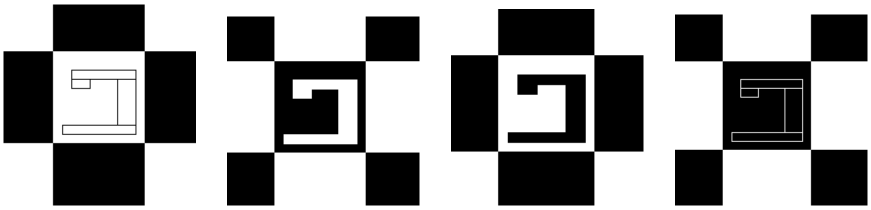


Figure 6: Four ways of drawing a knight.

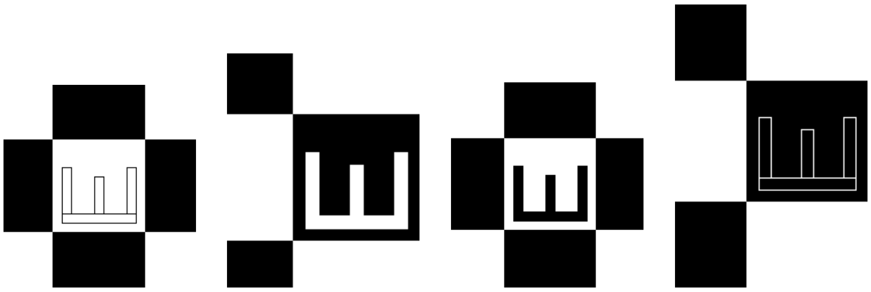


Figure 7: Four ways of drawing a queen.

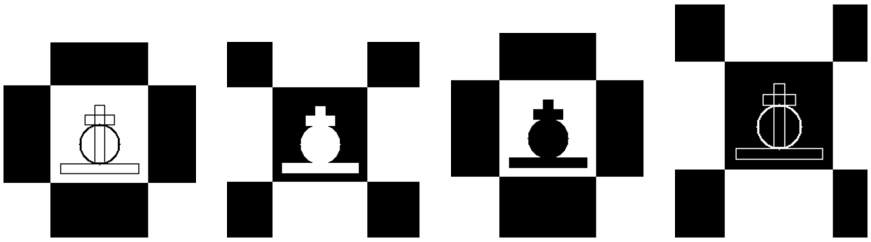


Figure 8: Four ways of drawing a king.

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.