8.2 Practice Programs

Sample solutions to these problems will be included by the beginning of 2013 at the following website:

http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3223/book/

Note: Most programs with two dimensional arrays tend to involve more complexity than many of the previous examples in this book. Thus, solutions to most of these practice problems are significantly longer than practice programs in previous chapters and are likely to take significantly more time than previous practice problems may have taken. All of the solutions to these programs can be improved once functions (Chapters 9, 10) are incorporated.

1) Write a program that allows two users to play tic-tac-toe against each other. Ask the user to enter in their move with integers in between 0 and 2, inclusive, specifying the row and column of the square to place a piece. Assume that both users enter appropriate input. Your program should identify as soon as one of the players wins the game, print the appropriate message and end. If no user wins after the board is full, your program should print out that the outcome was a tie and end.

2) Add error checking to your program from question #1 so that if a user chooses an invalid square, your program states this and asks the user to re-enter their move. (Your program should continue to do this until the user enters a valid move.)

3) Write a program that creates a 5 x 5 game board and fills it with random integers in between 0 and 4. When displayed, the user will only see underscores. The user starts at the top left corner of the board (row 0, column 0) and their goal is to get to the bottom right corner of the board(row 4, column 4). At each step, ask the user if they want to go up, down, left or right. Assume the user enters a valid move. On a typical move, the number on the square to which the user moves is the amount added to her score. If the user moves to a square storing a 4, the game ends and the user loses (with a score of 0). If the user moves to a square storing a 0, the game ends and the user's score is equal to however many points the user had acquired up until that point. Alternatively, if the user makes it all the way to (4,4) without the game ending, the user's score is however many points she acquired on the full journey, including the points earned for the last square. (Note: If this last square stores a 4, the user gets these points added to their score.)

4) Add error checking for the program from #3 so that if the user enters a direction that moves him off the board or to a square he's previously been to, the move is not allowed and the user is asked to re-enter. As an added bonus, when prompting the user to move, only present them with a list of valid moves out of the four possible moves.

5) The peg game involves starting with a triangular square with five rows of holes (1 hole on row one, 2 holes on row 2, etc.) with 14 of the 15 holes filled with pegs. You can jump one peg over another so long as it allows you to land in the empty square. The goal of the game is to finish one one peg left. The empty peg at the beginning of the game is the middle one, on row three, Here is how the initial game board looks:

```
    P
   P P
  P x P
 P P P P
P P P P P
```

In order to move, ask the user to enter a row and column of the peg to jump and the row and column of the end location of the jump. (Note: The rows should be numbered 1 to 5, and the valid columns on row R are 1, 2, ..., R.) Assume that the entered values are valid. You should remove the peg in between the start and end locations of the jump and move the original peg from the start to end location. For example, at the beginning of hte game, if we moved from row 5 column 4 to row 3 column 2, the resulting board would be:

```
    P
   P P
  P P P
 P P x P
P P P x P
```

6) Add error checking for the program from #5 so that if the user enters invalid information for a jump, they are reprompted to enter again. As an extra enhancement, have the user select which peg they want to jump, and then provide for them

7) Write a program that reads in two 3 x 3 matrices from a file and adds the matrices and prints the result out to the screen.

8) Write a program that reads in two 3 x 3 matrices from a file and multiplies the matrices and prints the result out to the screen.

9) Write a program that randomly generates 5 distinct spots on a 5 x 5 grid, fills those spots with the '*' character, fills the rest with the space character and prints out the grid to the screen. For clarity, print a space between squares on the same row.

10) Edit the program for #9 so that instead of putting space characters in all the spots without a bomb, marked by the asterick character, a single character reflecting the number of adjacent bombs to that square, as is done in the game Minesweeper. For example, one possible board that could be printed is as follows:

```
* 3 * 2 1
2 * 3 * 1
1 2 3 2 1
0 1 * 1 0
0 1 1 1 0
```