COP 3530 – Data Structures and Algorithms I

Project 2

Due: Friday, March 2nd 11:59 P.M. CST

Objective:

This project is meant to help you illustrate concepts of multidimensional arrays and algorithms. In this project, you are asked to create the game of Battleship. In this project, you will have to be able to communicate with another program through intermediate files. Again, you will be asked to create a makefile.

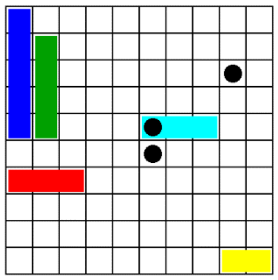
If the project fails to compile, the maximum grade that can be earned is a 15! These issues are caused by either incorrect syntax or not running on the SSH. Make sure you take care of both!

Problem Description:

Battleship is a classic Milton-Bradley game, where two players, in secret, position their navy on a grid. Each ship in the navy covers a contiguous horizontal or vertical section of grid cells. The type of ship determines the number of grid cells covered. The players, in turn, lob salvos at each other's navy by calling out grid locations. Should a salvo coincide with a grid cell covered by a ship location, the attacked player calls out "hit". If the salvo does not coincide with a cell covered by a ship, the attacked player calls out "miss". If all the cells covered by a single ship are hit, the attacked player calls out "sunk".

The first player to sink all of the opponent's ship wins.

Here is an example of one player's grid after three salvos by the opponent:



The salvo to grid location (2, 8) was a miss, the one to location (4, 5) was a hit, and the one to (5,5) was another miss.

The grid also illustrates a typical navy: a carrier in blue, a battleship in green, a destroyer in red, a submarine in aqua, and a patrol torpedo boat in yellow.

Your task is to build a battleship-playing program that will outsmart your classmates' programs.

May the best program win!

Program to Program Communication

The two competing programs will be designated player A or player B. When each program starts up, it will position its navy. Player A will start the competition by appending a grid location to the file A.salvos. Player B will then read the A.salvos file and append the response, either MISS, HIT, SUNK, or DEFEATED, to the file B.responses. Player B will then append a grid location to the file B.salvos. Player A will read B.salvos and append a response to A.responses. Player A then reads B.responses to guide its next salvo. This continues until a program has its last ship sunk, at which time, the program should print the message DEFEATED and quit.

File Formats

The ships position file will consist of a number of entries. Each entry is composed of a ship designator followed by row and column indices. The legal ship designators are the following letters:

Designator	Ship type	Number of cells
C	Carrier	5
B	Battleship	4
D	Destroyer	3
S	Submarine	3
P	Patrol Boat	2

The grid is made up of 10 rows and 10 columns. The top row of the grid is row 0 and the leftmost column of the grid is column 0.

Each entry in the salvos file is composed of row and column indices. For the prior example, the salvos file would look like:

```
2 8
4 5
5 5
```

The corresponding responses file would look like:

```
MISS
HIT
MISS
```

With respect to the responses file, if a hit results in a sunk ship, only the response SUNK is appended to the file. If a hit results in the **last** remaining ship being sunk, only the response DEFEATED is appended to the file.

NOTICE: all files are free format. It would be perfectly OK to try to cause your opponents program to fail by adding extraneous whitespace (within reason) to an addition to the salvos or responses file. For example, it would be legitimate to append the salvo (6, 8) to the salvos file as:

```
2 8
4 5
5 5

            6
  8
```

All that is required is that the file is appended and the row comes before the column.

Remember to close all files after appending to them!

Handling Exchanges

Your program should pause at the very beginning if it is player B.

Your program should also pause after each addition to the salvos files, waiting until you tell it to continue (by pressing any key for instance). This will allow the other program time to read the salvos file and append a response and a new salvo. At each pause, your program should print out the status of your ships and the status of your salvos. For example, the status of your ships (on the left) and salvos (on the right) might look like:

```
ships                                  salvos

 -  -  -  -  -  -  -  x  P  -           -  x  o  -  -  -  -  -  -  -
 -  C  -  -  -  -  -  -  -  -           -  x  o  -  -  -  -  -  -  -
 -  C  -  -  -  -  -  -  -  -           -  o  o  -  -  -  -  -  -  -
 -  C  -  -  x  x  B  B  -  -           -  -  -  -  -  -  -  -  -  -
 -  C  -  -  -  -  -  -  -  -           -  -  -  -  x  -  -  *  -  -
 -  C  -  -  x  -  -  -  -  -           -  -  -  -  -  -  -  x  -  -
 -  -  -  -  D  -  -  -  -  -           -  -  -  -  -  -  -  O  -  -
 -  -  -  -  D  -  -  -  -  -           -  -  x  -  -  -  -  -  -  -
 -  -  -  -  S  S  S  -  -  -           -  -  -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -  -  -           -  -  -  -  -  -  -  -  -  -

Round: 22
```

Here, the *BCDPS* letters stand for the ships and the *x*'s and *o*'s refer to hits and misses, respectively. The * indicates the latest salvo. After the boards are printed, the number of the round should be printed. Player A's rounds are 1, 3, 5, 7, ... while Player B's rounds are 2, 4, 6, 8, ... The round numbers are useful for keeping track of whose turn it is. *Remember to close the files after each turn.*

Placing Ships on the Board

Ships will be placed on the boards at random positions. In order to do this, you will need to randomly select a starting location (random your row and column) and an orientation (let 0-3 represent top, down, left, and right). Then attempt to place the ship on to the board. There are two things you will need to take care of. First, you will need to be careful about whether a ship has already been placed on one of the spots. In which case, you may not place a ship on top of another ship. Secondly, you need to be sure that the ship does not extend off the board. It may be beneficial to you to create two different functions. One function places the ship on the board, while the other function checks whether it is legal to place a ship in a specific location (would need to be passed the starting location, orientation, and length of the ship). The first function would then call the second function before placing the ship.

Running Your Program

Your program should be started with one command line argument that indicates the player for your program. The argument is the letter A or the letter B, designating player A or player B. Here is an example of how the program should be ran:

```
$./battleship.x A
```

Additional Guidance

You should create a struct that can capture the information about the state of the game. This struct should contain at minimum, the two boards, with updated information and the amount of hits on each ship (so that you can accurately report when a ship is sunk). Note that you can have additional information if you find that you need it.

Develop the program incrementally. Break it down to the various tasks and use functional stubs when needed. This can help you ensure that you can submit a compiling program.

Try not to repeat code when it is not needed. Instead, identify tasks that are appropriate to break out into new functions and use those instead. I am allowing you freedom to design this project how as you see fit, but try to keep in mind the concepts we have talked about in class.

Extra Credit Opportunities

(10 points) Write a recursive function that handles making the next guess based off of the previous non-sinking hit. This function will recur as long as there is a hit that continues from a previous hit. You can do this by having a function that will recur on top, bottom, left, and right positions from a previous hit and does not return unless there is a miss or a ship is sunk.

(10 points) When starting a game, allow the player to decide whether they wish to place the ships randomly or load from a file of ship layouts. The file structure should store each ship layout as ship layouts were displayed in “Handling Exchanges”. Have your file contain 5 possible configurations. When a player decides that they want to load from file, prompt them for a number 0-4. Then, in your program, load that configuration instead of asking for ship positions. This will only be given extra points if you jump to the chosen ship layout using random file access and the

fseek function (no extra points will be given if you have to read from the beginning of the file). More on fseek can be found by reading the online documentation or reading from tutorialpoint here: https://www.tutorialspoint.com/c_standard_library/c_function_fseek.htm.

(5 points) Your program beats a randomly selected opponent from the class. This is to encourage more thought into your guessing algorithm. A tournament would be nice, but there is limited classroom time to do so.

Breakdown of Graded Items:

The following is a breakdown of what will be graded in your project. I am not giving exact point values as it gives me the freedom to be lenient when necessary:

1. Your program should be able to successfully place ships onto a board.
2. Your program should be able to successfully print the status of the ships and the salvos.
3. Your program should be able to handle writing and reading the responses and salvos files.
4. Your program should be able to successfully determine when a ship is sunk.
5. Your program should be able to successfully determine when you have been defeated.
6. Your program should be able to successfully output the winner and loser of the game.
7. Your program should be able to successfully exit on the completion of a game.
8. Your program should provide a makefile that will properly compile your program.

In addition, please add your name as a comment to the top of each file.

Submission Instructions:

Make sure that before submitting your program that you have tested it on the ssh server (new domain: cs-ssh.uwf.edu). There is information for how to do this on eLearning. For this project, submit a zip file containing all files required to run your project. *For this project, you have the freedom of choosing your file layout, however this means that your makefile is vital to getting your program to compile*. Submit your project using the eLearning dropbox. Remember that all students who turn in their projects by the due date that receive a B or better will receive an added 0.5% on their final overall average.