Faculty of Arts & Sciences
Department of Computer Science
CMPS 270 – Software Construction
Fall 2024 – Course Project

Guidelines and Grading Policy

A program that does not compile will receive a **zero**. Make sure you upload your solution to your GitHub repository and maintain proper version control practices (commit your work regularly and meaningfully). You are also required to include the specifications of each method, in addition to the testing strategy and test-cases used to validate your program's correctness. You are also expected to submit a report explaining your strategy, and to present your project orally. The report, presentation, GitHub repository, specifications, and testing skills will account for 50% of the grade. The remaining 50% is for the program's correctness. Not every program that runs will be considered correct. Note that the testing strategy, specifications, report, and presentation are only expected as part of phase two's submission. In phase two, your program should perform better than random. It should perform the winning move if such a move exists. Note that the grades are not distributed evenly among the two phases, with phase two having a larger chunk.

Battleship - Game Description

In this project, you will develop an advanced version of the game Battleship, using the C programming language. This version will not only include the traditional gameplay but also introduce strategic elements such as radar sweeps, torpedo strikes, and more.

Battleship is a traditional strategy game that dates back to before World War I and was later commercialized as a popular board game by Hasbro. The primary objective of the game is to sink all of the opponent's ships.

Game Setup

- Number of players: 2
- Game Boards: Each player has one grid sized 10x10.
- Ships: Each player has a fleet of ships that vary in size and number. This fleet includes:
  (a) 1 Carrier (5 cells)
  (b) 1 Battleship (4 cells)
  (c) 1 Destroyer (3 cells)
  (d) 1 Submarine (2 cells)
- Objective: To sink all of the opponent's ships.

In this project, you will develop a digital version of Battleship that not only replicates its foundational mechanics but also introduces advanced gameplay elements to enhance strategic depth and player engagement. These elements include new types of weaponry, defensive tools, and different modes that challenge the player's tactical skills in new ways.

The project will be divided into two phases: building a two-player version in phase one, and then creating a bot for a human versus computer gameplay in phase two.

Phase One - Due October 16th at 7 am

Implement a two-player version of Battleship where players can interact through a console-based interface.

Requirements

Game Initialization:

Implement a function to create and display a 10x10 grid filled with water (∼). Originally, the grid should look like:

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
1	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
2	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
3	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
4	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
5	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
6	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
7	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
8	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
9	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼
10	∼	∼	∼	∼	∼	∼	∼	∼	∼	∼

- Next, your program should ask for the tracking difficulty level. Note that there are two tracking difficulty levels (easy, hard). In the easy mode, the grid will track both the hits and the misses, whereas in the hard mode, the grid will only track hits.

- The program should then ask each player to enter their name.

- Both players have an equal chance of playing first. Your program should randomly choose a first player and inform the players of the choice.

- Your program should then in turn ask each player to place their ships on a grid using coordinates and orientation (horizontal or vertical). The program should ask each player for the coordinates of every ship, one by one. Ships should not overlap with others inputted by the same player. Ships should not extend beyond the grid. Your program should display an error message if the player enters ship coordinates that overlap with another ship or extend beyond the grid. The coordinates entered by the player are the starting position of the ship. Horizontal moves from left to right, and vertical moves from top to bottom. For example, if the player enters *B3,horizontal* when asked for the coordinates of the destroyer, this means that the destroyer will occupy cells B3, C3, and D3. If instead the player entered *B3,vertical* this means that the destroyer will occupy cells B3, B4, and B5.
  Note that your program should clear the current screen to preserve the secrecy of the ship positions after every player is done entering their ships.

- Note that the game has two grids, one for each player. In other words, each grid contains the ships of one player. When displaying the grid at every new turn (described below), it is the opponent's

grid that should be displayed (keep in mind that each player is trying to hit their opponent's ships).

Gameplay:

Players take turns performing a single move per turn. At the beginning of each turn, your program should display the updated game grid of the current player's opponent, followed by the list of available moves. Your program should then ask the current player (using the player's name) for their choice and receive the player's move as an input. The list of possible moves are:

1. **Fire**: The basic action where a player tries to hit an opponent's ship by guessing a coordinate.
   Command structure: *Fire [coordinate]*. Example: *Fire B3*
   Command output: Either *hit* or *miss*

2. **Radar Sweep**: Reveals whether there are any *opponent* ships in a specified 2x2 area of the grid without showing exact locations of ships.
   Command structure: *Radar [top-left coordinate]*. Example: *Radar B3* reveals whether there are any opponent ships in cells B3, C3, B4, C4
   Command output: Either *Enemy ships found* or *No enemy ships found.*
   Note that every player is allowed *3* radar sweeps per the entire game. If a player attempts to perform more than 3 sweeps, they lose their turn.

3. **Smoke Screen**: Obscures a 2x2 area of the grid by hiding it from radar sweeps.
   Command structure: *Smoke [top-left coordinate]*. Example: *Smoke B3* obscures whether there are ships in cells B3, C3, B4, C4, such that if the opponent later on performs a radar sweep that touches any of these cells, it would count them as misses regardless of whether or not ships resided in them.
   Note that players are allowed one smoke screen per ship they have sunk. If a player attempts to perform more than their allowed limit, they lose their turn. (For example, a player who has sunk three ships so far and has only performed one smoke screen, can still be allowed to perform two more smoke screens).
   Also note that if the player performs a smoke screen move, the program should clear the current screen to preserve the secrecy of the move.

4. **Artillery**: An attack move that works similarly to **Fire** but targets a 2x2 area.
   Command structure: *Artillery [top-left coordinate]*. Example: *Artillery B3* will hit any ships in cells B3, C3, B4, C4
   Command output: Either *hit* or *miss*.
   Condition: this move is unlocked only once *during the next turn of the player who sinks the other player's ship in the current turn.*

5. **Torpedo**: A powerful attack that targets an entire row or column.
   Command structure: *Torpedo [row/column]*. Example: *Torpedo B* will hit any ships in column B
   Command output: Either *hit* or *miss*
   Condition: this move is unlocked only once *during the next turn of the player who sinks the other player's third ship in the current turn.*

Note that:

- The system should validate the coordinates and update the grid to show hits (*) and misses (o). Note that if the Hard difficulty tracking mode was chosen by the players, then the misses should not be displayed. If the player inputs invalid coordinates, they lose their turn.
- You should ensure proper turn-based mechanics so each player gets a chance to play after the other.
- After each move, your program should display the updated grid and a message indicating the result of the move, in addition to a message saying when a ship has been ship sunk (and what type of ship it was).
- The game ends when all ships of one player are sunk.
- At the end of the game, your program should display a message indicating who the winner is.

Phase Two - Due November 13th at 7 am

Implement a bot that plays the Battleship game. Write a program that runs similarly to Phase 1 but where one of the players is the bot you implemented. In other words, the program asks for a single player's name, then proceeds in a fashion similar to phase one and accepts moves submitted by the player and others generated by the bot. You may use any strategy to implement your bot, as long as you can clearly explain it in your oral presentation. Note that your bot should perform better than random, and perform winning moves whenever possible.

Bonus (Extra 10% of the project's grade)

Modify your program to offer different difficulty levels (Easy, Medium, Hard) that launch different bots with clear differences in terms of the strategies used. All levels should perform better than random.

Battle Royale

A knockout tournament between the bot implementations will be held at the end of the semester. The top 3 groups will receive bonus points that will be added to the final course grade. Participation in the competition is optional (but strongly encouraged!).

The competition will proceed as follows:

1. Pairs of players will be matched randomly.
2. Every pair of players will play two games, so that each player will have the chance to be the starting player.
3. If each of the two players wins a game, the tie will be broken based on a time rule: the winner will be the player that took less time making moves (across the two games).

More details will be shared later.

In addition to your GitHub submission, add your programs to one folder: group-name.battleship.zip (or .rar) and submit it to moodle.