

## Project Battleship: Documentation

### Overview

- Create a grid-based battleship game, where the player selects a location on the grid each turn, and either records a hit or miss. The game ends when all ships have been sunk.
- Implement a one-player mode with various difficulty levels that vary the size of the grid. The player will have a set amount of ammo to find all ships.
- Package output in concise form
  - C++ executable, include updated Makefile
- Considerations:
  - Front end:
    - Provide a visually appealing, intuitive interface with clear instructions for the player
    - Make the program resilient to unexpected inputs from the player
  - Back end:
    - The code should be optimized and well-organized, easy to read and debug
    - Unexpected inputs should be handled wherever inputs are taken
    - Format should facilitate development by multiple programmers: use GitHub, separate program into appropriate header files
  - Marketability
    - How does this vary/expand upon traditional tabletop version of battleship?
    - What are the benefits of creating a digital version of the game?
    - Other possibilities to expand the scope of the game?
- Gameplay outline:
  - Start screen/menu
  - Instructions/initialize starting grid
  - Game mode:
    - User input recorded, result output to user
    - User input stored to track progress in game
    - Endgame indicated and result output to user
    - Replay/exit

## Documentation:

When the player runs the executable, significant steps occur as follows:

- Open menu.cpp (main.cpp line 35)
  - Outputs the menu to the player, allows them to select a username
  - Player chooses to start the game, view credits, or exit
- Create battleship object (main.cpp line 42)
  - Initializes a battleship object, storing data on the ships for the game, the board to be used for the game, and the number of turns the player is allowed
- Call Instruction(difficultylevel) (main.cpp line 58)
  - User inputs difficulty level and relevant instruction set is called
  - In all cases, player is introduced to grid layout and shown how to interpret whether targets hit/miss
  - Player is given the rules for the ship size/layouts, as well as their number of turns
  - Difficulty-level-specific rules are shown to the player
- Call Initialize(difcultylevel) (main.cpp line 64)
  - Sets up the initial board to be used throughout the game:
    - setBoard() sets the size of the board
    - initializeBoard() creates the board of the specified size
    - populate() fills in the initial characters for the board
    - setmunitions() sets the amount of shots the player gets for the game
    - placeship() is called for each ship to be created and initializes a random location and direction to plot it. The function verifies that the generated location is valid, and plots each ship
    - theConsole() and printBoard() are called and print out the initial amount of ammo and hits and shows the starting grid
- Call getline(cin, objects) (main.cpp line 75)
  - Reads the player's input for the selected target
  - Calls setCoords(), checklinkedlist(), checkhit(), and dolinkedlist() to check if the user entered a valid grid location, and if so, registers the hit/miss and saves the entry to a linked list
  - Overrides the getline function
  - Is passed onto the board object using the override operator
- Call theList() (main.cpp line 81)
  - Prints out the linked list storing the ordered series of player's entries, and indicates whether each entry was a hit or miss

- Repeat previous two items until player wins or loses, and prompt player to play again or quit

#### Input handling when running

- User is prompted to input again if they enter a name with more than 1 word
- When entering in coordinates, the program will prompt user to input again if the first input is a number instead of a character or the user presses enter without inputting
- In the menu screen, the user will be prompted to input again if they select options outside of the ones displayed on the menu screen

#### Marketability

- Limited ammo adds to the complexity of battleship, and allows for competition between players and across multiple games
- Automatically storing player's entries simplifies the player experience
- Further expansions to prepare the game for marketing could include:
  - Building the game into an app
  - Add ability for player to input their own board
  - Allow the players to connect by sending boards to each other, or by storing high scores based on least number of turns to win, or number of consecutive games won

#### Expectations/Reality

- One of the unique aspects of a battleship-type game is the variety of information being stored. In the tabletop version for two players:
  - Player 1 has a grid containing their ships, and the coordinates of player 2's guesses
    - Store player 1 ships, player 2 guesses
  - Player 1 has a grid containing their guesses on player 2's grid
    - Store player 2 ships, player 1 guesses
  - Player 2 has a grid containing their ships, and the coordinates of player 1's guesses
    - Store player 2 ships, player 1 guesses
  - Player 2 has a grid containing their guesses on player 1's grid
    - Store player 1 ships, player 2 guesses
- Four sets of data are being stored: player 1/2 ship locations and player 1/2 guess locations
- The game ends when all of one player's guesses overlap with all of the other player's ships
- The difficulty in programming this game in C++ is to store all this information, and make the appropriate comparisons as well as displaying to the player the relevant information to them, without revealing any information that should be hidden

- In hindsight, learning about the STL opens up streamlined options to store a variety of data for a project such as this
- Without the STL, perhaps the best option would be to use a grid to simultaneously store various information, but this is difficult to keep track of and utilize
- Instead, our approach shifted as we rethought how to present the game to the player, in a way that simplifies the amount of information stored