Phoenix

November 2024

1 Group Members

Charbel Dawlabani, Elie Jbara, Crissie Tawk, Marianne Elias

2 GitHub URL

https://github.com/Dawlabani/CMPS270Project1

3. High-level Description and Strategies Used

This project presents an advanced implementation of the classic Battleship game in the C programming language, featuring both human and computer-controlled players. The game is enriched with special moves and an intelligent bot opponent that elevates the traditional gameplay to a more strategic and challenging level.

At its core, the game is structured around several key components: data structures representing the game entities, functions for initializing the game state, gameplay mechanics that define how the game progresses, and the artificial intelligence strategies that govern the bot's actions.

Data Structures and Initialization

The game utilizes structured data types to represent the essential elements:

- **Coordinate**: Defines positions on the grid with x and y values.
- **Ship**: Contains information about each ship, such as its name, size, hit count, sunk status, and symbol.
- **Fleet**: Represents a collection of ships belonging to a player.
- **Player**: Holds player-specific data, including their grids, special move availability, and status of their fleet.

The initializePlayer and initializeGrid functions set up the initial state for each player, preparing empty grids and resetting all counters and flags to their default values.

Gameplay Mechanics

The game operates on a turn-based system where players alternate turns to perform actions. Each player has access to a variety of moves:

- **Fire**: Attack a specific coordinate on the opponent's grid.
- **Radar Sweep**: Scan a 3x3 area to detect nearby ships, limited to three uses.
- **Smoke Screen**: Deploy a defensive area that obscures the player's ships from radar detection.
- **Artillery Strike**: Attack a 3x3 area, unlocked after sinking an opponent's ship.
- **Torpedo Attack**: Target an entire row or column, unlocked when the opponent has only one ship remaining.

Bot Implementation and Strategies

The bot is designed to simulate human-like decision-making while leveraging algorithmic strategies to optimize its gameplay. During ship placement, the bot randomly positions its ships on the grid, ensuring that they do not overlap and stay within the grid boundaries. This randomness adds unpredictability to the game, making it more challenging for the human player to locate the bot's ships.

During its turn, the bot follows a hierarchical decision-making process:

1. **Smoke Screen Deployment**: If the bot has smoke screens available and has recently had ships hit, it prioritizes deploying a smoke screen over areas where its ships are located. This defensive tactic aims to prevent the human player from using radar sweeps effectively in those areas.
2. **Artillery and Torpedo Usage**: When special moves are available, the bot assesses the game state to decide the optimal time to use them. For artillery strikes, it calculates the coordinate that would cover the most untargeted squares, increasing the chance of hitting the player's ships. For torpedo attacks, it analyzes rows and columns to find the one with the highest number of untargeted cells.
3. **Targeted Attacks**: If the bot has previously hit a ship, it enters a targeting mode where it focuses on adjacent coordinates. This strategy increases the likelihood of sinking the ship by methodically attacking surrounding positions.
4. **Radar Sweeps**: When lacking immediate targets and with radar sweeps remaining, the bot may perform a radar sweep to reveal potential ship locations. It chooses coordinates that cover unexplored areas to maximize the effectiveness of the sweep.
5. **Random Attacks**: If no better options are available, the bot selects random coordinates to fire upon, ensuring that it does not choose the same coordinate twice.

Intelligent Decision-Making

The bot's intelligence is derived from its ability to adapt to the game's progression. It maintains a list of potential targets, especially after successful hits, and updates this list based on subsequent results. The bot avoids redundant actions by checking whether a coordinate has already been targeted or is under the effect of a smoke screen.

When using the artillery strike, the bot evaluates all possible coordinates to find the one that would potentially yield the highest number of hits. It counts the number of untargeted tiles within the artillery's area of effect for each coordinate, selecting the one with the maximum count.

For torpedo attacks, the bot calculates the number of untargeted cells in each row and column, choosing to attack the one with the highest count. This approach increases the probability of hitting the player's remaining ships, especially in the late game.

Balancing Randomness and Strategy

While the bot incorporates random elements in its decision-making, particularly in initial attacks, it balances this with strategic planning. The randomness ensures that the bot's actions are not entirely predictable, while the strategic components make it a formidable opponent that can adapt to the player's tactics.

User Interaction and Experience

The game provides clear feedback to the player after each move, displaying messages about hits, misses, and sunk ships. Special moves are introduced progressively, adding new layers of strategy as the game advances. The bot's actions are described to the player, maintaining transparency about its decisions without revealing specific internal logic.

Conclusion

The implementation successfully delivers a complex and engaging Battleship game that challenges players with an intelligent bot opponent. By combining strategic algorithms with adaptive behaviors, the bot enhances the gameplay experience, requiring players to think critically and adapt their strategies. The code's modular design allows for easy maintenance and potential future enhancements, such as increasing the bot's difficulty or introducing additional features.

4. Issues

During the development of the Battleship game, several significant challenges emerged that affected both the gameplay mechanics and the overall functionality of the program. These issues primarily centered around handling edge cases for special moves that operate over a 2x2 grid area, designing the bot's intelligent behavior to make it a formidable opponent, managing the bot's deployment of smoke screens effectively, and addressing the unintended side effects that occurred when modifying interdependent game functions.

Handling Edge Cases in Special Moves

A major issue was the proper handling of edge cases for special moves like smoke screens, radar sweeps, and artillery strikes. These moves affect a 2x2 area on the game grid, which posed problems when the target area was located near the edges or corners of the grid. In such cases, part of the intended area would fall outside the grid boundaries, leading to potential out-of-bounds errors or incomplete execution of the move.

For instance, if a player attempted to deploy an artillery strike at coordinate J10 (the bottom-right corner of the grid), the program needed to account for the fact that a 2x2 area centered on this coordinate would extend beyond the grid's limits. Failing to handle these edge cases could result in the game crashing or the move not functioning as intended, thus disrupting the gameplay experience.

Mapping Out the Bot's Intelligent Behavior

Designing the bot to be a smart and challenging opponent was another significant challenge. Determining the optimal strategies for the bot required careful consideration to balance difficulty and fairness. It was essential to program the bot to make decisions that mimicked human strategic thinking while also leveraging computational advantages.

One approach was to implement a checkerboard strategy after the bot scored a hit on a player's ship. The idea was for the bot to target adjacent tiles around the successful hit to increase the chances of sinking the ship quickly. However, programming this behavior introduced complexity in tracking the state of the game and ensuring the bot could efficiently select the most promising targets without wasting moves.

Moreover, making the bot too predictable or too random could either frustrate the player or make the game unchallenging. Striking the right balance in the bot's decision-making algorithms was a nuanced problem that required extensive testing and refinement.

Bot's Deployment of Smoke Screens

Effectively managing the bot's use of smoke screens presented another layer of difficulty. The challenge lay in programming the bot to deploy smoke screens over areas where its ships were located to reduce the player's ability to detect them using radar sweeps. However, keeping track of the bot's ship placements and deciding when and where to deploy smoke screens without making the bot's behavior predictable was complex.

Deploying smoke screens indiscriminately could lead to them being placed over empty areas, rendering them useless and potentially giving the player unintended hints about where ships were not located. Conversely, always deploying smoke screens immediately after a ship was hit could signal to the player that a ship was in that vicinity.

The difficulty was in programming the bot to make strategic decisions about smoke screen deployment that protected its ships effectively while maintaining a level of unpredictability to keep the game challenging for the player.

Interdependency and Conflicts Between Game Functions

An ongoing issue was the interdependency of game functions, where modifying one method could lead to unintended consequences in others. This was particularly evident with the smoke screen and radar sweep functionalities. Adjustments made to the smoke screen mechanics, such as changing how the 2x2 area was calculated or stored, sometimes caused the radar sweep function to malfunction.

For example, after modifying the smoke screen deployment to correctly handle edge cases, the radar sweep might begin to incorrectly identify areas covered by smoke screens or fail to detect ships in areas that were not actually obscured. This conflict arose because both functions relied on shared data structures and needed to interpret the game grid consistently.

These unintended side effects made debugging challenging, as fixing one issue could inadvertently introduce new bugs elsewhere. Ensuring that all game functions interacted seamlessly and maintained the integrity of shared resources required meticulous attention to detail and thorough testing.

Example of Conflicting Functionalities

A specific example of this issue occurred when we attempted to fix a bug in the smoke screen deployment logic. The smoke screen was supposed to prevent the radar sweep from detecting ships within its area of effect. However, after modifying the smoke screen function to handle edge cases properly, we found that the radar sweep no longer functioned correctly.

The radar sweep began to either ignore smoke screens entirely or incorrectly assume that larger portions of the grid were covered by smoke, even when they were not. This led to situations where the player could not detect ships in areas that should have been clear or received misleading information about the presence of enemy ships.

This example highlighted the complexity of managing interdependent functions and the challenges involved in ensuring that changes in one part of the code did not negatively impact other areas.

5 Resolutions

To address these challenges, we implemented several solutions that involved refining our code logic, enhancing our algorithms, and improving our testing procedures.
Handling Edge Cases in Special Moves

To resolve the issues with edge cases in special moves, we introduced boundary checks and adjusted the calculations for the areas of effect to ensure they remained within the grid's limits. Specifically, we created a function called handleEdgeCoordinates that adjusted the starting and ending indices for the x and y coordinates based on the grid boundaries. For example, when a special move like an artillery strike was initiated at the edge of the grid, the function ensured that the area of effect did not exceed the grid's size by setting the coordinates to the maximum allowable values. This prevented out-of-bounds errors and ensured that the moves executed correctly regardless of their position on the grid. We also standardized the way each special move calculated its area of effect, using consistent logic across the smoke screen, radar sweep, and artillery strike functions. This consistency reduced the likelihood of errors and made the code easier to maintain.

Mapping Out the Bot's Intelligent Behavior

To enhance the bot's intelligence, we developed algorithms that allowed it to adapt its strategy based on the game state. After the bot scored a hit, it would add the adjacent coordinates to a list of potential targets. This approach enabled the bot to focus on sinking ships efficiently by targeting surrounding areas where the remainder of the ship might be located. We also implemented a probability-based targeting system where the bot prioritized coordinates that had a higher likelihood of containing a ship based on previous hits and the remaining ships' sizes. The bot used radar sweeps strategically when it had no immediate targets, helping it to locate ships more effectively without relying solely on random guesses. To prevent the bot from becoming too predictable, we introduced randomness into its decision-making processes, such as occasionally choosing to deploy a smoke screen or use a special move at different times rather than always following a set pattern.

Bot's Deployment of Smoke Screens

We improved the bot's smoke screen deployment by programming it to monitor the status of its ships and the player's recent actions. The bot now deploys smoke screens in areas where its ships are located, especially if the player has recently targeted nearby coordinates. This defensive strategy helps protect the bot's ships from detection without making its behavior too predictable.

To keep track of its ship placements, the bot maintains an internal map of its grid, allowing it to identify critical areas that need protection. We also randomized the timing of smoke screen deployment to prevent the player from easily deducing the bot's ship locations based on when and where smoke screens appear.

Interdependency and Conflicts Between Game Functions

To resolve the conflicts between interdependent functions, we modularized our code and clearly defined the interfaces between different functionalities. By isolating the logic for smoke screens and radar sweeps into separate, well-defined functions, we minimized the risk of unintended side effects when changes were made. We also implemented comprehensive unit tests for each function, allowing us to detect and fix bugs more efficiently. When modifying one function, we ran these tests to ensure that other functions continued to operate correctly. This approach helped us identify issues early in the development process and maintain the integrity of the game's overall functionality.

Example of Conflicting Functionalities Resolved

In the case of the smoke screen and radar sweep conflict, we revisited the way both functions interacted with the game grid. We ensured that the smoke screen's area of effect was accurately recorded and that the radar sweep function checked against this data when determining which areas were obscured. By standardizing the data structures used to represent smoke screens and ensuring that both functions accessed this information consistently, we eliminated the discrepancies that led to incorrect behavior. We also added additional checks in the radar sweep function to handle edge cases where the smoke screen's area might overlap partially with the radar sweep's area, ensuring accurate detection of ships outside the smoke-covered zones.

Through these resolutions, we successfully addressed the challenges encountered during development. By refining our code, enhancing the bot's intelligence, and improving our testing procedures, we created a more robust and enjoyable Battleship game. The bot became a smarter

and more challenging opponent, and the game's special moves functioned correctly across all scenarios, providing players with a strategic and engaging experience.

6 Limitations

Despite the comprehensive features and strategic depth incorporated into the Battleship game, there are several limitations that affect the overall gameplay experience and player engagement. One significant limitation is the predictability of the AI's behavior. The AI tends to use all its radar functions at the very beginning of the game, deploys smoke screens as soon as possible, and utilizes artillery strikes and torpedo attacks whenever they become available. While this aggressive strategy makes the AI a competitive and formidable opponent, it also introduces a level of predictability that experienced players might exploit. This design choice was intentional to keep the game exciting and to ensure that the AI poses a significant challenge by attempting to cloak its ships quickly to prevent the player from winning easily and by maximizing its offensive capabilities early on.

However, the consistent pattern of the AI's actions can reduce the challenge over time, as players learn to anticipate and counter its strategies. The AI's lack of adaptive behavior means it does not adjust its tactics based on the player's actions or previous games, which limits the game's replayability and strategic depth. Incorporating more varied and unpredictable AI behaviors could enhance the challenge and keep players engaged over multiple playthroughs.

Another limitation lies in the game's pacing after all special moves have been utilized. Once the radar sweeps, smoke screens, artillery strikes, and torpedo attacks have been exhausted, the game often devolves into random firing across the grid. This shift can make the late-game experience less exciting and more reliant on chance rather than strategy. The absence of new mechanics or

strategic options at this stage means that players might find the gameplay repetitive and less engaging as they search for the remaining ships without additional tools or guidance.

To address this issue, the game could benefit from introducing hints or assistance mechanisms that help players locate enemy ships in the late game. For example, offering a limited number of hints after a certain number of consecutive misses could maintain player interest and reduce frustration. Additionally, implementing a feature where new ships arrive or sunk ships are revived after sinking a specific number of opponent ships could add a dynamic element to the gameplay. Trading current special moves like smoke screens and artillery strikes for this new mechanic might balance the game and refresh the strategic options available to players.

Moreover, the balance of the game can be affected by the AI's early aggressive use of powerful special moves. The AI's immediate deployment of smoke screens and utilization of artillery and torpedoes can lead to early dominance, making it challenging for players to recover. Without mechanisms to counteract these strong opening moves, players might feel at a disadvantage, impacting their overall enjoyment of the game. Introducing defensive special moves or opportunities for players to regain lost advantages could improve the game's balance and fairness.

In conclusion, while the Battleship game offers a range of strategic features and an AI designed to be competitive, the predictability of the AI's behavior and the lack of late-game excitement are notable limitations. Enhancing the AI to exhibit more varied and adaptive strategies would increase the challenge and replayability. Additionally, introducing new mechanics in the late game, such as hints or ship revivals, could maintain player engagement and add depth to the gameplay. By refining these aspects, the game can provide a more dynamic and engaging

experience that appeals to both new and experienced players, encouraging repeated playthroughs and long-term enjoyment.

7 Assumptions

In the development of the Battleship game, several key assumptions were made to align the code implementation with the intended game design and mechanics. These assumptions guided the handling of special moves, input validation, AI behavior, and the presentation of game information to the players, ensuring a cohesive and balanced gameplay experience.

One primary assumption is that the smoke screen special move remains active for only one radar sweep. This means that once a smoke screen is deployed, its effect is temporary, obscuring the player's radar scans for a single turn. After this turn, the smoke screen dissipates, allowing normal radar functionality to resume. This limitation was implemented to balance the defensive advantage provided by the smoke screen, preventing it from permanently hindering the player's ability to detect ships and maintaining the game's strategic equilibrium.

Another critical assumption is that any invalid input entered by a player results in the loss of their turn. This rule was established following the instructions provided by Dr. Zalghout, ensuring that players adhere strictly to the expected input formats and commands. While this approach increases the game's difficulty by penalizing errors, it also reinforces the importance of accurate input and strategic planning. Players must carefully consider their moves to avoid forfeiting their turns, thereby enhancing the game's challenge and engagement.

Error handling was a significant focus area, particularly concerning the placement of ships on the map and targeting already targeted tiles. It was assumed that robust error checking mechanisms would be in place to prevent players from placing ships outside the grid boundaries or overlapping ships. Additionally, the game was designed to detect and respond appropriately

when a player attempts to target a coordinate that has already been fired upon. This ensures the integrity of the gameplay by preventing redundant actions and maintaining a clear and fair game state.

For special moves such as radar sweeps, artillery strikes, and smoke screen deployments, it was assumed that these moves would only accept coordinates positioned at the edge of the map, such as J9. This design choice ensures that the area-of-effect mechanics of these moves are consistently applied without exceeding the grid's limits. By restricting the activation of these special moves to edge coordinates, the game maintains predictable and manageable areas of impact, simplifying both implementation and player strategy.

The AI bot was programmed with a prioritized system for utilizing special moves, following the sequence: smoke screen, torpedo, artillery, and finally, standard fire. This priority system was based on the strategic importance of each move, with smoke screens offering defensive capabilities, torpedoes providing powerful offensive strikes, and artillery strikes serving as area-of-effect attacks. By adhering to this hierarchy, the bot was designed to maximize its effectiveness in both protecting its fleet and attacking the player's ships, thereby presenting a challenging opponent that balances offense and defense.

In terms of game presentation, it was assumed that each player would only be able to view the map of their opponent, not their own map. This design choice maintains the classic element of Battleship, where players must deduce the locations of enemy ships without revealing their own ship placements. Furthermore, when a smoke screen is deployed, the game does not disclose the exact coordinates of the deployment. Instead, the map is immediately hidden, adding an element of mystery and preventing players from gaining information about the bot's defensive maneuvers. This approach enhances the game's strategic depth, as players must infer the

presence of smoke screens and adjust their strategies accordingly without direct knowledge of their locations.

These assumptions collectively shaped the game's mechanics and user experience, ensuring a balanced and engaging Battleship game. By limiting the duration of smoke screens, enforcing strict input validation, prioritizing special moves for the AI bot, and controlling the visibility of game maps, the development team was able to create a game that is both challenging and faithful to the traditional Battleship gameplay.

8 Bonus Part—Bot Difficulty

As an enhancement to the standard gameplay, we have incorporated a bonus feature that introduces varying difficulty levels for the AI-controlled bot opponent. This addition caters to players of different skill levels and provides a customizable challenge that enhances the overall gaming experience. The bot now operates under three distinct difficulty settings: Easy, Medium, and Hard, each meticulously designed with specific behaviors and strategies.

Easy Difficulty

- **Behavior and Strategy:**
    - The bot exhibits straightforward and less aggressive behavior.
    - Prioritizes basic firing actions.
    - Does not employ advanced targeting strategies after scoring a hit unless a radar sweep has previously detected enemy ships.
- **Special Moves Usage:**
    - Radar Sweep: Performed between the 6th and 10th turns in every 10-turn interval.
    - Artillery Strike: Deployed between the 7th and 10th turns when available.
    - Smoke Screen: Used on the 10th turn of every 10-turn interval when available.

    - Torpedo Attack: Attempted between the 10th and 15th turns in every 15-turn interval when available.

- **Purpose:**
    - Ensures the bot remains a manageable opponent.
    - Suitable for novice players or those seeking a more relaxed game.
    - Allows players to learn and enjoy the game mechanics without excessive difficulty.

Medium Difficulty

- **Behavior and Strategy:**
    - Offers a balanced challenge with more strategic gameplay.
    - Utilizes smarter targeting behavior, entering targeting mode after successful hits to focus on sinking ships efficiently.
- **Special Moves Usage:**
    - Uses special moves with higher probability.
    - Radar Sweeps: Utilized more frequently to detect player ships.
    - Artillery, Torpedoes, and Smoke Screens: Used more often and less predictably.
- **Purpose:**
    - Suitable for players with some experience.
    - Requires players to adapt their strategies and think more critically to outmaneuver the bot.

Hard Difficulty

- **Behavior and Strategy:**
    - Presents a formidable challenge with advanced strategies and aggressive tactics.

    - Aggressively seeks to detect and destroy the player's ships.
    - Enters targeting mode after each successful hit.
- **Special Moves Usage:**
    - Uses all special moves as soon as they become available, maximizing offensive and defensive capabilities.
    - Strategically deploys smoke screens to protect its fleet.
- **Purpose:**
    - Demands that players employ advanced strategies.
    - Requires constant vigilance to compete effectively.
    - Ideal for players seeking a high level of difficulty and challenge.

By implementing these difficulty levels, we have enhanced the replayability and accessibility of the game. Players can select the level that best matches their skill and desired challenge, making the game enjoyable for a wider audience. The varying behaviors of the bot across difficulty levels add depth to the gameplay, encouraging players to improve their skills and strategies to overcome increasingly challenging opponents.

Specifications and Test Cases for Battleship Game Methods

Below are the detailed specifications and test cases for the methods defined in the Battleship game implementation.

Table of Contents

---

1. initializePlayer
Specification
Purpose: Initializes a Player structure with default values, including setting up empty grids and resetting all counters and flags.
Prototype:
void initializePlayer(Player* player, bool isBot, DifficultyLevel difficulty);
Parameters:

- player: Pointer to the Player structure to initialize.
- isBot: Boolean indicating whether the player is a bot (true) or a human player (false).
- difficulty: The difficulty level for the bot (ignored for human players).

Behavior:

- Initializes the player's grid and tracking grid with empty water ('~').
- Resets all counters, including radar sweeps used, smoke screens used, ships sunk, and ships remaining.
- Sets the availability of special moves (artilleryAvailable, torpedoAvailable) to false.
- Initializes bot-specific fields like potential targets and last artillery coordinates.
- Resets the turn number for the bot.

Test Cases

1. **Test Case 1**: Initialize a human player.

Input:
Player player;
initializePlayer(&player, false, MEDIUM);
Expected Result:

- player.isBot is false.
- Grids are initialized with '~'.
- All counters are set to zero.
- Special moves are unavailable.
- Difficulty level is set to MEDIUM.

2. **Test Case 2**: Initialize a bot player at HARD difficulty.

Input:
Player bot;
initializePlayer(&bot, true, HARD);
Expected Result:

- bot.isBot is true.
- Grids are initialized with '~'.
- All counters are set to zero.
- Special moves are unavailable.
- Difficulty level is set to HARD.
- Bot-specific fields are properly initialized.

---

2. initializeGrid
Specification
Purpose: Initializes a game grid by filling it with water ('~').
Prototype:
void initializeGrid(char grid[GRID_SIZE][GRID_SIZE]);
Parameters:

- grid: A 2D array representing the game grid to initialize.

Behavior:

- Sets every cell in the grid to '~'.

Test Cases

1. **Test Case**: Initialize a grid.

Input:

```
char grid[GRID_SIZE][GRID_SIZE];
initializeGrid(grid);
```

Expected Result:

- All cells in grid contain '~'.

---

3. displayGrid

Specification

Purpose: Displays the game grid to the console, optionally showing ships.

Prototype:

```
void displayGrid(char grid[GRID_SIZE][GRID_SIZE], bool showShips);
```

Parameters:

- grid: The grid to display.
- showShips: If true, displays ships; if false, hides ship symbols.

Behavior:

- Prints the grid with column headers (A-J) and row numbers (1-10).
- Replaces ship symbols with '~' if showShips is false.

Test Cases

1. **Test Case 1**: Display grid without ships.

Input:

```
char grid[GRID_SIZE][GRID_SIZE];
initializeGrid(grid);
grid[0][0] = 'C'; // Place a ship symbol
displayGrid(grid, false);
```

Expected Result:

- The displayed grid shows '~' at position (0,0), hiding the ship.

2. **Test Case 2**: Display grid with ships.

Input:

```
displayGrid(grid, true);
```

Expected Result:

- The displayed grid shows 'C' at position (0,0), revealing the ship.

---

4. placeShips

Specification

Purpose: Allows a human player to place their ships on the grid.

Prototype:

```
void placeShips(Player* player, Fleet* fleet);
```

Parameters:

- player: Pointer to the player placing ships.
- fleet: Pointer to the player's fleet.

Behavior:

- Prompts the player to input coordinates and orientation for each ship.
- Validates placement and updates the player's grid accordingly.
- Clears the screen after successful placement.

Note:
- If player->isBot is true, calls placeShipsBot instead.

Test Cases

1. **Test Case**: Simulate ship placement (assuming valid inputs).

Input:
   - Player inputs: "A1 h" for Carrier, "B2 v" for Battleship, etc.

Expected Result:
   - Ships are placed at the specified coordinates with the correct orientation.
   - Grid is updated accordingly.

---

5. placeShipsBot

Specification

Purpose: Automatically places ships for a bot player randomly on the grid.

Prototype:

void placeShipsBot(Player* bot, Fleet* fleet);

Parameters:
- bot: Pointer to the bot player.
- fleet: Pointer to the bot's fleet.

Behavior:
- Randomly selects starting coordinates and orientations for each ship.
- Ensures ships do not overlap and are within grid boundaries.
- Updates the bot's grid with ship placements.

Test Cases

1. **Test Case**: Verify that ships are placed without overlapping.

Input:

Player bot;
Fleet fleet;
initializePlayer(&bot, true, MEDIUM);
placeShipsBot(&bot, &fleet);

Expected Result:
   - All ships are placed on the bot's grid.
   - No ships overlap.
   - All ships are within the grid boundaries.

---

6. isValidPlacement

Specification

Purpose: Checks if a ship can be placed at the specified location without overlapping and within grid boundaries.

Prototype:

bool isValidPlacement(char grid[GRID_SIZE][GRID_SIZE], Coordinate coord, int size, char orientation);

Parameters:
- grid: The grid to check.

- coord: Starting coordinate for the ship.
- size: Size of the ship.
- orientation: 'h' for horizontal, 'v' for vertical.

Returns:
- true if the placement is valid.
- false otherwise.

Behavior:
- Verifies that the ship fits within the grid.
- Checks that the placement does not overlap with existing ships.

Test Cases

1. **Test Case 1**: Valid horizontal placement.

Input:
Coordinate coord = {0, 0};
int size = 5;
char orientation = 'h';
bool result = isValidPlacement(grid, coord, size, orientation);
Expected Result:
    - result is true if positions (0,0) to (4,0) are empty.
2. **Test Case 2**: Invalid vertical placement (overlapping).

Input:
grid[1][0] = 'C'; // Existing ship
Coordinate coord = {0, 0};
int size = 4;
char orientation = 'v';
bool result = isValidPlacement(grid, coord, size, orientation);
Expected Result:
    - result is false due to overlap at (0,1).
3. **Test Case 3**: Invalid placement (out of bounds).

Input:
Coordinate coord = {8, 0};
int size = 3;
char orientation = 'h';
bool result = isValidPlacement(grid, coord, size, orientation);
Expected Result:
    - result is false because the ship would extend beyond column 9.

---

7. placeShipOnGrid
Specification
Purpose: Places a ship on the grid at the specified location and orientation.
Prototype:
void placeShipOnGrid(char grid[GRID_SIZE][GRID_SIZE], Coordinate coord, int size, char orientation, char symbol);
Parameters:
- grid: The grid to update.
- coord: Starting coordinate for the ship.
- size: Size of the ship.

- orientation: 'h' for horizontal, 'v' for vertical.
- symbol: Character symbol representing the ship.

Behavior:

- Updates the grid cells with the ship's symbol along the specified orientation and size.

Test Cases

1. **Test Case**: Place a ship on the grid.

Input:

Coordinate coord = {0, 0};
int size = 3;
char orientation = 'v';
char symbol = 'D';
placeShipOnGrid(grid, coord, size, orientation, symbol);

Expected Result:

- Grid cells at positions (0,0), (0,1), (0,2) contain 'D'.

---

8. parseCoordinate

Specification

Purpose: Parses a coordinate string (e.g., "A5") into a Coordinate struct.

Prototype:

Coordinate parseCoordinate(const char* input);

Parameters:

- input: String representing the coordinate.

Returns:

- Coordinate with valid x and y if parsing is successful.
- Coordinate with x = -1 and y = -1 if invalid.

Behavior:

- Converts column letter to x index (0-9).
- Converts row number to y index (0-9).

Test Cases

1. **Test Case 1**: Valid coordinate.

Input:

Coordinate coord = parseCoordinate("A1");

Expected Result:

- coord.x is 0, coord.y is 0.

2. **Test Case 2**: Invalid coordinate (out of bounds).

Input:

Coordinate coord = parseCoordinate("K11");

Expected Result:

- coord.x is -1, coord.y is -1.

3. **Test Case 3**: Invalid format.

Input:

Coordinate coord = parseCoordinate("1A");

Expected Result:

- coord.x is -1, coord.y is -1.

---

9. clearScreen

Specification
Purpose: Clears the console screen.
Prototype:
void clearScreen();
Behavior:

- Executes the appropriate system command to clear the console screen based on the operating system.

Test Cases

1. **Test Case**: Invoke clearScreen and verify that the console is cleared.

Input:
clearScreen();
Expected Result:

- Console screen is cleared.

---

10. gameLoop
Specification
Purpose: Main game loop handling turns and checking for win conditions.
Prototype:
void gameLoop(Player* currentPlayer, Player* opponent, Fleet* currentFleet, Fleet* opponentFleet, bool hardMode);
Parameters:

- currentPlayer: Pointer to the player whose turn it is.
- opponent: Pointer to the opponent player.
- currentFleet: Pointer to the current player's fleet.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled (affects tracking grid visibility).

Behavior:

- Alternates turns between players.
- Invokes performMove or performBotMove based on the player type.
- Checks for win conditions after each turn.
- Ends the loop when a player wins.

Test Cases

1. **Test Case**: Simulate a game loop where a player wins after sinking all opponent's ships.

Input:

- Initialize players and fleets.
- Set up conditions for a quick game (e.g., small grid or pre-determined moves).

Expected Result:

- Game loop runs until checkWin returns true.
- The winning player's name is printed.

---

11. performMove
Specification
Purpose: Handles a human player's move, processing commands and executing actions.
Prototype:
void performMove(Player* player, Player* opponent, Fleet* opponentFleet, bool hardMode);
Parameters:

- player: Pointer to the player making the move.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Displays the player's tracking grid.
- Prompts the player for a move (fire, radar, smoke, artillery, torpedo).
- Validates the input command and arguments.
- Executes the move and updates the game state.
- Provides feedback to the player about the result.

Test Cases

1. **Test Case 1**: Player fires at a valid coordinate and hits a ship.

Input:

   - Command: "fire A1"

Expected Result:

   - If a ship is at (0,0), reports "Hit!".
   - Updates tracking grid.

2. **Test Case 2**: Player enters an invalid command.

Input:

   - Command: "fly A1"

Expected Result:

   - Reports "Invalid command or command not available."
   - Player loses their turn.

---

12. performBotMove

Specification

Purpose: Handles the bot's move based on its difficulty level.

Prototype:

void performBotMove(Player* bot, Player* opponent, Fleet* opponentFleet, bool hardMode);

Parameters:

- bot: Pointer to the bot player.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Determines the bot's action based on its difficulty level.
- Uses strategies appropriate for EASY, MEDIUM, or HARD difficulties.
- Executes special moves or fires at the player's grid.
- Updates the game state and provides feedback.

Test Cases

1. **Test Case**: Bot in EASY difficulty performs a random fire.

Input:

   - Bot's difficulty set to EASY.

Expected Result:

   - Bot selects a random untargeted coordinate.
   - Updates tracking grid accordingly.

---

13. fire

Specification

Purpose: Fires at a specified coordinate, updating grids and ship statuses.

Prototype:

int fire(Player* player, Player* opponent, Fleet* opponentFleet, Coordinate coord, bool hardMode, char* sunkShipName);

Parameters:

- player: Pointer to the player making the fire.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- coord: Coordinate to fire at.
- hardMode: Boolean indicating if hard mode is enabled.
- sunkShipName: Buffer to store the name of a sunk ship, if any.

Returns:

- 0: Miss.
- 1: Hit.
- 2: Hit and sunk a ship.
- 3: Already targeted.
- -1: Invalid or ineffective shot.

Behavior:

- Checks if the coordinate is under a smoke screen.
- Updates opponent's grid based on the result.
- Updates player's tracking grid if appropriate.
- Updates ship hit counts and sunk status.

Test Cases

1. **Test Case 1**: Fire at an empty cell.

Input:

Coordinate coord = {0, 0};

int result = fire(player, opponent, opponentFleet, coord, false, sunkShipName);

Expected Result:

    - result is 0 (Miss).
    - Opponent's grid at (0,0) is updated to 'o'.

2. **Test Case 2**: Fire at a ship and sink it.

Input:

    - Coordinate of the last remaining part of a ship.

Expected Result:

    - result is 2 (Hit and sunk).
    - sunkShipName contains the name of the sunk ship.
    - Ship's sunk status is updated.

---

14. radarSweep

Specification

Purpose: Performs a radar sweep at the specified coordinate.

Prototype:

void radarSweep(Player* player, Player* opponent, Coordinate coord);

Parameters:

- player: Pointer to the player performing the radar sweep.
- opponent: Pointer to the opponent player.
- coord: Coordinate where the radar sweep is deployed.

Behavior:

- Checks a 2x2 area around the coordinate for enemy ships.
- Considers active smoke screens that may obscure the area.
- Reports to the player whether enemy ships were detected.
- If the area is obscured by smoke, the radar sweep is ineffective, and the smoke screen deactivates.

Test Cases

1. **Test Case**: Radar sweep detects enemy ships.

Input:

- Radar sweep at coordinate where opponent has ships within the area.

Expected Result:

- Reports "Radar detected enemy ships near the target area."
- If the player is a bot, potential targets are added.

2. **Test Case**: Radar sweep area is covered by a smoke screen.

Input:

- Radar sweep at coordinate covered by an active smoke screen.

Expected Result:

- Reports "Radar sweep found no enemy ships (area obscured by smoke)."
- Smoke screen deactivates.

---

15. smokeScreen

Specification

Purpose: Deploys a smoke screen at the specified coordinate.

Prototype:

bool smokeScreen(Player* player, Coordinate coord);

Parameters:

- player: Pointer to the player deploying the smoke screen.
- coord: Coordinate where the smoke screen is deployed.

Returns:

- true if successfully deployed.
- false otherwise.

Behavior:

- Checks if the player has smoke screens available.
- Deploys the smoke screen, activating it over a 2x2 area.
- Updates the player's smoke screen data.

Test Cases

1. **Test Case 1**: Deploy smoke screen successfully.

Input:

player.shipsSunk = 1;
player.smokeScreensUsed = 0;
bool result = smokeScreen(&player, (Coordinate){0, 0});

Expected Result:

    - result is true.
    - Smoke screen is active at the specified coordinate.
2. **Test Case 2**: Fail to deploy smoke screen (none available).

Input:
player.shipsSunk = 0;
player.smokeScreensUsed = 0;
bool result = smokeScreen(&player, (Coordinate){0, 0});
Expected Result:
    - result is false.
    - Reports "No smoke screens available."

---

16. artillery
Specification
Purpose: Performs an artillery strike at the specified coordinate.
Prototype:
void artillery(Player* player, Player* opponent, Fleet* opponentFleet, Coordinate coord, bool hardMode);
Parameters:
- player: Pointer to the player performing the artillery strike.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- coord: Coordinate where the artillery strike is deployed.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:
- Attacks a 2x2 area centered around the coordinate.
- Updates grids and ship statuses based on hits.
- Reports total hits and misses.
- If ships are sunk, updates the game state accordingly.

Test Cases
1. **Test Case**: Perform artillery strike and sink a ship.

Input:
    - Artillery strike at coordinate covering the last parts of a ship.
Expected Result:
    - Reports hits and misses.
    - Sunk ship is reported.
    - Ship's sunk status is updated.

---

17. torpedo
Specification
Purpose: Performs a torpedo attack on a specified row or column.
Prototype:
void torpedo(Player* player, Player* opponent, Fleet* opponentFleet, const char* input, bool hardMode);
Parameters:
- player: Pointer to the player performing the torpedo attack.
- opponent: Pointer to the opponent player.

- opponentFleet: Pointer to the opponent's fleet.
- input: String indicating the row (number) or column (letter) to attack.
- hardMode: Boolean indicating if hard mode is enabled.

Behavior:

- Determines if the input is a row or column.
- Attacks all cells in the specified row or column.
- Updates grids and ship statuses based on hits.
- Reports total hits and misses.
- If ships are sunk, updates the game state accordingly.

Test Cases

1. **Test Case 1**: Torpedo attack on a row.

Input:
torpedo(player, opponent, opponentFleet, "5", false);
Expected Result:
    - Attacks row 5.
    - Reports hits and misses.
    - Updates grids accordingly.
2. **Test Case 2**: Torpedo attack on an invalid column.

Input:
torpedo(player, opponent, opponentFleet, "K", false);
Expected Result:
    - Reports "Invalid column."
    - No action taken.

---

18. checkWin
Specification
Purpose: Checks if all ships in the fleet are sunk.
Prototype:
bool checkWin(Fleet* fleet);
Parameters:

- fleet: Pointer to the fleet to check.

Returns:

- true if all ships are sunk.
- false otherwise.

Behavior:

- Iterates through all ships in the fleet.
- Returns false if any ship is not sunk.

Test Cases

1. **Test Case**: Fleet with all ships sunk.

Input:
    - All ships in fleet have sunk set to true.
Expected Result:
    - checkWin(fleet) returns true.
2. **Test Case**: Fleet with at least one ship not sunk.

Input:
    - At least one ship in fleet has sunk set to false.

Expected Result:
    - checkWin(fleet) returns false.

---

19. updateShipStatus
Specification
Purpose: Updates the sunk status of a ship based on hits.
Prototype:
void updateShipStatus(Ship* ship);
Parameters:
- ship: Pointer to the ship to update.

Behavior:
- Sets ship->sunk to true if ship->hits >= ship->size.

Test Cases
1. **Test Case**: Ship has hits equal to its size.

Input:
ship.size = 3;
ship.hits = 3;
updateShipStatus(&ship);
Expected Result:
    - ship.sunk is true.
2. **Test Case**: Ship has fewer hits than its size.

Input:
ship.size = 4;
ship.hits = 2;
updateShipStatus(&ship);
Expected Result:
    - ship.sunk is false.

---

20. unlockSpecialMoves
Specification
Purpose: Unlocks special moves for the player based on game conditions.
Prototype:
void unlockSpecialMoves(Player* player, Player* opponent);
Parameters:
- player: Pointer to the player unlocking special moves.
- opponent: Pointer to the opponent player.

Behavior:
- Unlocks artilleryAvailable after sinking a ship.
- Unlocks torpedoAvailable when the opponent has only one ship remaining.
- Allows additional smoke screens based on the number of ships sunk.

Test Cases
1. **Test Case**: Player sinks a ship; artillery becomes available.

Input:
player.artilleryAvailable = false;
opponent.shipsRemaining = 3;
unlockSpecialMoves(&player, &opponent);

Expected Result:
    - player.artilleryAvailable is set to true.
2. **Test Case**: Opponent has one ship remaining; torpedo becomes available.

Input:
player.torpedoAvailable = false;
opponent.shipsRemaining = 1;
unlockSpecialMoves(&player, &opponent);
Expected Result:
    - player.torpedoAvailable is set to true.

---

21. displayTrackingGrid
Specification
Purpose: Displays the player's tracking grid, showing the results of their attacks on the opponent.
Prototype:
void displayTrackingGrid(Player* player, bool hardMode);
Parameters:
- player: Pointer to the player whose tracking grid is to be displayed.
- hardMode: Boolean indicating if hard mode is enabled (affects ship visibility).

Behavior:
- Calls displayGrid with the player's tracking grid.
- In hard mode, hides ship symbols; otherwise, shows them.

Test Cases
1. **Test Case 1**: Display tracking grid in normal mode.

Input:
displayTrackingGrid(&player, false);
Expected Result:
    - The tracking grid is displayed with ships visible (if any have been detected).
2. **Test Case 2**: Display tracking grid in hard mode.

Input:
displayTrackingGrid(&player, true);
Expected Result:
    - The tracking grid is displayed with ships hidden, showing '~' instead of ship symbols.

---

22. isValidCommand
Specification
Purpose: Validates if a command is valid and available for the player.
Prototype:
bool isValidCommand(const char* command, Player* player);
Parameters:
- command: String representing the command input by the player.
- player: Pointer to the player attempting to execute the command.

Returns:
- true if the command is valid and available.
- false otherwise.

Behavior:

- Checks if the command is among the allowed commands: "fire", "radar", "smoke", "artillery", "torpedo".
- Verifies if the player has the special move available (e.g., artillery is unlocked).

Test Cases

1. **Test Case 1**: Valid command "fire".

Input:
bool result = isValidCommand("fire", &player);
Expected Result:

- result is true.

2. **Test Case 2**: Command "torpedo" when torpedo is not available.

Input:
player.torpedoAvailable = false;
bool result = isValidCommand("torpedo", &player);
Expected Result:

- result is false.

3. **Test Case 3**: Invalid command "fly".

Input:
bool result = isValidCommand("fly", &player);
Expected Result:

- result is false.

---

23. getInput
Specification
Purpose: Safely gets input from the user, handling buffer overflows and input termination.
Prototype:
void getInput(char* input, int size);
Parameters:

- input: Buffer to store the input.
- size: Size of the input buffer.

Behavior:

- Reads a line from stdin into the input buffer.
- Removes the trailing newline character.
- Flushes the input buffer if the input exceeds the buffer size.

Test Cases

1. **Test Case**: User inputs "fire A1".

Input:
char input[50];
// Simulate user input "fire A1\n"
// (In actual unit testing, this may involve mocking stdin)
getInput(input, sizeof(input));
Expected Result:

- input contains "fire A1".

---

24. coordinateToString
Specification

Purpose: Converts a Coordinate struct to a string representation (e.g., {0,4} -> "A5").
Prototype:
void coordinateToString(Coordinate coord, char* coordStr);
Parameters:

- coord: The coordinate to convert.
- coordStr: Buffer to store the string representation.

Behavior:

- Converts the x index to a column letter (A-J).
- Converts the y index to a row number (1-10).
- Stores the result in coordStr.

Test Cases

1. **Test Case**: Convert coordinate {2, 4}.

Input:
Coordinate coord = {2, 4};
char coordStr[5];
coordinateToString(coord, coordStr);
Expected Result:

- coordStr contains "C5".

2. **Test Case**: Convert coordinate {9, 9}.

Input:
Coordinate coord = {9, 9};
char coordStr[5];
coordinateToString(coord, coordStr);
Expected Result:

- coordStr contains "J10".

---

25. toLowerCase
Specification
Purpose: Converts a string to lowercase.
Prototype:
void toLowerCase(char* str);
Parameters:

- str: The string to convert.

Behavior:

- Converts all uppercase letters in str to lowercase.

Test Cases

1. **Test Case**: Convert "Fire A1".

Input:
char str[] = "Fire A1";
toLowerCase(str);
Expected Result:

- str is "fire a1".

---

26. flushInputBuffer
Specification
Purpose: Flushes the input buffer to remove any extraneous input.

Prototype:
void flushInputBuffer();
Behavior:

- Reads and discards characters from stdin until a newline or EOF is encountered.

Test Cases

1. **Test Case**: Simulate overflowing input and ensure buffer is flushed.

Input:
// User inputs a long string exceeding buffer size
flushInputBuffer();
Expected Result:

- Input buffer is cleared.

---

27. getRandomNumber
Specification
Purpose: Generates a random number between min and max (inclusive).
Prototype:
int getRandomNumber(int min, int max);
Parameters:

- min: Minimum value.
- max: Maximum value.

Returns:

- A random integer between min and max.

Test Cases

1. **Test Case**: Generate random number between 1 and 10.

Input:
int num = getRandomNumber(1, 10);
Expected Result:

- num is an integer between 1 and 10.

---

28. getRandomCoordinate
Specification
Purpose: Generates a random coordinate within the grid.
Prototype:
Coordinate getRandomCoordinate();
Returns:

- A Coordinate with x and y between 0 and GRID_SIZE - 1.

Test Cases

1. **Test Case**: Generate random coordinate.

Input:
Coordinate coord = getRandomCoordinate();
Expected Result:

- coord.x and coord.y are integers between 0 and 9.

---

29. getNextTarget
Specification

Purpose: Selects the next target for the bot by choosing the coordinate with the highest probability based on the probability grid.
Prototype:
Coordinate getNextTarget(Player* bot, Fleet* opponentFleet);
Parameters:

- bot: Pointer to the bot player.
- opponentFleet: Pointer to the opponent's fleet.

Returns:

- A Coordinate representing the next target.

Behavior:

- Calculates a probability grid using calculateProbabilityGrid.
- Selects coordinates with the highest probability that haven't been targeted yet.
- If multiple coordinates have the same highest probability, selects one at random.

Test Cases

1. **Test Case**: Bot selects next target based on probability.

Input:
Coordinate target = getNextTarget(&bot, &opponentFleet);
Expected Result:

- target is a valid coordinate.
- The coordinate corresponds to a cell with the highest calculated probability.

---

30. calculateProbabilityGrid
Specification
Purpose: Calculates a probability grid representing the likelihood of each cell containing a ship.
Prototype:
void calculateProbabilityGrid(Player* bot, Fleet* opponentFleet, int probabilityGrid[GRID_SIZE][GRID_SIZE]);
Parameters:

- bot: Pointer to the bot player.
- opponentFleet: Pointer to the opponent's fleet.
- probabilityGrid: 2D array to store the calculated probabilities.

Behavior:

- Considers the bot's tracking grid and remaining ships in the opponent's fleet.
- Assigns higher probabilities to cells where ships are more likely to be based on remaining ship sizes and previous hits.
- Increases probability for cells adjacent to hits.

Test Cases

1. **Test Case**: Probability grid reflects higher likelihood near hits.

Input:

```
// Assume bot's tracking grid has a hit at (5,5)
bot.trackingGrid[5][5] = '*';
int probabilityGrid[GRID_SIZE][GRID_SIZE];
calculateProbabilityGrid(&bot, &opponentFleet, probabilityGrid);
```

Expected Result:

- Cells adjacent to (5,5) have higher probability values.
- Cells with misses ('o') have zero probability.

---

31. addAdjacentTargets

Specification

Purpose: Adds adjacent tiles to the bot's potential target queue after a successful hit, considering ship orientation.

Prototype:

void addAdjacentTargets(Player* bot, Coordinate coord);

Parameters:

- bot: Pointer to the bot player.
- coord: Coordinate where the hit occurred.

Behavior:

- If aligned hits are found, continues targeting in that direction.
- Adds valid adjacent coordinates to bot->potentialTargets.

Test Cases

1. **Test Case**: Add adjacent targets after a hit.

Input:

```
// Assume bot hit at (5,5)
Coordinate hitCoord = {5, 5};
addAdjacentTargets(&bot, hitCoord);
```

Expected Result:

- Coordinates (5,6), (6,5), (5,4), (4,5) are added to bot->potentialTargets if not already targeted.

2. **Test Case**: Extend search in a specific direction.

Input:

```
// Bot has hits at (5,5) and (5,6)
bot.trackingGrid[5][5] = '*';
bot.trackingGrid[5][6] = '*';
Coordinate hitCoord = {5, 6};
addAdjacentTargets(&bot, hitCoord);
```

Expected Result:

- Only adds coordinates in the same column (vertical direction), e.g., (5,7), to bot->potentialTargets.

---

32. addPotentialTarget

Specification

Purpose: Adds a new potential target to the bot's target queue if it hasn't been added already.

Prototype:

void addPotentialTarget(Player* player, Coordinate coord);

Parameters:

- player: Pointer to the player (bot).
- coord: Coordinate to add to the potential target queue.

Behavior:

- Checks if the coordinate is already in player->potentialTargets.
- Adds it to the queue if it's not already present.

Test Cases

1. **Test Case**: Add a new potential target.

Input:
Coordinate coord = {5, 5};
addPotentialTarget(&bot, coord);
Expected Result:
- coord is added to bot->potentialTargets.

2. **Test Case**: Attempt to add a duplicate potential target.

Input:
addPotentialTarget(&bot, coord); // coord already added
Expected Result:
- coord is not added again.
- bot->potentialTargetCount remains the same.

---

33. getBestArtilleryTarget
Specification
Purpose: Determines the optimal 2x2 area for deploying an artillery strike based on untargeted tiles.
Prototype:
Coordinate getBestArtilleryTarget(Player* bot);
Parameters:
- bot: Pointer to the bot player.

Returns:
- A Coordinate representing the top-left corner of the best artillery target area.

Behavior:
- Scans the grid for 2x2 areas with the highest number of untargeted ('~') tiles.
- Returns the coordinate of the area with the maximum untargeted tiles.

Test Cases
1. **Test Case**: Bot selects artillery target area with maximum untargeted tiles.

Input:
Coordinate target = getBestArtilleryTarget(&bot);
Expected Result:
- target is the coordinate of a 2x2 area with the most untargeted tiles.

---

34. countUntargetedTilesInArtilleryArea
Specification
Purpose: Counts the number of untargeted tiles within a 2x2 artillery strike area.
Prototype:
int countUntargetedTilesInArtilleryArea(Player* bot, Coordinate coord);
Parameters:
- bot: Pointer to the bot player.
- coord: Coordinate representing the top-left corner of the artillery area.

Returns:
- Number of untargeted ('~') tiles in the specified area.

Behavior:
- Adjusts for grid boundaries.
- Counts untargeted tiles within the 2x2 area.

Test Cases

1. **Test Case**: Count untargeted tiles in a given area.

Input:
Coordinate coord = {5, 5};
int count = countUntargetedTilesInArtilleryArea(&bot, coord);
Expected Result:
    - count reflects the number of untargeted tiles in the area starting at (5,5).

---

35. chooseTorpedoTarget
Specification
Purpose: Selects the best row or column to deploy a torpedo based on untargeted tiles.
Prototype:
bool chooseTorpedoTarget(Player* bot, Player* opponent, Fleet* opponentFleet, bool hardMode);
Parameters:
- bot: Pointer to the bot player.
- opponent: Pointer to the opponent player.
- opponentFleet: Pointer to the opponent's fleet.
- hardMode: Boolean indicating if hard mode is enabled.

Returns:
- true if a torpedo was successfully deployed.
- false otherwise.

Behavior:
- Evaluates all rows and columns to find the one with the most untargeted tiles.
- Deploys torpedo attack on the optimal row or column.

Test Cases
1. **Test Case**: Bot selects a row with the most untargeted tiles.

Input:
bool success = chooseTorpedoTarget(&bot, &opponent, &opponentFleet, false);
Expected Result:
    - Torpedo is deployed on the selected row or column.
    - success is true.
2. **Test Case**: No valid torpedo targets available.

Input:
// All rows and columns have been targeted
bool success = chooseTorpedoTarget(&bot, &opponent, &opponentFleet, false);
Expected Result:
    - success is false.

---

36. isUnderSmoke
Specification
Purpose: Checks if a coordinate is under an active smoke screen.
Prototype:
bool isUnderSmoke(Player* opponent, Coordinate coord);
Parameters:
- opponent: Pointer to the opponent player.
- coord: Coordinate to check.

Returns:

- true if the coordinate is under an active smoke screen.
- false otherwise.

Behavior:

- Iterates through the opponent's active smoke screens.
- Determines if coord falls within any smoke screen areas.

Test Cases

1. **Test Case**: Coordinate is under smoke.

Input:

```
// Opponent has an active smoke screen at (5,5)
opponent.smokeScreens[0].active = true;
opponent.smokeScreens[0].coord = {5, 5};
bool result = isUnderSmoke(&opponent, (Coordinate){5, 5});
```

Expected Result:

- result is true.

2. **Test Case**: Coordinate is not under smoke.

Input:

```
bool result = isUnderSmoke(&opponent, (Coordinate){0, 0});
```

Expected Result:

- result is false.

---

37. getSmokeScreenCoordinateForBot

Specification

Purpose: Determines the best coordinate for the bot to deploy a smoke screen.

Prototype:

```
Coordinate getSmokeScreenCoordinateForBot(Player* bot);
```

Parameters:

- bot: Pointer to the bot player.

Returns:

- A Coordinate representing the best smoke screen location.
- { -1, -1 } if no suitable location is found.

Behavior:

- Scans the bot's grid for areas containing ships.
- Prefers areas where multiple ships are present.
- Returns the coordinate for deploying the smoke screen.

Test Cases

1. **Test Case**: Bot selects a coordinate over its ships.

Input:

```
Coordinate coord = getSmokeScreenCoordinateForBot(&bot);
```

Expected Result:

- coord corresponds to a location where the bot has ships.

2. **Test Case**: No ships remaining; bot cannot deploy smoke screen.

Input:

```
// All bot's ships are sunk
Coordinate coord = getSmokeScreenCoordinateForBot(&bot);
```

Expected Result:

- coord is { -1, -1 }.

---

38. handleEdgeCoordinates
Specification
Purpose: Adjusts coordinate start and end values to prevent out-of-bounds access.
Prototype:
void handleEdgeCoordinates(int* start, int* end);
Parameters:
- start: Pointer to the starting index.
- end: Pointer to the ending index.

Behavior:
- Ensures that start is not less than 0.
- Ensures that end does not exceed GRID_SIZE - 1.

Test Cases
1. **Test Case**: Adjust coordinates at grid boundaries.

Input:
int xStart = -1;
int xEnd = 10;
handleEdgeCoordinates(&xStart, &xEnd);
Expected Result:
- xStart is adjusted to 0.
- xEnd is adjusted to GRID_SIZE - 1 (9).

---

39. addArtilleryHitTargets
Specification
Purpose: Adds targets around successful artillery hits to the bot's potential target queue.
Prototype:
void addArtilleryHitTargets(Player* bot, Coordinate coord);
Parameters:
- bot: Pointer to the bot player.
- coord: Coordinate where the artillery hit occurred.

Behavior:
- Adds all untargeted tiles within a 3x3 area centered on coord to the bot's potential targets.
- Resets bot->lastArtilleryHits to zero.

Test Cases
1. **Test Case**: Bot adds potential targets after artillery hits.

Input:
// Assume artillery hit at (5,5)
addArtilleryHitTargets(&bot, (Coordinate){5, 5});
Expected Result:
- All untargeted coordinates within the area are added to bot->potentialTargets.
- bot->lastArtilleryHits is set to 0.