## Game Setup and Initialization Functions:

- **char set_game_difficulty();**
  - Test case 1: invalid input (more than one character)
    Input: QWHS
    Output: "Invalid input. Please enter 'E' or 'H'.
  - Test case 2: invalid input (single character)
    Input: Q
    Output:" Invalid input. Please enter 'E' or 'H'."
  - Test case 3: valid input
    Input: E or e
    Output: accepted
- **void initialize_player(Player* player);**
  - Test Case: Valid partially initialized struct
    Input: a Player struct where Player.name=" Z" & Player.Turn=1
    Output: The function overwrites all fields:
    *Player Initialization:*
    *turn == 0*
    *numOfShipsSunken == 0*
    *numOfArtillery == 0*
    *numOfRadars == 3*
    *numOfSmokeScreensPerformed == 0*
    *numOfTorpedo == 0*
    *Board Initialization:*
    *player->board, player->hits, and player->obscuredArea are fully populated*
    *Ship Array Setup:*
    *Ships have the correct names, sizes, IDs, and occupiedCells initialized to zeros.*

  - Test Case: Valid uninitialized struct
    Input: a player struct with all parameters except name not yet initialized
    Output: *Player Initialization:*
    *turn == 0*
    *numOfShipsSunken == 0*
    *numOfArtillery == 0*
    *numOfRadars == 3*
    *numOfSmokeScreensPerformed == 0*
    *numOfTorpedo == 0*
    *Board Initialization:*
    *player->board, player->hits, and player->obscuredArea are fully populated*
    *Ship Array Setup:*

*Ships have the correct names, sizes, IDs, and occupiedCells initialized to zeros.*

- **void initialize_board(char board[GRID_SIZE][GRID_SIZE]);**
  - Test Case 1: Fresh board initialization
    Input: Valid board array
    Output: All cells set to default state ('~')
  - Test Case 2: Already populated board
    Input: C C C C C ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    B B B B ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    D D D ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    S S ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    Output: Complete overwrite of previous data
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

- **void initializeBotPlayer(Player *bot);**
  - *Test Case 1: Complete bot initialization*
    *Input: A valid Player pointer (bot)*
    *Output: Player Initialization:*
    *turn == 0*
    *numOfShipsSunken == 0*
    *numOfArtillery == 0*
    *numOfRadars == 3*
    *numOfSmokeScreensPerformed == 0*
    *numOfTorpedo == 0*
    *Board Initialization:*
    *bot->board, bot->hits, and bot->obscuredArea are fully populated*

*Ship Array Setup:*

*Ships have the correct names, sizes, IDs, and occupiedCells initialized to zeros.*

*Hunt Queue Initialization:*

*bot->huntQueue is initialized to an empty state*

- **void startGame(Player *currentPlayer, Player *opponent, char game_difficulty);**

  Input:

  human and bot pointers are initialized with all fields correctly set up.

  game_difficulty = 'E' or 'H'.

  Expected Behavior:

  The game loop runs as expected until a winner is determined.

  Both players alternate turns based on the turn field.

  At the end of the game, the correct winner is displayed.

  Boards are printed accurately.

## Board and Display Functions:

- **void displayBoard(Player *player);**
  - Test Case 1: normal board display

    Input:

    C C C C C ~ ~ ~ ~ ~

    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    B B B B ~ ~ ~ ~ ~ ~

    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    D D D ~ ~ ~ ~ ~ ~ ~

    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    S S ~ ~ ~ ~ ~ ~ ~ ~

    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    Output:

    A B C D E F G H I J

    1 C C C C C ~ ~ ~ ~ ~

    2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    3 B B B B ~ ~ ~ ~ ~ ~

    4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    5 D D D ~ ~ ~ ~ ~ ~ ~

    6 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

    7 S S ~ ~ ~ ~ ~ ~ ~ ~

    8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
10 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

- Test Case 2: Empty board
  Input: empty board
  Output:
  A B C D E F G H I J
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10

- **void display_opponent_grid(char board[GRID_SIZE][GRID_SIZE], char game_difficulty);**
  - Test Case 1: when difficulty is E
    Input
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ * ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ o ~*~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    Output:
    A B C D E F G H I J
    1 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    2 ~ * ~ ~ ~ ~ ~ ~ ~ ~
    3 ~ ~ o ~ * ~ ~ ~ ~ ~
    4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    6 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

7 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
10 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

- Test Case 2: when difficulty is H
  Input
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ * ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ o ~*~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  Output:
  A B C D E F G H I J
  1 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  2 ~ * ~ ~ ~ ~ ~ ~ ~ ~
  3 ~ ~ ~ ~ * ~ ~ ~ ~ ~
  4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  6 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  7 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  10 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

## Ship Placement Functions:

- **void placeShips(Player *player);**
  - Test Case: Standard Grid Placement
    Player Input:
    Carrier: User enters starting position A1 and orientation H.
    Battleship: User enters starting position B3 and orientation V.
    Destroyer: User enters starting position D5 and orientation H.
    Submarine: User enters starting position F7 and orientation V.

Output : Console Out:
Place your Carrier (size: 5). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): A1 H
Carrier placed successfully.

Place your Battleship (size: 4). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): B3 V
Battleship placed successfully.

Place your Destroyer (size: 3). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): D5 H
Destroyer placed successfully.

Place your Submarine (size: 2). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): F7 V
Submarine placed successfully.

- Test Case 2: Out of bounds
  Player Input:
  Carrier: User enters the starting position K1
  Console Output: Place your Carrier (size: 5). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): K1 H Invalid coordinates. Please stay within the grid (A1 to J10).
- Test Case 3: invalid orientation
  Player Input:
  Carrier: User enters starting position A1 Z
  Console Output: Place your Carrier (size: 5). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): A1 Z
  Invalid orientation. Use 'H' for horizontal or 'V' for vertical.
- Test Case 6: orientation is valid but would extend outside grid
  Player Input:
  Carrier: User enters starting position A10 V
  Console Output:
  Place your Carrier (size: 5). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): A10 V
  Ship would extend beyond the bottom edge. Try a different position.
- Test Case 5: Overlapping Ships
  Player Input:
  Console Output: Place your Carrier (size: 5). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): A1 H
  Carrier placed successfully.

Place your Battleship (size: 4). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): B3 V
Battleship placed successfully.

Place your Destroyer (size: 3). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): B3 H
Ship placement overlaps with another ship.
Invalid placement. Try again.

Place your Destroyer (size: 3). Enter starting position (e.g., B3) and orientation (H for horizontal, V for vertical): D5 H
Destroyer placed successfully.

- **int checkShipOverlap(Player *player, Ship *ship, int startRow, int startCol, char orientation);**
  - Test Case 1: Valid(No ovelap)
    Input:Player board empty(~), carrier to be placed. startCol 0, startRow 0 and orientation ‘H’
    Output:Returns 1
  - Test Case 2:Ship to be placed overlaps another ship
    Input: Carrier is placed horizontally at A1. Attempting to place Battleship at A3
    Output: Console Output: "Ship placement overlaps with another ship.” and funct returns 0
- **void placeShipOnBoard(Player *player, Ship *ship, int startRow, int startCol, char orientation);**
  - No edge test cases here since all input parameters are checked before being passed to this
  - Valid input test case:
    Input: Player Board: Empty grid initialized with '~'.
    Ship: Carrier (size = 5, id = 'C').
    Starting Position: startRow = 0, startCol = 0 (A1).
    Orientation H
    Output: occupied cells are updated correctly as well as the board
    Console Output:(if and only if player is human)
    A B C D E F G H I J
    1 C C C C C ~ ~ ~ ~ ~
    2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    3 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    4 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
    5 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

6 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
7 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
8 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
9 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
10 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

## Player and Move Management Functions:

- **void selectMove(Player *attacker, Player *defender, char game_difficulty);**
  - Test Case 1: Human Player Standard Move
    Input:
    Move Type: "Fire"
    Coordinate: "A1"
    Output: The output indicates the player's move and show the updated grid (hits/misses).
    The message "Player's turn" should be displayed.
    If the move is valid, the grid should be updated with the hit or miss.
    The turn is switched to the defender after the valid move.
  - Test Case 2: Invalid Move Selection
    Input:
    Move Type: "Artillery"
    Coordinate: "Z1" (Invalid column)
    Output: The output indicates that the coordinate is invalid
    The move is rejected and re-prompted without changing the game state.
    No change is made to the grid, and the player's turn continues.
  - Test Case 3: Invalid Move Format
    Input:Move Type: "Fire"
    Coordinate: "A" (Missing row number)
    Output:
    The move is rejected, and the player is asked to enter the move again.
  - Test Case 4: Move Exceeds Input Length
    Input: Move Type: "Fire"
    Coordinate: "A1"
    Additional Text: "Extra text after valid input" (Input exceeds allowed length)
    Output:
    The system should print a message like: "Input too long. Use format 'MoveType Coordinate'".
    The input is rejected, and the prompt is displayed again.
  - Test Case 5: Out of Bound Coordinate
    Input: Move Type: "Fire"

Coordinate: "K1" (Invalid column outside the 10x10 grid)
Output:
The system should print a message like: "Coordinates out of bounds."
The move is rejected, and the player is prompted for a new valid move.

- **void selectBotCoordinate(Player *bot, Player *opponent, int *x, int *y, char moveType);**
    - Test Case 1: Torpedo move with hunt queue
      Input:
      bot->huntQueue = [/*some hunt coordinates*/]
      moveType = 'T'
      Output:
      The coordinates are dequeued from the hunt queue and assigned to x and y.
      Either x = 0 or y = 0 is set randomly (row/column).
    - Test Case 2: Torpedo move with no hunt queue
      Input:
      bot->huntQueue = []
      moveType = 'T'
      Random unexplored cells available on the grid.
      Output:
      The bot randomly selects a row or column, then targets the row or column with the highest number of unexplored cells.
    - Test Case 3: Artillery move with hunt queue
      Input:
      bot->huntQueue = [/*some hunt coordinates*/]
      moveType = 'A'
      Output:
      The coordinates are dequeued from the hunt queue and assigned to x and y.
    - Test Case 4: Artillery move with no hunt queue
      Input:
      bot->huntQueue = []
      moveType = 'A'
      A grid with some unexplored cells.
      Output:
      The bot searches for a 2x2 area with the highest unexplored cells.
      If no dense areas are found, the bot randomly selects an unexplored 2x2 area.
    - Test Case 5: Radar move
      Input:
      moveType = 'R'
      A grid with unexplored cells and varying density.
      Output:
      The bot selects the 2x2 area with the highest unexplored density and assigns those coordinates to x and y.

- Test Case 6: Smoke screen move
  Input:
  moveType = 'S'
  Bot has vulnerable regions.
  Output:
  The bot calls findVulnerableRegions to determine where to place the smoke screen. If no vulnerable regions are found, the bot calls findDenseClusterOrRandom to select a random or dense area for the smoke screen.
- Test Case 8: Fire move with hunt queue
  Input:
  bot->huntQueue = [/*some hunt coordinates*/]
  moveType = 'F'
  Output:
  The coordinates are dequeued from the hunt queue and assigned to x and y.
- Test Case 9: Fire move with no hunt queue
  Input:
  bot->huntQueue = []
  moveType = 'F'
  Random unexplored cells available on the grid.
  Output:
  The bot randomly selects an unexplored cell on the grid.

- **char selectBotMoveType(Player *bot);**
  - Test Case 1: Torpedo available
    Input:
    bot->numOfTorpedo = 1
    bot->numOfArtillery = 0
    bot->numOfRadars = 0
    bot->numOfShipsSunken = 0
    bot->numOfSmokeScreensPerformed = 0
    Output:
    "BOT: Torpedo unlocked! Preparing torpedo attack..."
    Move: 'T' (Torpedo).
  - Test Case 2: Artillery available
    Input:
    bot->numOfTorpedo = 0
    bot->numOfArtillery = 1
    bot->numOfRadars = 0
    bot->numOfShipsSunken = 0
    bot->numOfSmokeScreensPerformed = 0

Output:
"BOT: Artillery unlocked! Preparing artillery strike..."
Move: 'A' (Artillery).

- Test Case 3: Radar available and unexplored percentage exceeds threshold
  Input:
  bot->numOfTorpedo = 0
  bot->numOfArtillery = 0
  bot->numOfRadars = 1
  bot->numOfShipsSunken = 0
  bot->numOfSmokeScreensPerformed = 0
  calculateUnexploredPercentage(bot, opponent) = 0.8
  calculateRadarThreshold(bot, opponent) = 0.5
  Output:
  "BOT: Radar available. Scanning..."
  Move: 'R' (Radar).
- Test Case 4: Radar available with random factor
  Input:
  bot->numOfTorpedo = 0
  bot->numOfArtillery = 0
  bot->numOfRadars = 1
  bot->numOfShipsSunken = 0
  bot->numOfSmokeScreensPerformed = 0
  calculateUnexploredPercentage(bot, opponent) = 0.3
  calculateRadarThreshold(bot, opponent) = 0.5
  Random factor: 0.2
  Output:
  "BOT: Radar available. Scanning..."
  Move: 'R' (Radar).
- Test Case 5: Vulnerability threshold exceeded, smoke screen
  Input:
  bot->numOfTorpedo = 0
  bot->numOfArtillery = 0
  bot->numOfRadars = 0
  bot->numOfShipsSunken = 2
  bot->numOfSmokeScreensPerformed = 1
  calculateVulnerabilityScore(bot) = 6
  Output:
  "BOT: Let me obscure a vulnerable area..."
  Move: 'S' (Smoke Screen).
- Test Case 6: No special moves, default to fire

Input:
bot->numOfTorpedo = 0
bot->numOfArtillery = 0
bot->numOfRadars = 0
bot->numOfShipsSunken = 0
bot->numOfSmokeScreensPerformed = 0
Output:
"BOT: Performing a standard fire attack..."
Move: 'F' (Fire).

- **void botSelectMove(Player *bot, Player *human, char game_difficulty);**
  **this move heavily depends on the two calls at its start that provide the movetype and where to target**
    - Test Case 1: Fire move (Standard Attack)
      Output: FireMove function is called with appropriate parameters (bot, human, x, y, and game_difficulty).The game state is updated based on the attack, including checking whether a ship was hit or sunk.

    - Test Case 2: Artillery move with ship sunk
      Output: ArtilleryMove is called.
      The bot checks if a ship was sunk, and if so, keeps Artillery available for the next turn.bot->numOfArtillery remains 1.
    - Test Case 3: Artillery move with no ship sunk
      Output:ArtilleryMove is called.
      The bot disables Artillery for the next turn (bot->numOfArtillery = 0).
    - Test Case 4: Torpedo move
      Output:TorpedoMove is called with appropriate parameters (bot, human, x, y, and game_difficulty).
      bot->numOfTorpedo is set to 0.
    - Test Case 5: Radar move
      Output:RadarMove is called.
      bot->numOfRadars is decremented by 1.
    - Test Case 6: Smoke move
      Output:SmokeMove is called with appropriate parameters (bot, x, y).
      bot->numOfSmokeScreensPerformed is incremented by 1.
    - Test Case 7: Bot with no available special moves
      Output:FireMove is called, as no special moves are available. The bot defaults to performing a Fire move.

## Move Execution Functions:

- **void FireMove(Player* attacker, Player* defender, int x, int y, char game_difficulty);**
  - Test Case 1: Hit on Ship
    Input:
    x = 3, y = 4
    The defender's grid has a ship at this coordinate.
    Expected Output:
    The hit is registered on the defender's grid, with an * symbol.all relevent resources are updated
    The message "Hit!"should be displayed.
    The specific ship's occupied cell should be updated with a 1 (indicating a hit).
    If the ship sinks, the relevant ship name is printed
  - Test Case 2: Miss on Water
    Input:
    x = 6, y = 7
    The defender's grid has no ship at this coordinate.
    Expected Output:
    The defender's grid shows an o at the coordinates x = 6, y = 7.
    The message "Miss!"should be displayed.
- **void ArtilleryMove(Player* attacker, Player* defender, int x, int y, char game_difficulty);**
  - Test Case 1: Artillery Move - Hit on 2x2 Area
    Input: x = 3, y = 4
    The defender's grid has a ship occupying part of this 2x2 area.
    Output:
    The affected ship's occupied cells are updated with a hit.
    If any ship sinks, a message is displayed.
    The attacker's numOfArtillery remains 1 if a ship is sunk, or is reset to 0 otherwise.
  - Test Case 2: Artillery Move - No Ships in Area (Miss)
    Input:
    x = 2, y = 3 (Top-left corner of a 2x2 area with no ships).
    The defender's grid has no ships in this area.
    Expected Output:Each of the 4 cells is marked with a miss ('o').
    The attacker's numOfArtillery is set to 0.
- **void TorpedoMove(Player* attacker, Player* defender, int x, int y, char game_difficulty);**
  - Test Case 1: Torpedo Move - Row Attack (Hits Ship)
    Input:
    x = 0, y = 2 (Coordinates indicating the row for the torpedo attack).
    A ship is located somewhere along the row at the coordinate y = 2.
    Output:

The ship’s occupied cells in the row are updated with hits ('*').
If the ship sinks, the message is displayed.
The attacker’s numOfTorpedo is set to 0.

- Test Case 4: Torpedo Move - Column Attack (Miss)
  Input:
  x = 0, y = 5 (Coordinates indicating the column for the torpedo attack).
  The defender’s grid has no ship in the column at y = 5.
  Output:
  Each cell in the column is marked with a miss ('o').
  The attacker’s numOfTorpedo is set to 0.

**void RadarMove(Player *attacker, Player *defender, int x, int y);**

- Test Case 1: Ships Found in 2x2 Area (Unobscured)
  Input:
  x = 2, y = 3 (coordinates of the top-left corner of a 2x2 area).
  attacker = bot, defender = human (example players).
  defender->board[x+0][y+0] = 'C', defender->board[x+0][y+1] = '~', defender->board[x+1][y+0] = '~', defender->board[x+1][y+1] = 'D' (two ships present).
  defender->obscuredArea[x+0][y+0] = ' ', defender->obscuredArea[x+0][y+1] = ' ', defender->obscuredArea[x+1][y+0] = ' ', defender->obscuredArea[x+1][y+1] = ' ' (not obscured).
  Output:
  "Enemy ships found."Test Case 2: No Ships in 2x2 Area
- Input:
  x = 5, y = 5 (coordinates of a 2x2 area with no ships).
  attacker = bot, defender = human (example players).
  defender->board[x+0][y+0] = '~', defender->board[x+0][y+1] = '~', defender->board[x+1][y+0] = '~', defender->board[x+1][y+1] = '~' (no ships).
  defender->obscuredArea[x+0][y+0] = ' ', defender->obscuredArea[x+0][y+1] = ' ', defender->obscuredArea[x+1][y+0] = ' ', defender->obscuredArea[x+1][y+1] = ' ' (not obscured).
  Output:
  "No enemy ships found."
- Test Case 3: Ship Present but Obscured
  Input:
  x = 1, y = 1 (coordinates of the top-left corner of a 2x2 area).
  attacker = bot, defender = human (example players).
  defender->board[x+0][y+0] = 'B', defender->board[x+0][y+1] = '~', defender->board[x+1][y+0] = '~', defender->board[x+1][y+1] = 'S' (ships are present).

defender->obscuredArea[x+0][y+0] = 'S', defender->obscuredArea[x+0][y+1] = ' ', defender->obscuredArea[x+1][y+0] = ' ', defender->obscuredArea[x+1][y+1] = ' ' (one ship is obscured).
Output:
"No enemy ships found."

- Test Case 4: All Ships in 2x2 Area Obscured
  Input:
  x = 6, y = 6 (coordinates of the top-left corner of a 2x2 area).
  attacker = bot, defender = human (example players).
  defender->board[x+0][y+0] = 'C', defender->board[x+0][y+1] = 'D', defender->board[x+1][y+0] = 'S', defender->board[x+1][y+1] = 'B' (all ships are present).
  defender->obscuredArea[x+0][y+0] = 'S', defender->obscuredArea[x+0][y+1] = 'S', defender->obscuredArea[x+1][y+0] = 'S', defender->obscuredArea[x+1][y+1] = 'S' (all ships are obscured).
  Output:
  "No enemy ships found."
- Test Case 5: Edge of the Grid (Radar Sweep Near Grid Boundaries)
  Input:
  x = 8, y = 8 (coordinates at the edge of the grid).
  attacker = bot, defender = human (example players).
  defender->board[x+0][y+0] = 'C', defender->board[x+0][y+1] = 'S', defender->board[x+1][y+0] = '~', defender->board[x+1][y+1] = 'B' (some ships are near the edge).
  defender->obscuredArea[x+0][y+0] = ' ', defender->obscuredArea[x+0][y+1] = ' ', defender->obscuredArea[x+1][y+0] = ' ', defender->obscuredArea[x+1][y+1] = ' '.
  Expected Output:
  "Enemy ships found."

- **void SmokeMove(Player *attacker, int x, int y);**
  - Test Case 1: Attempt to obscure a valid 2x2 area within the grid:
    Input: x = 4, y = 4.
    Expected Output: "Obscured successfully!"
- **void markAffectedArea(int x, int y, char moveType, char orientation);**
  - Test Case 1: Single Cell Hit (Fire move)
    Input:
    x = 3, y = 4 (coordinates of the ship to be hit).
    moveType = 'F' (Fire move).
    orientation = 'H' (irrelevant for Fire move).
    Output:
    affectedArea[3][4] = 'X', all other cells remain '~'.
  - Test Case 2: 2x2 Area Hit (Artillery move)
    Input:

x = 2, y = 3 (top-left corner of the affected 2x2 area).
moveType = 'A' (Artillery move).
orientation = 'H' (irrelevant for Artillery move, as it affects a fixed 2x2 area).
Output:
affectedArea[2][3] = 'X', affectedArea[2][4] = 'X', affectedArea[3][3] = 'X', affectedArea[3][4] = 'X'.

- Test Case 3: Torpedo
  Input:
  x = 3, y = 0 (row move starting from column 0).
  moveType = 'T' (Torpedo move).
  orientation = 'H' (Horizontal orientation).
  Expected Output:
  The affected area should mark the entire row at index 3 as affected with 'X'.
- Test Case 4: Column-based 2D Area (Torpedo move, Vertical)
  Input:
  x = 2, y = 5 (starting at column 5).
  moveType = 'T' (Torpedo move).
  orientation = 'V' (Vertical orientation).
  Expected Output:
  The affected area should mark the entire column at index 5 as affected with 'X'.

- **void HitOrMiss(Player *attacker, Player *defender, int x, int y, char movetype, char orientation, char game_difficulty);**
  - Test Case 1: Valid Hit (Ship present)
    Input:
    x = 3, y = 4 (coordinates with a ship present).
    movetype = 'F' (Fire move).
    orientation = 'H' (horizontal orientation, but this doesn't affect Fire move).
    The defender has a ship at position (3, 4).
    Output:
    The cell (3, 4) is marked as hit in the hits array (defender->hits[3][4] = '*').
    The ship's occupied cell is marked as hit in occupiedCells.
    The HitRegister is incremented.
    If Player is the BOT, adds to queue the adjacent unexplored cells.
  - Test Case 2: Valid Miss (Water cell)
  - Input:
    x = 6, y = 2 (coordinates without a ship).
    movetype = 'F' (Fire move).
    orientation = 'V' (vertical orientation, but irrelevant for a miss).
    The defender has no ship at position (6, 2).
    Output:
    The cell (6, 2) is marked as a miss in the hits array (defender->hits[6][2] = 'o').

The affected area for this move is updated.

- Test Case 3: Multiple Hits on Different Ships
  Input:
  Multiple affected cells containing ships.
  movetype = 'A' (Artillery move, affects a 2x2 area).
  The affected area covers cells that are occupied by different ships.
  Output:
  The corresponding hits array cells are marked as hit for all ships hit in the affected area.
  The HitRegister is incremented by the number of ships hit.
  The occupiedCells for each ship are updated as hit.
  If a ship is sunk, the message "Ship [name] has been sunk!" is displayed.
  If the Player is BOT, adds adjacent unexplored cells of each cell that was hit
- Test Case 4: Bot's Adjacent Cells Updated after a Hit
  Input:
  attacker = bot, defender = human.
  x = 4, y = 5 (coordinates with a ship hit).
  movetype = 'T' (Torpedo move).
  The bot hits a ship at (4, 5).
  Output:
  The hits array for the defender is updated (defender->hits[4][5] = '*').
  The bot updates adjacent unexplored cells after a hit using addAdjacentUnexploredCells.
  The affected area is marked accordingly.
- Test Case 5: Ship Sunk
  Output:
  A message is displayed: "[Ship Name] has been sunk!"
  The numOfShipsSunken of the defender is incremented.
  If 3 ships are sunk, the attacker->numOfTorpedo is set to 1.
  If a ship is sunk, the attacker->numOfArtillery is set to 1.
- Test Case 7: Multiple Ships Hit but None Sunk
  Output:
  The hits array for each affected cell is marked as hit.
  No ship is sunk, so no message about ships being sunk is displayed.
  No special move (like Torpedo or Artillery) is unlocked for the attacker.

## Utility and Validation Functions:

- **void playerswitch(Player *attacker, Player *defender);**
  Test Case: Input valid attacker ptr and valid defender ptr.
  Attacker's turn is 1 and defender's turn is 0
  Output: defender's turn =1 Attacker's turn=0

Test Case: (bad turn handling from other functs)
Input valid attacker ptr and valid defender ptr.
Attacker's turn is 1 and defender's turn is 1
Output: defender's turn =0 Attacker's turn=0

- **int is_fire(char* moveType);**
  - Test Case 1:Valid Move Inputs
    Input:FIRE or fire or Fire or FiRe..
    Output: returns 1
  - Test Case 2: Invalid Move Inputs
    Input: f1re or flame or firee or abcde
    Output returns 0
- **int is_artillery(char* moveType);**
  Same test cases as is_fire but with different variations of the word Artillery
- **int is_torpedo(char* moveType);**
  Same test cases as is_fire but with different variations of the word Torpedo
- **int is_radar(char* moveType);**
  Same test cases as is_fire but with different variations of the word Radar
- **int is_smoke(char* moveType);**
  Same test cases as is_fire but with different variations of the word Smoke
- **int is_equal(char* str1, char* str2);**
  - Test Case 1:Strings are equal(Case insensitive)
    Input: str1=fire str2=FIRe
    Output returns 1
  - Test Case 2: Strings are not equal
    Input str1=fire str2=FLAME
    Output returns 0

- **void displayAvailableMoves();**
  - No test cases
- **void clear_screen();**
  - No Test cases
- **int column_to_index(char column)**
  - Test Case: Valid Input(Uppercase and lowercase inputs behave the same)//
    Input: 'A' or 'a'
    Output: returns 0
  - Test Case: invalid input(numerical values symbols etc)
    Input: 1
    Output: returns –1
- **int isShipSunk(Ship *ship);**

- Test Case 1:Ship is intact
  Input: ship->size = 4
  ship->occupiedCells = { {0, 0, 0}, {0, 1, 0}, {0, 2, 0}, {0, 3, 0} }
  Output: returns 0
- Test Case 2:Ship is partially sunk
  Input: ship->size = 4
  ship->occupiedCells = { {0, 0, 1}, {0, 1, 0}, {0, 2, 1}, {0, 3, 0} }
  Output: returns 0
- Test Case 3:Ship is completely sunk
  Input: ship->size = 4
- ship->occupiedCells = { {0, 0, 1}, {0, 1, 1}, {0, 2,1}, {0, 3, 1} }
  Output: returns 1

- **void HitOrMissMessageDisplay(int movesuccess);**
  - Test Case 1: movesuccess is 1
    Output: "Hit!"
  - Test Case 2:movesuccess is 0
    Output:"Miss!"
  - Test Case 3: movesuccess is neither
    Output: Nothing happens
- **int isBot(Player *player);**
  - Test case 1:Player is a human
    Input: Player whose name is "Bob"
    Output: returns 0
  - Test case 2: Player is the bot
    Input; Player whose name is "BOT"(exclusive case insensitive name reserved for the bot)
    Output: returns 1

## BOT Strategy Functions:

- **void addAdjacentUnexploredCells(Player *bot, Player *opponent, int x, int y);**
  - Test Case 1: Hit cell adjacent cell tracking
    Input:
    bot->huntQueue is empty, opponent->hits grid has opponent->hits[5][5] = '*', all adjacent cells are unexplored (~), x = 5, y = 5.
    Output:
    bot->huntQueue contains [(4, 5), (6, 5), (5, 4), (5, 6)].
  - Test Case 2: Grid boundary adjacent cells
    Input:
    bot->huntQueue is empty, opponent->hits[0][0] = '*', opponent->hits[0][1] = '~', opponent->hits[1][0] = '~', other cells out of bounds, x = 0, y = 0.

Output:
bot->huntQueue contains [(0, 1), (1, 0)].

- Test Case 3: Already queued cells
  Input:
  bot->huntQueue already contains [(4, 5)], opponent->hits[5][5] = '*', opponent->hits[4][5] = '~', opponent->hits[6][5] = '~', other adjacent cells also unexplored, x = 5, y = 5.
  Output:
  bot->huntQueue contains [(4, 5), (6, 5), (5, 4), (5, 6)].
- Test Case 4: Explored adjacent cells
  Input:
  bot->huntQueue is empty, opponent->hits[5][5] = '*', opponent->hits[5][6] = '*', opponent->hits[6][5] = '~', other adjacent cells unexplored, x = 5, y = 5.
  Output:
  bot->huntQueue contains [(6, 5), (5, 4)].

- **void findVulnerableRegions(Player *bot, int *bestX, int *bestY);**
  - Test Case 1: Partially damaged ship areas
    Input:
    bot->ships[0] (Carrier) has occupiedCells = {{2, 2, 1}, {2, 3, 1}, {2, 4, 0}, {2, 5, 0}, {2, 6, 0}}
    Other ships are fully intact or undamaged.
    calculateProtectionScore for (2, 4) is 10, for (2, 5) is 12, and for (2, 6) is 8.
    Output:
    bestX = 2, bestY = 5.
  - Test Case 3: Completely sunk ships
    Input:
    bot->ships[1] (Battleship) has all occupiedCells[i][2] = 1 (fully sunk).
    Remaining ships are intact, and no other cells are hit.
    Output:
    No vulnerable region identified; bestX and bestY remain unmodified.
- **void findDenseClusterOrRandom(Player *bot, int *bestX, int *bestY);**
  - Test Case 1: Dense ship placement scenario
    Input:
    bot->hits grid is all ~.
    bot->board has ships occupying positions [(3, 3), (3, 4), (4, 3), (4, 4)] forming a dense cluster.
    bot->obscuredArea is all ~.
    Output:
    bestX = 3, bestY = 3.
  - Test Case 2: Sparsely populated grid

Input:
bot->hits grid is all ~.
bot->board has ships at [(1, 1), (7, 7)], far apart.
bot->obscuredArea[1][1] = 'S'.
Output:
Random coordinates selected from unhit and non-obscured positions

- **float calculateUnexploredPercentage(Player *bot);**
  - Test Case 1: Early game exploration
    Input:
    opponent->hits has all cells as ~ (no exploration).
    Output:
    unexploredPercentage = 1.0 (100%).
  - Test Case 2: Late game exploration
    Input:
    opponent->hits has only 20 cells as ~.
    Output:
    unexploredPercentage = 0.2 (20%)
  - Test Case 3: Halfway explored grid
    Input:
    opponent->hits has only 50 cells as ~.
    Output:
    unexploredPercentage = 0.5 (50%).
- **int calculateVulnerabilityScore(Player *bot);**
  - Test Case 1: Heavily damaged ships
    Input:
    Carrier has 4 hit cells
    Submarine has 1 hit cell
    Battleship has 3 hit cells
    Destroyer has 2 hit cells
    Output: high vulnerability score

  - Test Case 2: Intact ship configuration
    Input: All ships have no hits on any of their cells
    Output:
    vulnerabilityScore = 0 (no hits on any ships).
- **int calculateProtectionScore(Player *bot, int x, int y);**
  - Test Case 1: well protected grid area
    Input:
    x = 3, y = 3.
    bot->ships[0].occupiedCells = {{3, 3, 0}, {3, 4, 0}, {4, 3, 0}, {4, 4, 0}}.

bot->obscuredArea[3][3] = 'S', bot->obscuredArea[3][4] = 'S', bot->obscuredArea[4][3] = 'S', bot->obscuredArea[4][4] = 'S'
Output:
score = 0

- Test Case 2: hit ships in area
  Input:
  x = 1, y = 1.
  bot->ships[0].occupiedCells = {{1, 1, 1}, {1, 2, 1}}.
  bot->obscuredArea is all ~ (no obscured areas).
  Output:
  score = 0 (no unhit ships in the 2x2 area).
- Test Case 3: Partially obscured grid section
  Input:
  x = 2, y = 2.
  bot->ships[0].occupiedCells = {{2, 2, 0}, {2, 3, 0}, {3, 2, 0}, {3, 3, 0}}.
  bot->obscuredArea[2][2] = 'S', bot->obscuredArea[2][3] = '~', bot->obscuredArea[3][2] = '~', bot->obscuredArea[3][3] = 'S'.
  Output:
  score = 2 (4 unhit ship cells minus 2 for obscured areas).

- **float calculateRadarThreshold(Player *bot);**
  - Test Case 1: Early game radar consideration
    Input:
    bot->numOfShipsSunken = 0 (no ships sunk).
    Output:
    radarThreshold = 0.5 (base threshold).
  - Test Case 2: Late game radar strategy
    Input:
    bot->numOfShipsSunken = 3 (two ships remaining).
    Output:
    radarThreshold = 0.6 (base threshold increased due to few remaining ships).
  - Test Case 3: Endgame scenario
    Input:
    bot->numOfShipsSunken = 4 (one ship remaining).
    Output:
    radarThreshold = 0. 6

## Utility String Functions:

- **void stringcopy(char* dest, char* src);**
  - Test Case 1:Non empty src with uppercase and lowercase letters
    Input: src = Hello
    Output: dest = Hello

- Test Case 2: src contains special symbols and numbers
  Input: src = Hello1!@#
  Output: dest = Hello1!@#
- Test Case 3: single character
  Input: src = H
  Output: dest = H
- Test Case 4: contains whitespaces and sentences
  Input: src = hello my name is bob
  Output: dest = hello my name is bob

- **void to_lowercase(char* src, char* dest);**
  - Test Case 1: A mix of upper and lowercase source
    Input:src = MiXeDStRiNG
    Output: dest= mixedstring
  - Test Case 2: lowercase-only source
    Input: src = mixedstring
    Output: dest= mixedstring
  - Test Case 3: uppercase-only source
    Input: src = MIXEDSTRING
    Output: dest =mixedstring// *Effects: Converts source string to lowercase in destination*
  - Test Case 4: the empty string
    Input: src=""
    Output: dest = ""
  - Test Case 5: source contains symbols, numbers and/or whitespaces
    Input: src ="!1@2#3 MIXed"
    Output: dest ="!1@2#3 mixed"

## Hunt Queue Management Functions:

- **void initHuntQueue(HuntQueue *queue);**
  Setup:
  Declare a HuntQueue instance.
  Call initHuntQueue(&queue);
  Expected Output:
  The queue should be initialized with:
  front = 0 ;rear = -1 ;size = 0
- **int isHuntQueueEmpty(HuntQueue *queue);**
  - Test Case 1: Queue is empty
    Input queue of size 0
    Output returns 1

    - Test case 2: Queue is not empty
      Input queue of size 10
      Output returns 0
- **void enqueueHunt(HuntQueue *queue, int x, int y);**
    - Test Case 1: Empty queue, adding an element
      Input:
      Add coordinate pair (x = 5, y = 10)
      Output:
      The queue should have the following state:
      front = 0
      rear = 0
      size = 1
      x[0] = 5
      y[0] = 10
    - Test Case 2: Standard queue insertion:
      Input: add coordinate pair (x=0 y=0) queue has one pair
      Output: coordinates are added to the end of x and y array
      The queue should have the following state:
      front = 0
      rear = 1
      size = 2
      x[1] = 0
      y[1] = 0
    - Test Case 3: Queue near full capacity
      Input: coordinate pair (x=0, y=0) almost maximum/full queue size
      Output: the function checks to see if queue is full and exits
- **void dequeueHunt(HuntQueue *queue, int *x, int *y);**
    - Test Case 1: Single element in queue dequeue
      Input:ptrs recipients of the dequeuing process x and y and a queue of size 1 and pair (x=0 y=0)
      Output: The returned values should be:
      x = 0, y = 0
      The queue state should be:
      front = 0
      rear = -1
      size = 0
    - Test Case 2: Standard element removal
      Input: ptrs to recipients of the dequeuing process
      Output: dequeues the coordinates at the front of the queue successfully removing them and sets x and y as those coordinates
    - Test Case 3: Empty queue dequeue attempt

Input: Queue with no elements
Output: if statement checks if its empty and exits. ie nothing happens

- **int isHuntQueueFull(HuntQueue *queue);**
  - Test Case 1: Queue has one element
    Input: queue of size 1
    Output returns 0
  - Test Case 2: Queue is empty
    Input: queue of size 0
    Output returns 0
  - Test Case 3: Queue has multiple pairs but not full
    Input: queue of size 9
    Output returns 0
  - Test Case 4: Queue is full
    Input: queue of size 100
    Output returns 1