Project Documentation

System Overview:

We will be implementing Project 1 – Battleship – using C++ in a command line interface application. Using the object-oriented design paradigm, we will be implementing a handful of classes for modularity. These will include the Executive, Game, Board, and Ship classes.

Interface

- All interfaces should be intuitively built such that the user can determine what to input at each prompt
- Type and value validation are included at each input point such that invalid input will not be accepted, and the user will be handed feedback accordingly
- Lines/blocks signify separation in the game flow that allow for easier reading and tracking of the game status
- Ships can be oriented vertically or horizontally and cannot be moved after being placed
- For the user's own board, an 's' signifies a ship placement
- For the user's enemy board, an asterisk signifies a hit on an opponent's ship and an 'M' signifies a missed fire attempt.
- Both the enemy and player's own board will be printed after each fire
- The game will end once all ships are sunk by either opponent

Class Descriptions:

- Executive:
    - The Executive class contains only a single significant method, Executive::play(), that instantiates the game class, welcomes the user, and continuously calls the fire method of game until a player is declared the winner.
    - Both the constructor and destructor are empty

- Game:
    - The Game class handles all necessary game logic including placing ships, firing, and checking if the game is completed. This class instantiates both the Board and Ship class to implement the classic Battleship game.
    - Game::shipPlacement() retrieves the number of ships the users wish to use for their game and places them on each of the boards. An array of ship object is created for each player as well two boards – one for the player's own board and one for the opposing player. Since the requirements of this project include coordinates of the **char** type, arithmetic is done on the inputted characters' ASCII values for validation.

    - Game::fire() performs a fire given the player's name who is completing the fire. Again, arithmetic on the ASCII values of the inputted characters is required before calling the Ship::hit() method. Once validation is completed, the firing coordinate is passed iteratively to each ship in the opposing player's board. Since the Ship::hit() method will throw an exception if a miss is detected, we utilized a counter that is compared against the number of ships. If these values are equal, we place a M on the "enemy" board and if they are unequal a "*" signifies a hit. The firing player's boards are printed after each fire, regardless of their success.
    - Game::gameEndCheck() utilizes the Ship::is_sunk() method to iteratively check both players ships. If either player has a surviving ship, their iteration is broken. Returns logical OR product of each player's sunk status.
    - Game::player1Won() returns if player 1 is the winner. This method is only called in Game::gameEndCheck() returns true, and is used to determine the winner
    - The Game constructor is empty and the destructor deletes both player's ship arrays.

- Board:
    - The Board class handles all necessary operations concerning the game board and is represented by a 10x10 grid with vertical axis (1-10) and horizontal axis (A-J). This includes placing ships, updating hits and misses, and printing. This class is instantiated by the Game class.
    - The board constructor creates the empty 10x10 game board 2D array. The destructor deletes it
    - Board::printBoard() prints the board's current status including the axis labels.
    - Board::updateBoard() takes a character entry and the coordinates to place the single entry. It then updates the 2D array given the coordinates. Again, ASCII arithmetic is used.
    - Board::placeShip() takes a Ship object by reference to place on the board. It's orientation is first recorded, then is iterated over its size and accesses the Ship array to store to the board array.
    - Board::isValidSpace() is a helper method that takes a coordinate to validate if ship can be placed

- Ship:
    - The Ship class contains the necessary values and functionality resembling a single ship. This includes its size, direction, starting coordinates, and hit and sunk status. This class is instantiated by the Game class
    - The constructor takes the ship's size, direction, and starting coordinates. Each of these values are validated and the ship array is constructed. The destructor deletes the ship array.
    - Ship::hit() takes a firing coordinate and updates the ship array if it is a hit. A runtime_error is thrown if it is a miss and is utilized in the Game class to check if a fire hits any of a player's ships. A "*" is added to the ship array if it is a hit.

    - Ship::is_sunk() iterates through the ship's array and breaks if an 's' is seen. Returns a Boolean value signifying its sunk status
    - A respective get method is available for each member variable: ship (array of chars), size (int), starting vertical coordinate (int), starting horizontal coordinate (char), and direction (char). It should be noted that Ship::get_ship() returns a pointer to the ship array as C++ cannot return an array by value.

Build System

- A makefile is included that handles all compilation and linkage. An executable "Battleship" is generated to play a game
- Command "make" builds the executable
- Command "make clean" removes the executable and all object files

Included libraries

- Iostream (C++ standard library)
- String (C++ standard library)
- Stdexcept (C++ standard library)