**Report: Programming Project 4**

For this battleship project, the board is represented by a 2D array of characters with the rows and columns going up to the max number of each given. There is also a struct ShipInfo which stores every ship’s info that is present on the board: their id, the numOfHits that have been made on them and their symbol. The ships added to the board are stored in a vector of ints by their shipId. There is also a vector of the type ShipInfo that stores all the ships on the board and an int to to keep track of how many ships have been destroyed.

On the other hand, for the Game Class, we had to create a separate struct to store the ship information. This struct has variables for the ship length, ship symbol, and ship name. We also create a vector to store all the ships added to the game. This vector and the ship struct help us develop functions for the game class.

***BoardImpl Class***

**Private Variables Defined**

A reference to game: const Game& m\_game

A char type 2d array for the board: char m\_board[MAXROWS][MAXCOLS];

Struct ShipInfo to store each ship’s id, the number of hits it has had, and it’s symbol

A vector of type ShipInfo to store all the ships on the board

Variable to store the number of ships destroyed: int numOfShipsDestroyed

**BoardImpl::BoardImpl(const Game& g)**

The constructor sets up an empty board by traversing through the whole board using two for loops(one for the columns and one for the rows), and sets every element of the board to ‘.’. We also initialise the number of ships destroyed variable to 0.

**void BoardImpl::clear()**

To clear the board, it traverses the whole board (2D array) in for loops to set every element of the board as ‘.’.

**void BoardImpl::block()**

To block exactly half the board, it sets up a variable to store the total number of points on the board, and calculates this by multiplying the number of rows in the game with the number of columns in the game. It also starts a counter (totalBlocked) to store the number of total elements blocked. Next, using a while loop with the condition that as long as totalBlocked is less than the total number of cells in the game divided by two, then it generates a random point on the board (using the randInt function on the total number of points), checks if the point is valid and if it is empty i.e ‘.’ and if yes, then and blocks it (using ‘#’). It also increments the totalBlocked counter.

**void BoardImpl::Unblock(const Game& g)**

To unblock the board, we use for loops to traverse the board, and if the element of the board is blocked i.e. has ‘#’, we replace it with ‘.’ setting the spot to be empty.

**bool BoardImpl::placeShip(Point topOrLeft, int shipId, Direction dir)**

This function places the ship on the board given the starting point, the shipId and the direction in which we have to place the ship.

First, we check if the provided shipId is valid or not by checking if the shipId exceeds the game’s nShips() function. This function is incremented every time a ship is added to the game and thus keeps track of all the shipIds. If the shipId is greater than or equal to the game’s nShips() function, then the function returns false.

Next, we initialise two variables shipLength and shipSymbol which store that particular ship’s length and symbol by calling game’s shipLength and shipSymbol function on the shipId.

We now check if the given point is valid or not by calling the game's isValid function, and if the point is not valid then the function returns false.

For each direction (HORIZONTAL or VERTICAL), we check if the ship can fit on the board, by comparing the length of the ship and the number of available spots, horizontally or vertically. If the ship length is greater than the available spots, then the ship cannot be placed and the function returns false.

We also chcek if the ship has already been placed on the board by iterating through the vector that stores all the ShipInfo’s on the board. If m\_id matches the shipId or if the m\_symbol matches the shipSymbol then the function returns false, as that shipId/shipSymbol has already been placed on the board.

If everything is fine, we place the ship by creating a variable of the type ShipInfo and assigning it is the appropriate symbol, id and number of hits(=0). We now place the ship in the direction specified by iterating through for loops to check if the board can fit the length of the ship. If yes, all the points are replaced by the symbol of the ship and the function returns true. Else, it returns false. For every ship placed successfully, we push the ship in the shipInfo vector.

**bool BoardImpl::unplaceShip(Point topOrLeft, int shipId, Direction dir)**

This function unplaces a ship from the board.

First, we check if the provided shipId is valid or not by checking if the shipId exceeds the game’s nShips() function. This function is incremented every time a ship is added to the game and thus keeps track of all the shipIds. If the shipId is greater than or equal to the game’s nShips() function, then the function returns false.

We now check if the given point is valid or not by calling the game's isValid function, and if the point is not valid then the function returns false.

Next, we initialise two variables shipLength and shipSymbol which store that particular ship’s length and symbol by calling game’s shipLength and shipSymbol function on the shipId.

Now, we check fo the specified direction(either HORIZONTAL or VERTICAL), if the ship is present there, by checking if all points in the direction starting from the point provided, have the ship symbol until the ship’s length. If yes, then all the points are replaced by ‘.’ indicating that they are empty and the function returns true. Else, it returns false. We also iterate through the shipInfo vector and find the element whose m\_id matches the shipId provided, and delete that entry.

**void BoardImpl::attack(Point p, bool& shotsHit, bool& shipDestroyed, int& shipId)**

This function is used to submit an attack against the board.

First, we check if the point p is valid by calling the game’s function: isValid(Point p). If the point is not valid, the function returns false.

We declare a variable shipChar and set it to the point in the board that we are attacking. If the point has previously been attacked, the function returns false.

If the point is not empty, then that means that there was a ship segment at that point. Thus we set the point to be ‘X’ and the variable shotHit to true. Otherwise, we set shotHit to false, and the point on the board as ‘o’ indicating a missed shot.

If the shotHit is true, then we iterate through the vector shipInfo and if the element’s symbol is the previously stored symbol in the variable symbolHit then we increment that ship’s m\_numOfHits by one and break out of the loop. We also check if the element’s numOfHits is equal to the ship length and if yes, then we store the element’s m\_id to shipId, set shipDestroyed to true and increment the m\_numOfShipsDestroyed by one.

If the point is attacked, then the function returns true.

**bool BoardImpl::allShipsDestroyed() const**

This function is used to check if all the ships on the board have been destroyed.

We just check if the m\_numOfShipDestroyed is equal to the shipInfo vector’s size, and if yes, all the ships have been destroyed.

***GameImpl Class***

**Private Variables Defined**

Variable to store number of rows on the board: int m\_rows

Variable to store the number of columns on the board: int m\_cols

Struct Ship to store each ships m\_length, m\_symbol, m\_name

Vector of Ship: vector<Ship> vShips

**GameImpl::GameImpl(int nRows, int nCols)**

The constructor initialises the number of rows and columns in the game. It checks if the nRows and nCols are greater than MAXROWS and MAXCOLS. If not, then it initialises m\_rows and m\_cols to nRows and nCols. Else, it initialises the m\_rows and m\_cols to MAXROWS and MAXCOLS.

**int GameImpl::addship(int length, char symbol, string name)**

This function adds a ship to the game.

If m\_rows and m\_cols are both less than length, then returns false.

If the symbol is ‘#’ then it prints that the character must not be used as a ship’s symbol and returns false.

It next checks if the symbol isascii, and is printable. If not, then the function returns false.

If everything is fine, then we create a variable newShip, and initialise its length, symbol and name and push it to the vector vShips, and return true.

**int GameImpl::nships() const**

For this function we can simply return the size of the vector vShips as it keeps track of all ships added.

**int GameImpl::shipLength(int shipId) const**

For this function, we check if the shipId is greater than nShips() i.e. number of ships in the game, and if yes we return -1. If not, we return the length of the element in vShips having the given shipId.

**int GameImpl::shipSymbol(int shipId) const**

For this function, we check if the shipId is greater than nShips() i.e. number of ships in the game, and if yes we return ‘X’. If not, we return the symbol of the element in vShips having the given shipId.

**int GameImpl::shipName(int shipId) const**

For this function, we check if the shipId is greater than nShips() i.e. number of ships in the game, and if yes we return “”. If not, we return the name of the element in vShips having the given shipId.

**Player\* GameImpl::play(Player\* p1, Player\* p2, Board& b1, Board& b2, bool shouldPause)**

This function sets the game play.

If the placeShips function for player p1, or p2, is false, then return nullptr. Next it sets a bool variable for p1’s turn to true. Now while, both the players have ships on their boards, if the shouldPause() function is true, then it waits for the player to press enter. If it is p1’s turn, it indicates that player’s turn by printing its name and the opponent's board. If the player isHuman(), then it displays the opponent's board. It prompts the user to enter the row and column for attack. If the player is not human then we don’t display the board. A point variable is set to the players, recommend attack function, and shotHit, shipDestroyed are set to false, and the shipId is set to 0. Calling the attack function on the opponent's board, and if it is true, then p1 record the attack result. If shotHit is true, then the variable hitType is set to the string hit something, and if shipDestroyed is true then the variable hitType is set to the string destroyed the shipName(shipId). If nothing, then the hitType is set to the string missed.

Now we print the name of the player and where they attacked and the result of the attack, and again display the board if the player is human. If the attack fails then we print that the player wasted a shot at the point.

Next, we do the same process for player 2.

We check if all the ships for either player’s board are destroyed, and if yes, then we indicate the other player as the winner.

***Player Class***

We create the class Timer for this to track the time elapsed.

***HumanPlayer Class***

This is a derived class of Player. It takes in input from the user and places the ships/attacks accordingly.

**HumanPlayer::HumanPlayer(string nm, const Game& g)**

The constructor initialises the name and g of the HumanPlayer.

**HumanPlayer::~HumanPlayer()**

This is the destructor for the HumanPlayer.

**bool HumanPlayer::placesShips(Board &b)**

This function places a ship in the board by taking in input from the user.

This first prints which ship the user must place. It prompts the user to enter a direction and sets the direction accordingly to HORIZONTAL or VERTICAL. If the input is for neither, then it prompts the user to enter the direction until they enter a valid character. It also prompts the user to enter a row and a column and keeps prompting the user for the same until the point is valid. If the point is valid but the ship cannot be placed there, then the function prints that the ship cannot be placed there and reprompts for point input.

If the point is valid and the ship can be placed the function returns true.

**bool HumanPlayer::isHuman() const**

This function simply returns true for the human player.

**Point HumanPlayer::recommendAttack()**

This function sets the point where the human wants to attack by taking inputs from the user.

It declares two int variables row and col, and calls the getLineWithTwoIntegers for these two variables. This function returns the point made by the row and col.

***MediocrePlayer Class***

This is a derived class of Player.

There is a 2D integer array which is initialised with 0, which represents the opponent’s board. If an attack is made, the position becomes 1. This algorithm prevents the player from taking a shot where one was already taken. There are 100 possible ints like this. MediocrePlayer also has an array that contains the shipId, arranged in an order from greatest to least. The MediocrePlayer traverses through the array, and places the largest ship first, the second largest and next and so on, to save time. There is also an integer variable state, which keeps track of the state. There are two possible states:

To attack random positions/ships

To hone into one particular ship

There are also some other variables, one which holds the point that was first attacked, one which holds the number of valid coordinates to get around the board.

***GoodPlayer Class***

This is a derived class of Player.

GoodPlayer has a boolean which is true if its smallest ship has length two or more.

There is a 2D integer array which is initialised with 0, which represents the opponent’s board. A missed attack is represented by 1. A hit but not a sink if represented by 2, and a sink is represented by 3. There are 100 possible ints like this. GoodPlayer has an array that contains the shipId, arranged in an order from greatest to least. The GoodPlayer traverses through the array, and places the ships on the board in a descending order according to their length. Such an order to place the ships saves time. There is also an integer variable state, which keeps track of the state and there are two possible states:

To attack random positions/ships

To hone into one particular ship

There are also some other variables, one which holds the point that was first attacked, one which holds the number of valid coordinates to get around the board.