**Izak Bunda - 305783387 – Project 3 – Due WEDNESDAY MAY 18, 2022 AT 11PM**

**DESIGN**

The board itself is created using a 2-D array that is set to the MAXROWS and MAXCOLS. This made sense because it was not going to be any greater than MAXROWS and MAXCOLS.

I created a struct for Ship which stores the shipId, length, symbol, and name of a ship. To store these ships, I created a vector of pointers to the Ship class. This made more sense to implement with a vector since the number of ships added varied from each game. It was more efficient to do it this way compared to using an array.

Both MediocrePlayer and GoodPlayer used two vectors called possibleShots and shotsTaken. possibleShots kept the possible shots that would be taken in the cross generated if an attack hit a ship but not destroy it. I could have simply generated an array with a maximum of 16 spots for MediocrePlayer or 4 spots for GoodPlayer, but I used vector’s push\_back(), front(), erase(), and clear() a lot of times which is a lot more efficient than repeatedly iterating through each of the items in an array.

GoodPlayer had an additional vector called everyOtherShots which simply stored the checkerboard pattern spots to attack in order.

I will say, that as the first time where I really used vectors, this was somewhat challenging. Most, if not all of my issues which took hours and hours of debugging–were caused by a vector not being properly serviced ( a value is not erased, a value is not pushed back) or not cleared after a certain condition is met. But otherwise, the built in vector functions were very useful and made the implementation of MediocrePlayer and GoodPlayer very succinct and clear.

**GOODPLAYER DESIGN**

Firstly, GoodPlayer places it ships in the same way every time. I did a quick google search and fell into a deep rabbit hole of the best methods to place a ships in a standard BattleShip game. I ultimately decided on this one (pictured in the pseudo code).

A picture containing text

Description automatically generatedThe way a GoodPlayer recommends an attack is from a checkerboard pattern initialized at the first round of the game. Then for the rest of the game, recommendAttack goes in order of the checkerboard. If it hits a ship and does not destroy it, then similarly to MediocrePlayer, it also generates a cross of possibleShots to attack. But instead of a cross of 16 shots, four at each direction, GoodPlayer only creates a cross of 4 shots only one at each direction. This is because with the checkerboard pattern, all shots will be eventually hit using the cross of one (see picture on the right).

The spot highlighted in red is an example of an attack recommened by the checkerboard and notice how the cross of four will always hit the shots that are NOT recommended by the cherboard.

Essentially, GoodPlayer is just the lovechild of MediocrePlayer and AwfulPlayer. It inherits MediocrePlayer’s cross, but makes it shorter and it inherits AwfulPlayer’s order of attack, but it just skips every other step.

The result of this simple change is a winrate of 85%+ versus the MediocrePlayer

**PSEUDO CODE**

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

Store each player’s number of ships in variables

Place each player’s ships on their respective board

If either of the player’s ships aren’t placed, then immediately return a nullptr

// otherwise, the game starts…

Do-while the all the ships are not destroyed in both boards…

Declare two Player pointers – winner and loser

// Player 1’s attack:

Call Player 1’s respective recommendAttack function and store that in a Point variable

If it is a valid attack…

Check if it did not hit a spot

Check if it hit a spot but did not destroy a ship

Check if it hit a spot and destroyed a ship

Otherwise, the attack is invalid.

Then record Player 1’s attack using recordAttackResult

And also using Player 2’s recordOpponentAttack

Display the current status of the board

Pause the game if shouldPause is true

Check if all ships were destroyed in both boards and point the winner and loser pointer to the appropriate player

// Player 2’s attack:

Call Player 2’s respective recommendAttack function and store that in a Point variable

If it is a valid attack…

Check if it did not hit a spot

Check if it hit a spot but did not destroy a ship

Check if it hit a spot and destroyed a ship

Otherwise, the attack is invalid.

Then record Player 2’s attack using recordAttackResult

And also using Player 1’s recordOpponentAttack

Display the current status of the board

Pause the game if shouldPause is true

Check if all ships were destroyed in both boards and point the winner and loser pointer to the appropriate player

If it gets through the do-while and does not return, then return a nullptr

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

For every point in each row

For every point in each column

Put a dot ‘.’

**void BoardImpl::clear();**

For every point in each row

For every point in each column

Put a dot ‘.’

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

return false if shipId is invalid

// For horizontal and vertical

Check if it will go outside the borders

Check if topOrLeft is a valid point

Check if it will go to a spot that is already taken by another ship

Check that the ship with that ship ID has previously been placed on this Board and not yet been unplaced since its most recent placement

Then actually place it on the board

For the entire length of the ship, replace the Point with its corresponding symbol

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

checks if the shipId is invalid

checks if the entirety of the ship is actually there

Then it replaces the entirety of the ship with ‘.’s

**bool BoardImpl::attack(Point p, bool& shotHit, bool& shipDestroyed, int& shipId);**

initialize a tempShipId and tempSymbol to the symbol at Point p

Checks if the attack is a valid point on the board

If not, the set shotHit to false, shipDestroyed to false, and return false

Check if that point has already been attacked

If not, the set shotHit to false, shipDestroyed to false, and return false

Check if the point is even a ship/ship segment

If not, the set shotHit to false, shipDestroyed to false, and return false

Go through the entire board to determine the shipId at that point

Try to match the tempSymbol to the symbol of an already declared ship

// everything above was just preparation, the attack starts now

Check that the point is NOT either a ‘X’ or ‘o’

If it is a ‘.’ Then mark it with ‘o’ and set shotHit to false

Otherwise, mark it with ‘X’ and shotHit to true

If so, then set shotHit to false

Then, loop through the board and try to see if the tempSymbol is still present on the board

If not, then the whole ship is destroyed

If so, then do nothing

If it gets through the whole thing, then return true

**MEDIOCRE PLAYER**

**bool placeHelper( int nShips, const Game& g, Board& b ); // the recursive one**

Set bool placedShips to true

If nShips is greater than or equal to zero // this is what is going to keep the recursion from infinitely

For every point on the board…

Try placing the ship horizontally

If this works…

Then recursively call placeHelper with nShips-1

And at the limit of the depth of recursion, return true

Otherwise try placing it vertically

Unplace the ship from before // this is backtracking

If it works vertically, then call placeHelper with nShips-1

And at the limit of the depth of recursion, return true

If this still doesn’t work

Unplace the ship and then move on to the next point (move on to the next iteration of the for loop.

Return the result of the (recursive) call

**bool MediocrePlayer::placeShips(Board& b);**

while it doesn’t exceed 50 times and all ships have not been placed

block the board using block()

call placeHelper

unblock the board

**Point MediocrePlayer::recommendAttack();**

Initialize a bool that keeps track if a shot has already been taken

If in State 1

While the shot has not been taken

Generate a random point on the board

If the vector which keeps track of the shots already taken is NOT empty

Check if this random point is in that vector

When it breaks out of the vector, it means that the shot has not been taken before

Add this random point to the shots taken

And return this point as the recommended point

If in State 2

If the vector which keeps the possible shots on the cross of a maximum of 16 shots is empty

Then for every cardinal direction

Add a maximum of four spots to the vector of possible shots IF they haven’t been already attacked before

If the vector is not empty… which means there are shots to attack

Then randomly pick from the available shots in the cross

Add it to the vector of shots taken

Erase it from the vector of possible shots

And recommend that point

**void MediocrePlayer::recordAttackResult(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId);**

If in State 1

If a shot is not hit, stay in state 1

If a shot is hit and a ship is destroyed, stay in state 1

If a shot is hit and a ship is not destroyed, go to state 2

If the vector of possible shots on the cross is empty, then save that point which will be used to generate the new cross

If in State 2

If the vector of possible shots is empty, then return to state 1

If a shot is not hit, stay in state 2

If a shot is hit and a ship is not destroyed, stay in state 2

If a shot is hit and a ship is destroyed, go back to state 1

Clear the vector of possible shots (the cross)

**GOOD PLAYER**

**bool GoodPlayer::placeShips(Board& b);**

Place the ships the same way every time:

Table

Description automatically generated

If a standard game is not called, then use the same placeShip as MediocrePlayer

**Point GoodPlayer::recommendAttack();**

If in State 1

If the vector which keeps track of a checkerboard pattern of shots to hit is empty

Generate a vector of a checkerboard pattern of shots to hit

\*\*\* this will only get called once, that’s why it is in State 1 and only called if it’s empty \*\*\*

Otherwise

Recommend the point in the front of this vector ( it will go in order )

Add that point to the shots taken vector

Erase that shot from the checkerboard vector

If in State 2

If the vector which keeps the possible shots on the cross of a maximum of 4 shots is empty

Then for every cardinal direction

Add a maximum of four spots to the vector of possible shots IF they haven’t been already attacked before

If the vector is not empty… which means there are shots to attack

Then randomly pick from the available shots in the cross

Add it to the vector of shots taken

Erase it from the vector of possible shots

And recommend that point

**void GoodPlayer::recordAttackResult(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId);**

If in State 1

If a shot is not hit, stay in state 1

If a shot is hit and a ship is destroyed, stay in state 1

If a shot is hit and a ship is not destroyed, go to state 2

If the vector of possible shots on the cross is empty, then save that point which will be used to generate the new cross

If in State 2

If the vector of possible shots is empty, then return to state 1

If a shot is not hit, stay in state 2

If a shot is hit and a ship is not destroyed, stay in state 2

If a shot is hit and a ship is destroyed, go back to state 1

Clear the vector of possible shots (the cross)